Université des Sciences et Technologies de Lille
Laboratoire d'Informatique Fondamentale de Lille — UMR 8022

THESE

pour obtenir le titre de
Docteur en Sciences
de l'UNIVERSITE des Sciences et Technologies de Lille
Discipline: Informatique

présentée et soutenue par
Dorina GHINDICI

Information flow analysis for embedded systems: from practical to theoretical aspects

Directeur de thèse: **Isabelle SIMPLOT-RYL**
Co-directeur: **Gilles GRIMAUD**
soutenue le 4 décembre 2008

Jury:

Presidént		
	Mireille Clerbout	LIFL, Univ. Lille 1
Rapporteurs		
	Yves Bertot	INRIA, Sophia-Antipolis
	Nadia Tawbi	Univ. Laval, Canada
Examinateurs		
	Bertil Folliot	LIP6, Univ. Paris 6
	David Pichardie	IRISA/INRIA Rennes - Bretagne Atlantique

Abstract

This work aims at providing a solution to confidentiality issues in multiapplicative systems: ensuring security for an application running on small and autonomous systems by verifying information flow properties at deployment time. Existing work on information flow does not scale to small open systems due to resources limitations and due to lack of modularity, which is essential in a dynamically evolving environment. In order to provide a complete solution, we address both practical and theoretical aspects. We first propose a model and a tool dedicated to small open systems running Java bytecode, with support for inheritance and override. Our approach is modular, hence the verification is incremental and is performed on the target device, the only place where the security can be guaranteed. To our knowledge, it is the first information flow verifier for embedded systems. To prove its usability, we ran different experiments and we tested the tool in several contexts. Secondly, we tackle the information flow issue from a theoretical point of view. We propose a formal model, based on abstract memory graphs; an abstract memory graph is a points-to graph extended with nodes abstracting input values of primitive type and flows arising from implicit flow. Our construction is proved correct with respect to non-interference. Contrary to most type-based approaches, our abstract memory graph is built independently on any security level knowledge. Information flow is checked *a posteriori* by labeling abstract memory graphs security levels.

Résumé

Nos travaux ont pour but de fournir une solution aux problèmes de confidentialité dans les systèmes multi-applicatifs: assurer la sécurité des applications dédiées aux systèmes portables et autonomes en vérifiant des propriétés de sécurité en termes de flot d'information au moment du chargement des applications, contrairement aux travaux existants qui ne sont ni modulaires ni dynamiques. Afin de fournir une solution complète, nous avons traité les aspects à la fois pratiques et théoriques du problème. Dans un premier temps, nous proposons un modèle et un outil adaptés aux contraintes inhérentes aux systèmes embarqués. Notre approche est modulaire et supporte l'héritage et la surcharge. La vérification est donc incrémentale et s'effectue sur le système cible, le seul endroit ou la sécurité peut être garantie. Il s'agit, à notre connaissance, du premier vérifieur embarqué pour l'analyse de flot d'information. Afin de prouver l'utilité pratique de notre modèle, nous avons mené des expérimentations et nous avons testé l'outil dans des contextes différents. Dans un deuxième temps, nous traitons les aspects théoriques: nous proposons un modèle formel basé sur des graphes de l'abstraction de la mémoire. Un graphe de l'abstraction de la mémoire est un graphe ≪points-to≫ prolongé par des noeuds de type primitif et par des liens issus de flots implicites. Notre construction est prouvée correcte par un théorème de non-interférence. De plus, les politiques de sécurité ne nécessitent pas d'être connues pendant l'analyse : le flot d'information est vérifié a posteriori en étiquetant le graphe de l'abstraction de la mémoire avec des niveaux de sécurité.

Contents

1 Introduction

Computer systems handle a considerable amount of data carrying sensitive information that should be protected from malicious users. Programs running on such systems may access data either to perform computations or to transmit it over an output channel. Thus they can violate the security of sensitive data either by releasing it to unauthorized users or by modifying it. In order to prevent such situations, tracing data manipulation throughout programs is mandatory.

Mechanisms such as access control or cryptography provide some protection for accessing and modifying confidential data, but they are insufficient to regulate data propagation once the information has been released. For example, an authorized program can read sensitive data and write it to a location accessible by an unauthorized program. These mechanisms guarantee security only when it comes to completely trusted programs. The illegal transfer of information resulting from interaction between untrusted programs may be circumvented by analyzing and controlling the underlying information flow.

Information flow analysis [SM03] consists of statically analyzing the code of a program in order to detect illicit data manipulations. Concretely, data manipulated by programs (*e.g.*, objects, parameters) are tagged with security labels and all information flows are traced. Usually, information flow is associated with non-interference [GM82] which prevents all information flows from sensitive data to non-sensitive data.

A considerable amount of work has been devoted to the design of methods for analysing information flow, but *despite their long history and appealing strengths, information-flow mechanisms have not yet been successfully applied in practice*[Zda04]. According to Steve Zdancewic [Zda04], *the real challenge for information-flow security is demonstrating that all of this theory and these language designs are actually useful—we need to apply the technology to real problems, or, failing that, understand why such an appealing technology is not useful in practice*.

In this thesis, we tackle information flow from a practical point of view. We give solutions for some challenges such as integration with existing security mechanisms, security verification integration within modern deployment schemes, the inadequacy of strict noninterference, and the difficulty of managing security policies.

Embedded information flow verification First, we address challenges raised by small open environments such as smart cards, PDA, mobile phones, etc. by proposing an embedded information flow model adapted to targeted platforms. As ubiquitous computing is evolving towards post issuance and automatic execution of untrusted code, the security issue is getting more and more intense with multiapplication platforms, dynamic code downloading and constant growth of the complexity. Integrating information flow certification for mobile code in Java-enabled small open embedded systems requires, at least, *(i)* separation of code and security policies and *(ii)* certification at loading time.

In order to perform the certification at loading time, we use type inference and a technique in the style of Necula's proof carrying code (PCC) [RR98, Nec97]. This verifier scheme is an extension of to the Java bytecode lightweight verification [RR98]. The idea is to separate the verification process in two parts. An off card part (*the prover*), that infers a certificate and some "proof" indicating that the code is correct with respect to the security policy and an on card part (*the verifier*), that uses the proof to certify the correctness of downloaded code. To our knowledge, this is the first implementation of an embedded information flow verifier. Experimental results show that our verifier can be efficiently used for our target applications and hence, it can be successfully applied in practice.

Defining security policies Information flow analysis does not guarantee security by itself: it is a powerful mechanism that can be exploited to implement the desired security policies. The difficulty is to ensure that local checks (mechanisms) actually implement the global security policy of the system (the security policy of each software unit in the system). Information flow mechanisms are too coarse to express desired policies, thus one of their common pitfalls is to define and verify complex policies, reflecting real attacking scenarios.

Hence, to enhance the power of the embedded information flow model and its usability, we enrich it with security policies describing allowed collaborations between different software units in a program. While the embedded information flow model enforces non-interference, the security policies allow a form of relaxation and they have enough power to express the desired security in a multiapplication small open system. In order to fit with the paradigm of Java Virtual Machine, the security policies are not mixed with the code, and they are verified at loading time. Hence, defining a new security policy does not require reanalyzing the entire system.

Integrating information flow certification in a framework Developping an information flow verifier and defining security policies are not enough to make information flow appealing to real users. A key issue in information flow security, and even computer security in general, is not only detecting and preventing attacks, but also *(i)* helping the developer to build safe applications, components and *(ii)* increasing user confidence in software products. In most current approaches to information flow security of software units, security flaws are fixed only after the full implementation of the software units or only after they have been exploited. To make information flow security more appealing to the software industry, more proactive and durable security solutions are needed, by addressing security requirements throughout the software system lifecycle, including requirements and design specification, testing, and maintenance phases. Appropriate information flow security analysis techniques must be used for each of these phases. Hence, to make information flow appealing in practice, we make a step further by integrating the embedded information flow model in a security analysis framework, which combines a quantitative design security analysis technique (developed at the University of Victoria, Canada), with the embedded verifier.

Proving the soundness of an information flow model In order to have a complete model, both usefull in practice and proved correct, the next step is to prove the soundness of the embedded information flow model. To prove the correctness of the analysis, we define a new analysis, which is more general as it does not need to consider the approximations imposed in the embedded model by the constraint resources. The new analysis is more precise than the previous one while keeping the main features: adapted to open systems, separation of concerns. The information flow is described by means of abstract memory graphs (AMGs). Security policies are enforced by labeling *a posteriori* the graph with security levels. As in the embedded information flow model, changing the security levels does not require reanalyzing the entire system. The embedded information flow model can be recovered as an approximation of the new analysis.

Throughout our approach, we tried to make usable our work on real Java software. This lead to the development of a functional prototype, available online[1], and to several practical experiments, especially on Java applications for mobile phones. We begin this thesis by presenting a state of the art for information flow security in the context of mobile code.

[1]`http://www.lifl.fr/~ghindici/STAN`

Introduction (french)

La présence de petits appareils tels que les téléphones mobiles, cartes à puce, PDA, ou encore GPS, pilotés par un *système embarqué* est de plus en plus forte. Dans un avenir proche, ces différentes applications ne seront plus exécutées par différents systèmes dédiés à celles-ci mais installées sur le même système hôte. Cette coexistence d'applications sur le même système pose un certains nombre de problèmes de sécurité. Les informations confidentielles (codes PIN, dossiers médicaux, etc) manipulées par ces applications doivent être accédées uniquement par des ayant-droits (programmes et/ou utilisateurs), ce qui nécessite une protection adéquate du système embarqué contre d'éventuelles attaques, délibérées ou non.

Les mécanismes de contrôle d'accès tels que l'identification par mot de passe permettent de limiter l'accès aux données sensibles aux utilisateurs/programmes autorisés. Cependant, cela ne permet nullement d'empêcher une application autorisée de divulguer une information secrète. Or une telle fuite d'information est possible et peut venir d'une mauvaise implémentation/conception du code lui-même, ou bien d'un code malfaisant. Pour détecter ces fuites, une analyse de flot d'information est nécessaire: les données sont étiquetées avec des niveaux de securité, et leur propagation lors de l'exécution d'un programme est étudiée.

Nous nous intéressons donc au flot des données au sein des applications. La vérification de cette propagation des données sensibles permet de garantir que l'utilisation de celles-ci reste correcte, et ne permet pas leur divulgation. Afin de fournir une solution complète, nous avons traité les aspects à la fois pratique et théorique du problème.

Les contraintes des systèmes portables (puissance de calcul, capacité mémoire) ne permettent pas la vérification de propriétés de sécurité directement sur le matériel cible. Pour rendre la vérification de flot d'information utilisable en pratique, nous avons proposé un schéma d'analyse en deux étapes: (1) une analyse externe [GGSR06a], qui peut être effectuée sur une machine assez puissante et qui réalise un calcul du type, et (2) une analyse embarquée [GGSR07] qui vérifie, au moment du chargement, l'exactitude de calcul du type effectué à l'extérieur. Le modèle présenté respecte la non-interférence: en considérant deux niveaux de sécurité, *secret* et *public*, un programme est sûr par rapport à la non-interference si, à partir des données étiquetées *public* en sortie, nous ne pouvons déduire aucune information sur les données étiquetées *secret*.

La non-interférence est une proprieté puissante qui n'autorise que les flots d'information de *secret* vers *secret*. Pourtant, la non-interférence ne fait aucune distinction entre les sources de données *secret*. Les politiques de sécurité définies avec cette relation sont trop restrictives. De plus, dans la majorité des cas et en particulier dans le cas des systèmes multi-applicatifs, elles ne sont pas assez expressives. Pour offrir une solution à ce problème, nous raffinons la non-interférence en définissant des politiques de sécurité plus complexes, qui prennent en considération les sources de données *secret* [GSR08]. Pour échapper à la rigueur de la non-interférence, nous avons défini un langage dédié qui décrit les flots d'information autorisés dans un programme et nous montrons comment les politiques définies avec ce langage sont verifiées dans le schéma d'analyse presenté précédemment.

En collaboration avec l'université de Victoria, Canada, nous avons étudié l'utilisabilité de notre solution sur un cas d'étude [GGSR$^+$06b], qui consiste en l'utilisation d'une carte à puce réunissant plusieurs applications médicales qui doivent respecter les politiques de sécurité définies par le BMA (British Medical Association).

Ensuite, nous avons traité les aspects théoriques en présentant une analyse de flot d'information pour le bytecode Java [GSRT09, GSRT07]. Notre approche consiste à calculer, pour différents points du programme, un graphe de dépendances qui représente l'influence que les valeurs en entrée d'une méthode ont sur les sorties. Ce calcul inclut une analyse de pointeurs (illustrant les dépendances entre objets) à laquelle sont ajoutées les dépendances issues des données de type primitif et du flot de contrôle du programme. La construction de notre graphe est prouvée correcte par un théorème de non-interférence qui énonce qu'une valeur de sortie n'est pas liée à une valeur d'entrée dans le graphe de dépendances si la valeur de sortie ne change pas lorsque la valeur d'entrée varie. À l'inverse de beaucoup de techniques basées sur des systèmes de types, notre approche ne nécessite pas de connaître la politique de sécurité lors du calcul du graphe : le respect d'une politique de ≪sécurité≫ en termes de flot d'information est vérifié en étiquetant ≪a posteriori≫ le graphe de dépendances avec des niveaux de sécurité.

2 Preliminaries and state of the art

Contents

This dissertation involves several areas of reseach, including confidentiality, information flow, type inference, Java security in the presence of dynamic downloading, software verification. We first define confidentiality, then we introduce information flow security (*e.g.*, challenges, approaches, solutions). After that, we give a large view on application security for a Java Virtual Machine, and we concentrate on information flow threats for multiapplication systems. We conclude with our contributions and document structure.

2.1 Protecting data confidentiality

Computer security has been an important problem since computers have existed. In a rapidly evolving computing infrastructure, where the security threats increase constantly, offering more and more security-sensitive services is an essential goal. One of the main challenges in computer security is to formally define security policies and to develop mechanisms which guarantee the legitimate use of sensitive data.

Computer systems handle a considerable amount of data carrying sensitive information that should be protected from malicious users. Programs running on such systems may access data either to perform computations or to transmit it over an output channel. Thus they can violate the security of sensitive data either by releasing it to unauthorized users or by modifying it.

Two major data security properties have been identified:

```
p = false;
if(s)
   nop;
else
   p = true;
```

Figure 2.1: Implicit flow example

Confidentiality ensures that sensitive data is read only by authorized users.

Integrity ensures that sensitive data is modified only by authorized users.

This thesis focuses on data confidentiality. Since integrity and confidentiality go hand in hand, deducting the integrity properties from the confidentiality properties is straightforward.

Access control The concept of confidentiality in computer systems is usually associated with discretionary access control [DPS03], which consists of annotating each data/resource with a label defining which users/programs may read/write it. An identification mechanism allows dynamic monitoring of each access and unauthorized access is rejected. Even if it provides protection for accessing and modifying confidential data, access control is insufficient to regulate data propagation once the information has been released. For example, an authorized program can read sensitive data and write it to a location accessible by an unauthorized program. This mechanism guarantees security only when it comes to completely trusted programs.

Mandatory access control In order to address data propagation, access control techniques have been gradually refined, converging to *mandatory access control* [Den82]. This mechanism is a transitive extension of access control, as it enlarges the scope of access restrictions from the point where data has been released to any point where data is being used. To achieve the extended access restrictions, data is labeled with security levels; data propagation is regulated by the system, which augments normal data computation within programs with security label computations.

In this context, Bell and LaPadula [BL76] proposed a model which focuses on data confidentiality and access to classified information. This security model is characterized by the phrase: *no read up, no write down*. In other words, users can view content only at or below their own security level and can create content only at or above their own security level.

The Bell-LaPadula model addresses only data confidentiality. To overcome this weakness, the Biba model [Bib77], which describes rules for the protection of data integrity, has been developed. In contrast with the Bell-LaPadula model, the phrase which characterizes the model is: *no write up, no read down*, meaning that users can only create content at or below their own integrity level and they can only view content at or above their own integrity level.

The most common approach to mandatory access control implementation is dynamic enforcement. Fenton's Data Mark Machine [Fen74] is one of the earliest abstract model that used the concept of dynamic enforcement of confidentiality policies. But, besides the obvious runtime overhead (in execution time and occupied memory), the weakness of dynamic enforcement mechanisms consists of predicting implicit information flows which arise from the control structure of a program, as opposed to explicit flows which are generated by direct assignment of confidential data to public variables. Handling implicit flows correctly is essential for computer security, as they may disclose data from the execution or non-execution of some program statements. Let us consider the example in Figure 2.1. where `s` contains sensitive data and `p` insensitive data. The statement `p = true` is executed only if `s` is not `true`. Hence, the insecurity arises from the fact that an observer of the public value `p` may induce information about the sensitive data `s`. Dynamic enforcement for *mandatory access control* never detects an illegal data flow when executing the program with `s = true`, but an observer could infer information from the final value of `p` and hence from the non-execution of the statement `p = true`.

2.2 Information flow

Another more promising solution consists of analysing *a priori* the system or only a part of it (some programs) to which we wish to grant some privileges on sensitive data, in order to determine how it uses these privileges. If the static analysis guarantees that the behaviour of the system is safe or acceptable, then the system is granted the privileges without any restriction and it is executed without any further verification. If the analysis fails, its execution is rejected. This kind of mechanism for enforcing security is called *information flow analysis* and unlike the previous mechanisms, it does not rely on any trust relation. The correctness proof is sufficient to guarantee security.

Information flow analysis is a language based technique as it uses language semantics and static analysis to enforce security properties. Due to its static character, this approach presents two major advantages. First, the analysis does not introduce any runtime overhead; the execution of a certified system is similar to the execution of a system that does not perform any verification. Secondly, the analysis can take into account all possible execution paths, instead of the single execution path followed by a dynamic mechanism. Contrary to mandatory access control, an information flow analysis detects implicit flows, as the one describes in Figure 2.1. Hence, information flow analysis increases precision by correctly handling implicit flows.

2.2.1 Channels

Information flow analysis aims at enforcing confidentiality policies by controlling how data flows take place within computations of programs. Data flows occur in different forms, or *channels*. The most common channels, whose purpose is information transfer, are named *direct channels* or *direct flows*. Direct flow means direct value passing including assignment and argument or return value passing through method call. It is obvious that the statement `p = s+10;` generates a dependency of `p` on `s`, through direct flow.

Channels which are not intended to serve as media for information transfer are called *covert channels* [Lam73]. The most important covert channel is *implicit flow*, as already discussed for example in Figure 2.1. As the implicit flow has already showed, covert channels represent a real challenge in enforcing confidentiality properties. Other covert channels [SM03] include timing channels (leaks information through the time at which an event occurs), termination channels (leak information through the termination or nontermination of a program/statement), resource allocation (leak data through the exhaustion of a limited resource such as memory, files or hard disk space), power consumption, probabilistic channels, synchronization channels (in multithreaded environments) etc.

This thesis accounts for the most important type of channels: direct flows and implicit flows (excluding unchecked exceptions). To avoid data leaks from termination channels, we assume that all programs terminate. Synchronization channels and hence multithreading are partially supported. The other covert channels are left out of the scope of this document, as a model usually reflects limited aspects of the real system and hence it cannot capture all channels.

2.2.2 Information flow policies

In order to specify which information flows are allowed, data sources are labeled with security levels (in the same way as in mandatory access control). Data confidentiality is guaranteed if information

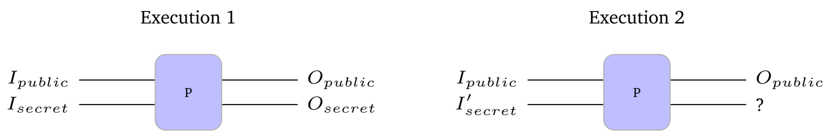


Figure 2.2: Non-interference

is read only by subjects with an allowed security level. Hence, an ordering relation must be given for security levels. Information is allowed to flow from a resource to another if the source has a lower security level than the target. Complementary, to ensure data integrity, data of a resource is allowed to flow to another one if it has a higher security level than the target.

Using the order relation, information flow policies can be described over a lattice of security levels. This model was initially introduced by Denning [Den76]. The model defined in this thesis follows the same approach, *i.e.*, information is allowed to flow in only one direction, from lower to higher security levels. Some applications need complex security lattice but usually, for sake of simplicity, the security lattice is made of two security levels, secret (or high) and public (or low). The order relation is obvious: low $\leq$ high, hence information flow is allowed to flow from public to secret, but not in the opposite direction.

The decentralized label model [ML97, ML98] proposes a new type of information flow policies: data belongs to owners (one or many) who decide how the information can be disseminated. Again data is labeled with security levels, the difference from the lattice model being the definition of a security level: in the decentralized model, the security level is a set of pairs with owners and readers. A data can have multiple owners, and each owner can define its own security policy. Data is released only if all owners agree.

2.2.3 Non-interference

The fact that a program is secure with respect to information flow is formally captured by the notion of *non-interference* [GM82], which states that public, low outputs should not depend on secret, high inputs. If we represent the execution of a program P as a box with public and secret inputs and outputs, we define non-interference as *changes in secret inputs do not cause changes in public outputs*. Figure 2.2 shows two executions of a program which satisfies non-interference. Since the only difference in the two executions is the initial value of secret inputs (I_{secret} and I'_{secret}), the output value of public variables O_{public} cannot be different. However, the secret output can be changed, since non-interference does not put any restrictions on the output of secret, high variables.

2.3 Embedded security in multiapplication environments

Information flow security represent a real threat in all environments handling confidential data, including personal devices storing sensitive data (*e.g.*, smartcards, mobile phones, PDAs). In this thesis, we study information flow in the context of such devices, offering multiple and variable services.

2.3.1 Information flow in multiapplication systems

Perhaps the greatest appeal of open systems (and especially smart card technology) is their ability to consolidate multiple applications in a single, dynamic device. These devices simplify life for end-users, by replacing many cards for payment and other transactions (loyalty card, ID card, medical card, etc). Multiapplication cards are beneficial for issuers as well, because they create unique marketing opportunities. Because these cards deliver such highly personalized applications, their perceived value among end-users is much higher and helps build stronger than average customer loyalty.

These cards also raise special challenges; one of the most sensitive domain which must adapt to these cards is security. Information flow and security in general has always been a big concern for small embedded systems, but the issue is getting more intense with multiapplication platforms, dynamic code downloading and the constant growth of the complexity. For example, a malicious applet running on your mobile phone or smart card can do a lot of harm: it can disclose confidential information, financial data, address book, social security number and medical files, etc.

Moreover, if the system runs multiple software units, possibly untrusted, which share code (*e.g.*, API) or data (*e.g.*, collaborative applications), then the underlying information flow [Gir99] must be verified in order to ensure data confidentiality. The security threat may originate from the code itself or from code shared with some malicious software which cannot be trusted.

2.3.2 Dynamic code downloading

Ubiquitous computing is evolving towards automatic execution of untrusted code. The main idea is to enhance experience by automatically downloading and executing code embedded in various files. Automatic execution of untrusted code has become popular with Java Web applets, and recently in desktop computing. Nowadays, applet-style downloading of software units is also becoming popular in certain embedded systems like mobile phones (Java “midlets” for MIDP-enabled phones), smart cards (JAVACARD), etc.

Open embedded systems like JAVACARD provide developers with an opportunity to rapidly develop software units. Moreover, they offer the possibility to download software units into the system *a posteriori*. The main drawback with this kind of systems is the risk to download a hostile software unit that may exploit a faulty implementation module of the platform.

Hence, allowing dynamic downloading dictates the need to verify that the incoming applet respects desired security properties, *e.g.*, type correctness, data confidentiality. In the case of Java mobile code, compiled code (JVM bytecode) is downloaded through an unsecured channel. Thus, similar to the bytecode verifier [RR98], the information flow certification must be done onboard, preferably at loading time, in order to avoid runtime overhead. This way the security is enforced by the virtual machine itself. Moreover, we argue that, in a runtime environment able to support the deployment of several software units provided by self-sufficient issuers, the compilation of each software unit must be done without any knowledge about the other potential software. In this case, and because each runtime environment can embed a distinct set of software units, the only place where the whole software units can be checked as a whole is the runtime environment where they are run.

2.3.3 Motivating examples

In this section, we present two examples with security requirements which, due to their dynamic environment, cannot be satisfied using existing techniques. The example focuses on multiapplication smart cards supporting post issuance (*i.e.*, dynamic code downloading).

Loyalty card

The LoyaltyCard is a multiapplication Java-enabled smart card composed of four loyalty software units: two air companies (FlyFrance, FlyMaroc), a car rental company (MHz) and a hotel (Illtone). The software units are installed dynamically, according to the deployment diagram in Figure 2.3. Three of these software units form a group of partners. Partners can collaborate and share loyalty

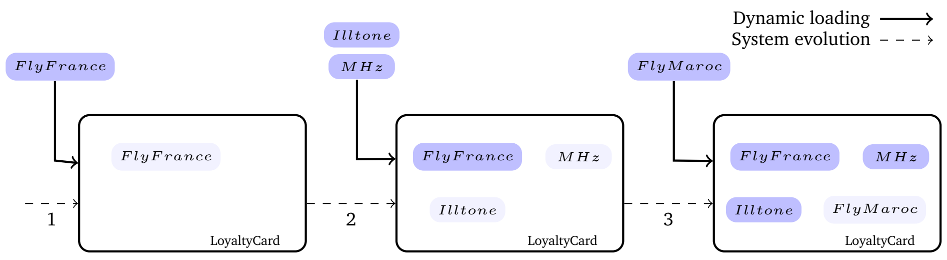


Figure 2.3: LoyaltyCard: Deployment process

points while collaborations with external software units, as depicted in Figure 2.4, may lead to illegal flows of information.

According to the commercial agreement between the group of partners, part of FlyFrance points can be used to obtain MHz and Illtone loyalty points. Meantime, FlyFrance does not want FlyMaroc, a rival company, to learn any information about the status of its clients (*e.g.*, number of miles; status: gold, silver). But FlyMaroc also has an agreement with MHz and offers a discount, depending on the status of the MHz client. If asked by FlyMaroc, MHz returns not only its loyalty points, but also loyalty points of its partners (FlyFrance). FlyMaroc could obtain information about the FlyFrance loyalty points through MHz, as depicted in Figure 2.4. This way an illegal information flow would be established. Illtone also offers a discount for MHz clients but this time the flow of information is allowed as Illtone is one of the partners of FlyFrance.

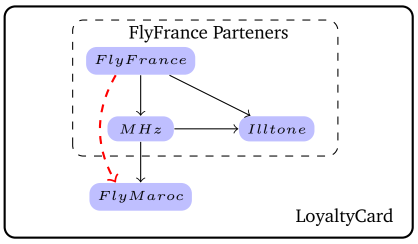


Figure 2.4: LoyaltyCard: Information flow in the deployed system

```
class FlyFrance {
  private int miles;
  [..]
  public void updates() {
    int i=0;
    for(;i<noLoyalties;i++)
      update(loyalties[i]);
  }
  void update(Loyalty l){
    l.update(miles);
  }
}
```

```
class MHz extends Loyalty{
  private int points;
  private int ppoints;
  [..]
  public void update(int p){
    this.ppoints += p;
  }
  public int getLevel() {
    if(points+ppoints>LGOLD)
      return GOLD;
    return SILVER;
  }
}
```

```
class FlyMaroc {
  private int discount;
  [..]
  void discount(MHz h){
    int level;
    level=h.getLevel();
    if(level == GOLD)
      discount = 20;
  }
}
```

Figure 2.5: Excerpt from the Java implementation of LoyaltyCard

We want to be able to show that the implementations of the FlyFrance, MHz, FlyMaroc and Illtone enforce information flow policies, *i.e.*, each program shares confidential data only with trusted software units.

Figure 2.5 shows an extract from the Java code of `FlyFrance`, `FlyMaroc` and `MHz` classes. The confidential data of `FlyFrance` is stored in the field `miles`. The method `update` in `FlyFrance` updates the points of its partners (`MHz` and `Illtone`). `MHz` stores the points of its partners in field `ppoints`. Method `MHz.getLevel()` returns the status of `MHz` based on `MHz` points and on partner points. This method is called by `FlyMaroc.discount` in order to offer a discount which leads to an unauthorized flow of information to a software unit untrusted by `FlyFrance`. The `MHz.getLevel()` method is also called by `Illtone`, but in this case, the flow of information is authorized as `FlyFrance` has an agreement with `Illtone`.

All these software units must communicate with remote software (*e.g.*, terminal, ATM); in order to secure the information exchange encryption mechanisms are used. Communication with the terminal is encrypted using an encryption algorithm of type `Cipher`. Some algorithms, extending the `Cipher` class, are provided in the initial system but other algorithms may be installed onboard *a posteriori*. Insecurity may arise from a malicious `Cipher` which overrides the encryption method in a malicious way: for example the encryption key could be copied in a shared location. Our aim is to be able to allow the installation of new software units only if they enforce security for the software units already installed.

Care card

The care card is a medical identification and information card which securely maintains electronic medical records of a specific patient who also is the holder of the card. A patient's record consists of two sections: prescription and treatment. The treatment section represents the core of the medical record in the sense that it carries the medical history of the patient. Only authorized users (applets) can read or modify the medical history.

The card is issued with an applet, the CardManager, which handles the medical record. Two other applets, Doctor and Nurse, are also installed and allow users to interact with the card. Doctors have access to medical history but they are not allowed to give it to other applets; nurses do not have access to medical history. In the same time doctors and nurses share data as doctors may write care indications to nurses.

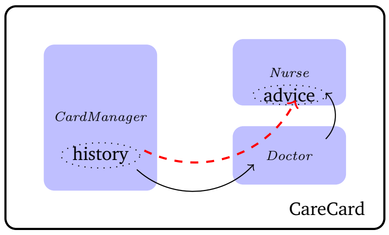


Figure 2.6: CareCard example

Figure 2.6 summarizes the information flow between applets. Dashed arrows indicate illegal information flow. The security threat is the following: the doctor applet can copy the medical history from the CardManager applet to the advice field of nurses, which can send it through the terminal to unauthorized users.

Consider now the following scenario: the patient joins a health insurance company and downloads the corresponding applet. The health insurance company has access to expenses but not to medical history. The goal is to allow installation of new applications, such as the insurance company, while enforcing security for already installed applets and the new applet.

2.3.4 Definitions and challenges

The examples highlight the following definitions:

Software unit a coherent code unit compiled simultaneously by its producer in order to be deployed in a multiapplication environment.

Open system a set of software units which can be extended dynamically with new software units *a posteriori*.

The LoyaltyCard is an *open system* comprising four *software units*, while the CareCard is issued with a software unit (CardManager) and extended with two other software units (Nurse and Doctor).

From the LoyaltyCard example, we identify two security challenges for open systems:

Consistency guarantees the security of already installed software units while installing a new software unit,

Incrementality certifies the secure information flow of a newly deployed software unit, according to its security policy.

To prevent leaks, analyzing the underlying information flow is mandatory. Due to its complexity, few models and implementations addressing information flow issues for small embedded systems exist, and none of them offers support for post issuance. A few operating systems have been proposed to manage multiapplication open platforms [MM08], but Java and Java Card technology is the clear leader. The example described above is a real example with security risks of illegal information flow. Therefore, in this thesis we address information flow issues in the context of open systems running on a Java Virtual Machine.

2.3.5 Java Virtual Machine background

Java is a high-level language which gives the ability to write programs that can run on a variety of platforms. Due to its portability and support for dynamic code loading, in the last decades Java has emerged itself as the most common language for small systems. Java programs are compiled into a form called Java bytecode, placed into class (.class) files, which are executed by the Java Virtual Machine (JVM) [LY99]. Thus the Java bytecode can be thought of as the machine language of the JVM.

To run a Java software unit the JVM needs to load the .class file containing the `main()` method of that software unit. The loading process is performed by a *class loader* which, besides the loaded class, loads various supporting classes such as the `Object` class inherited by all Java classes. A class loader loads one class at a time. Class loaders are responsible for importing binary data that define the running program's classes and interfaces. Actually, there may be more than one class loader inside a JVM, as the JVM has a flexible class loader architecture that allows a Java software unit to load classes in custom ways, by downloading class files across a network for example. Moreover, custom verifications, such as information flow security policies, can be performed at loading time.

Security for Java Because the class files of an applet are automatically downloaded (*e.g.*, when a user goes to the containing Web page in a browser), it is likely that a user will encounter applets from untrusted sources. Without any security, this would be a convenient way to spread viruses and perform unauthorized operations. Thus, Java has many built-in security mechanisms [JG96] (*e.g.*, type safety, class and bytecode verifier, sandboxing, the JAVACARD firewall) which help make Java suitable for dynamic downloading of untrusted code.

Java security fundamentally relies on ensuring type safety: running programs can access memory only in a safe, structured way. Several built-in security mechanisms are operating as Java virtual machine bytecodes. These mechanisms, which make Java programs robust, are: type-safe reference casting, structured memory access (no pointer arithmetic), automatic garbage collection (cannot explicitly free allocated memory), checking references for null, etc.

The class verifier contributes [JG96] to the Java Virtual Machine (JVM) security model by ensuring that class files loaded from untrusted sources are safe for the JVM to use. Rather than crashing upon encountering an improperly formed class file, the JVM class verifier rejects the malformed class file and throws an exception. The class verifier catches problems caused by buggy compilers, malicious crackers, or innocent binary incompatibility. One of the more important aspects of Java architecture is the bytecode verifier [Ler03], a mechanism that can verify the integrity of a sequence of bytecodes by performing a data-flow analysis on them. All JVM implementations must verify the integrity of bytecodes in some way. The verification process is a real burden in an embedded sytem, due to its sparse resources. Eva Rose [RR98] proposes an alternative solution, *the lightweight bytecode verification*, which splits the verification process in two parts: an off-card part which builds a verification certificate shipped with the code, and an on-card part which certifies the code using the verification certificate. Because information flow certification is even more expensive than bytecode verification, implementing an on-card information flow verifier is unrealistic. To overcome resource limitations, we adopt a technique in the style of Eva Rose's lightweight bytecode verification.

The Java *sandbox security model* [GM96] allows downloading code from any source. But as it is running, the sandbox restricts code from untrusted sources from taking any actions that could possibly harm the system. This approach largely prevents applications from sharing data. In some cases, sharing data must be allowed.

As we can see, Java provides many security mechanisms. But either they are too restrictive for open systems (the sandbox model forbids data sharing) or they do not ensure security in terms of information flow (well typed programs can hide illegal flow of information). Hence, information flow must be enforced by dedicated tools, which must fit nevertheless in the current JVM security architecture.

2.3.6 Flow sensitivity vs. flow insensitivity

Enforcing non-interference requires the analysis of flow of information throughout the control flow of the program. The control flow of a program plays an important role in static analysis and on the precision of the results. With respect to the interpretation of control flow, program techniques can be divided in two major categories: flow-insensitive and flow-sensitive.

Flow-insensitive analysis is independent of the control flow encountered as it does not take into consideration the order in which the instructions are executed. The program is considered as a set of statements. Information given simply indicates that a particular fact may hold anywhere in the program because it does hold somewhere in the program. *Flow-sensitive* [HS06] analysis depends on control flow. The program is considered as a sequence of statements. A given piece of information indicates that a particular fact is true at a certain point in the program.

Considering that `p` is a public, low variable and `s` is a secret, high variable, the example

```
p = s;
p = 0;
```

yields an illegal flow of information in a flow-insensitive analysis, yet the program clearly satisfies non-interference. Hence, flow-insensitive information is fast to compute, but not very precise. Flow-sensitive analysis usually provides more precise information than flow-insensitive analysis but it is also usually considerably more expensive in terms of computational time and space. This thesis presents a flow-sensitive approach to information flow analysis. Considering that our target systems have limited resources the flow-sensitive approach makes our goal more difficult to reach but in the same time more challenging.

2.4 State of the art

Information flow enforcement is a well studied area. A considerable amount of work on information flow control, based on static analysis, has been achieved in the last decades [Den76, Mye99a, VIS96]. A survey on language-based security and information flow techniques is presented in [SM03].

2.4.1 Information flow analysis approaches

One of the early contribution in this area is the security lattice model of information flow defined in [Den76] by Dorothy Denning. Information flow is allowed only when security labels on data along the path increase monotonically. Based on this model, Denning and Denning presented in [DD77] a certification mechanism for statically verifying the secure information flow in a program. These models are informal, they do not provide correctness proof.

Most of the approaches are *typed-based* [BN02, SBN04]. In type-based approaches the security level of any variable belongs to its type. Information flow security is checked by means of type systems: well-typed programs enforce non-interference. Volpano *et al.* [VIS96, VS97] are the pioneers of type-based approaches as they developed an elegant type system proved to enforce non-interference. They considered a sequential imperative language with procedures; the language is rather simple and it does not support object-oriented features nor polymorphism.

Type systems are usually simple to implement but they are often too imprecise. Their main weakness is that they are flow insensitive. The program `p = s; p = 0;` (where `s` is a secret variable and `p` a public variable) is rejected, as type systems require every subprogram to be well typed. Recently, Hunt and Sands proposed a family of flow-sensitive type systems [HS06] for tracking information flow.

In recent years, much of the litterature has focused on proving results for object-oriented [Mye99a, BN02] and low-level languages [KS02, EBM05, RMB05]. As our goal is to check information flow security for applications downloaded in a Java-enabled environment, in the following, we discuss work that adresses non-interference for Java and object oriented languages [Mye99a] and low level languages, more specifically, Java bytecode [BBR04, KS02, GS05].

Object-oriented languages and Java The type system of Volpano and Smith was extended by Banerjee and Naumann [BN02, BN05] to support a realistic Java-like object-oriented language with pointers, mutable object fields, dynamic dispatch and inheritance, type casts, mutually recursive classes and methods. Their type system is proved to enforce non-interference manually [BN02] and later mechanically using the PVS theorem prover [Nau05]. The result is important but the language still lacks some advanced features such as exceptions and interfaces.

Sun *et al.* [SBN04, Sun08] use the sequential class-based language defined in [BN02] to present an automatic inference algorithm of security type annotations of well-typed programs. Modular inference is achieved in the presence of method inheritance and override. Security levels of fields in a class are either defined or typed with a level variable which appears in types of method parameters and in the result type. Hence, as a result of the modular inference, each class is parametrized by the levels in its fields and each method in a class can be given a polymorphic signature. Due to its modularity, this approach can be successfully used in an open environment. But, unfortunately, it works as a source to source compiler and it does not addresses problems raised by our target systems (*i.e.*, scarce resources).

Amtoft *et al.* [ABB06] use pointer alias in order to perform an inter-procedural and flow sensitive information flow. Their analysis is also modular and it can be performed even in the absence of security levels. But again, their model checks a simple high-level imperative language, and hence it cannot be applied in the context of embedded systems.

Other techniques have been used to enforce information flow security. Hammer et. al. use program dependence graphs (PDG's), usually used for program slicing [XQZ$^+$05], to model information flow through a high-level Java program [HKS06]. The PDGs are more powerful than type based techniques as they are flow sensitive. PDGs express dependencies between program statements and expressions and the order in which they should be executed; program statements or expressions are the graph nodes. A data dependence edge $x \rightarrow y$ means that statement x assigns a variable which is used in statement y (without being reassigned underway). A control dependence edge $x \to y$ means that the mere execution of y depends on the value of the expression x (which is typically a condition in an if- or while-statement). Computing PDGs is very expensive and hardly imaginable to be used in a small system.

Low-level languages As we have shown, the information flow certification for open systems must be done onboard, preferably at loading time, hence on already compiled code (*i.e.*, Java bytecode). Futhermode, in an ubiquitous environment, where software units are downloaded dynamically from untrusted issuers, the sources code is not available. Hence, approaches described above, which target Java source programs, cannot be applied in the desired environment. Working on bytecode implies a more complex analysis including abstract interpretation of the Java Virtual Machine framework (stack, local variables). Moreover, the certification of low-level languages is more complex as the absence of high-level control constructs dictates the needs for other mechanisms to correctly compute the implicit information flow. In addition, working directly on Java bytecode presents the advantage of being less sensitive to evolutions of Java language.

Some works for assembly languages try to recover the source-level program abstractions, by producing static type annotations which preserve the type compilation. Based on this idea, Zdancewic and Myers used ordered linear continuations to simulate source-level program structures [ZM02]. The notion of linear continuation has been developped by Bonelli *et al.* [EBM05] in SIFTAL, a typed assembly language for secure information flow, and improved in the SIF language [RMB05]. TALC [YI06] is another typed assembly language which enforces non-interference. However TALC is richer than SIF as it supports code pointers and call stack.

Barthe *et al.* [BBR04] presented an information flow type system for a simple low level language featuring jumps and calls (but no objects) and showed that the type system enforces non-interference. In [BR05], Barthe and Rezk extend the information flow model defined in [BBR04] with new features to include classes, objects and exceptions. They also prove that the information flow type system enforces non-interference. The information flow type system defined in [BR05] is improved in [BPR07] by adding support for arrays and methods; the soundness proof has been machine checked using the proof assistant Coq [The04, Ber01]. This formalization allowed the extraction of a certified bytecode verifier; it is the first sound and implemented information flow type system for an expressive fragment of the JVM. The type system is elegant and strong, but it is flow insensitive and it lacks features needed in an open systems, *i.e.*, modularity.

In [BRN06], Barthe, Naumann and Rezk use certified compilers to study the formal connection between security policies on source code and properties of compiled code. They extend the type system in [BN05] to include exceptions and prove that, if the source program is typeable, then the compliled bytecode is also typeable. They connect the source language to the low-level language of Barthe and Rezk [BR05], for which non-interference has been proved correct.

The PACAP case study [BCG$^+$02] uses model checking to verify secure interaction of multiple JAVACARD applets on a single smartcard. The environment considered in the PACAP project is the closest one to our target systems, but, unfortunately, even if multiple software units are considered, the smartcard is not open: the analysis relies on a call graph. The SMV model checker assures that an invariant which captures the absence of illegal information flows is maintained thoughtout the execution of the program. Their approach has been refined by Bernardeschi *et al.*, which propose in [BF02] the use of abstract interpretation and model checking techniques to verify secure information flow.

Bernardeschi *et al.* [BFLM04] propose a method that transforms the original Java bytecode and class hierarchy so that illicit information flows are detected by the standard Java type verifier. Security levels are modeled as abstract data types and the original code is transformed in such a way that a typing error detected by the verifier on the transformed code corresponds to a possible illicit information flow in the original code. Security levels are assigned to classes and the transformed code has a class file for each security level; the class hierarchy implements the order among security levels. Avvenute *et al.* [ABF03] propose an approach similar to type-level abstract interpretation used in standard Java bytecode verification. The algorithm adopted by the standard bytecode verifier is applied to the domain of secrecy levels (assigned to classes, methods parameters and returned values) instead of types.

Genaim and Spoto [GS05] use abstract interpretation [CC77] and boolean functions to verify context and flow-sensitive information flow for monothreaded Java bytecode. No formal proof is given. Zanardini [Zan06] considers a fragment of Java bytecode and presents an analysis for checking a weaker and more general form of non-interference called abstract non-intereference [GM04], which allows some selected part of secret information to flow to public parts of program. No implementation or solution for open systems is given.

Confidentiality and integrity properties for multiapplication JAVACARD smart cards have been formally proven in [ACL03, And06]. Authors consider the *applet isolation principle* which forbids all colaborations between applets. In this thesis, we make a step further as we address confidentiality issues for multiapplication systems allowing collaborations and data sharing.

Dynamic enforcement All the previous work uses a static analysis to check secure information flow. As discussed before, dynamic enforcement was considered unsatisfiable as leaks induced by the execution or the non-execution of a branch were difficult to predict. Recently, static analysis and dynamic enforcement have been combined [CF07, SST07] in order to avoid such leaks. Chandra and Franz [CF07] present the implementation of an information flow framework that dynamically certifies statically annotated Java bytecode programs. The dynamic enforcement of information flow generates a slowdown factor of 2. In contrast we aim at a static analyis performed before code execution which does not introduce any runtime overhead.

Another technique for dynamic information flow monitoring has been recently presented in [CC08]. Authors consider that the only way data can leak is through `output` commands. Hence, programs are statically analyzed, using slicing techniques. Dependeces between executed program statement and `output` statements are classified in three categories: never leak sensitive data, may leak data and always leak data. Commands that always leak sensitive data are statically replaced by a `skip` command, while commands that may leak sensitive data are replaced on the fly, if the execution is invalid. Monitoring is not implemented. Since dynamic on the fly code transformation relies on dependences graphs, they must be available at runtime. The monitor is not implemented so no indications on the possible overhead are provided.

2.4.2 Information flow policies

Most of these approaches enforce non-interference, which sometimes can be too restrictive. In certain cases a program needs to leak some confidential information in order to achieve its purpose. For example, a password checking application compares a user supplied password with a stored password, which is secret data. Even if the boolean result depends on the stored password, such a behaviour is acceptable and such programs should not be rejected by an information flow analysis.

Much research has been devoted to express more realistic information flow policies by modifying or relaxing the definition of non-interference or by defining new formulations for secure information flow. Smith defines probabilistic non-interference [Smi01] in which the initial values of secret variables should not affect the joint probability distribution of the possible final values of any public variables, while in [Lau01] secure information flow is defined in terms of computational indistinguishability. In contrast with non-interference, which forbids any flow from secret to public, this approach aims at verifying that an attacker cannot learn anything about secret inputs by observing the public outputs. Computational indistinguishability lies on the idea that, at a certain point, it is impossible to deduce secret data altered by arithmetical or logical operations from public outputs. For example, a trusted encryption function, which performs many operations on secret data, does not leak any information in reality [HKM05]. Yet, non-interference rejects such functions.

Most of the efforts have been done in the direction of data declassification. There are a lot of efforts on data downgrading using various approaches, including abstract non-interference [GM04, GM05], relaxed non-interference [LZ05], robust declassification [ZM01, MSZ06, MSZ04], certificate-based declassification [TZ05]. A complete survey on the recent research efforts on declassification is available in [SS05, SS07].

Static, type-based information flow analysis techniques typically assume a global security policy on object fields which can lead to the assignment of a fixed security level to each field. Recently, authors present in [BB08] a flow-sensitive type system for statically detecting illegal flows of information in a JVM-like language that allows the level of a field to vary at different object creation points. They also prove a non-interference result for this language.

Despite all these efforts in relaxing non-interference and formulating new security models, defining realistic information flow policies is still one of the most important challenges in information flow security. Moreover, the security policies and code analysis are usually mixed; verifying a new information flow policy requires the analysis of the entire system.

We claim that information flow analysis and security policies enforcement must be separated. The information flow analysis must compute information flows and security policies must be verified *a posteriori* using only the already computed result. The information flow analysis we present in this thesis lies on this idea of separation of concerns. Moreover, we make a step further in the direction of realistic policies by defining information flow policies using a domain specific language. The language allows, on the one hand, data declassification, and on the other hand, it refines non-interference by expressing allowed collaborations between programs.

2.4.3 Existing implementations

Despite a long history and a large amount of research, there are very few practical systems and software that perform information flow analysis and enforce information flow security policies. Some reasons could be the limitations of many existing information flow models. The more precise information flow models are, the more they become complex and difficult to implement in practice. The cost of implementing language-based security mechanism is considered too expensive. Nevertheless, there are a couple of interesting examples.

Jif[1] [Mye99a, Mye99b] is the most complete framework for detecting information flows in Java applications. It is based on *JFlow*, an extension of the Java language that permits static checking of flow annotations as an extended form of type checking. *Jif* offers a real programming environment and it is based on a decentralized data model [ML97, ML98, ML00], which supports multiple owners and declassification and allows users to explicitly declassify data. *JFlow* supports many language features like subclassing, mutable objects and exceptions. *JFlow* is a powerful tool structured as a source-to-source translator, the output being a Java program that can be compiled by any Java compiler. It adds reliability to the software implementation but not to the deployment and link on a platform. *JFlow* offers support to a reliable development by defining a new programming language which mixes source code and security policies in a coherent set, but it does not adresses modularity and incrementality issues.

Flow Caml [Sim03, Sim04] extends the Objective Caml language with a type system tracing information flow. The implemented type system, Core ML, has been formally presented in [PS02, PS03]. Moreover, a correctness proof for the non-interference property of the type system is provided. As *Jif*, Flow Caml is a source-to-source translator: programs containing ML types

[1] `http://www.cs.cornell.edu/jif/`

annotated with security levels are statically checked and compiled into regular Objective C, files that can be compiled by any compiler. In contrast to *Jif*, Flow Caml has full type inference as the system verifies, without requiring source code annotations, that every information flow performed by the analyzed program is legal w.r.t. the security policy specified by the programmer.

The PACAP framework is dedicated to Java enabled embedded systems [BCG+02]. The PACAP framework involves a technique to verify interactions for Java enabled smartcards, based on predefined patterns. However, it may not detect information flows that lie outside these models. Moreover, the verification relies on the call graph, so it cannot be trusted in a Java/JAVACARD open environment.

Recently a new platform for information flow security has been released. As *JFlow*, *SecJ*[2] [Sun08] acts as a source-to-source translator, the output being a Java file. In contrast with usual techniques, which require manually annotating all fields, methods and parameters with security types, *SecJ* is able to automatically infer security type annotations of well typed programs. Compared to *Jif*, the strength of *SecJ* lies on its capability to perform type inference interprocedurally and modularly. In our knowledge, it is the first tool that addresses the security type inference in a modular way and in the presence of method inheritance and overriding.

All existing models are powerfull but, besides *SecJ*, they do not address the problems raised by an open environment. Only *SecJ* takes modularity into account, but it works on source code and it cannot be applied in an environment where assembly code is dynamically loaded. One of the aims of this thesis is to provide a practical information flow model, which bridges the gap between models such as *JFlow* and PACAP. The target are small open multiapplication systems.

2.5 Contributions and document structure

As we have seen before, information flow analysis has been actively investigated for several years, leading to a rich theory and language design, based on type-checking or static analysis. However, the information-flow based enforcement mechanisms have been scarcely applied in practice [Zda04], even for desktop computers. Unfortunately, due to the high complexity of the algorithms and to the lack of embedded resources, an information flow verifier has not yet been implemented on a small, embedded device. The few practical approaches for embedded systems rely on the call graph, they deal with offboard static verification, and they do not address the challenges raised by open environments.

From the previous discussion, we identify some very important issues which had little or almost no interest for the research community but which are essential in a small open environment:

- the verification must be done onboard, as it is the only place where security can be certified,
- as a consequence, the certified language must be Java assembly code (*i.e.*, JVM bytecode),
- the verification algorithm must be lightweight, *i.e.*, it must adapt to constraint resources of open systems (*e.g.*, limited memory, power consumption),
- in order to allow dynamic class loading and future evolutions of the open system, the verification must be modular,
- the verification must be applied to real Java applications, as most of the applications are written in Java and are downloaded in an executable form. Hence, programs must be annotated with security policies *a posteriori*. Imposing a new programming language, in which the programmer annotates the source code with security policies, is difficult, if not impossible, to achieve.

[2]http://www.cs.stevens.edu/~sunq/secj/

In this thesis, we give a complete solution for applying information flow security in the context of open systems. We address all the issues described above and we aim at a practical and automated method, which requires only little interaction with the user. To achieve our goal, we identify three major steps:

1. to propose a tool that integrates in the existing infrastructure, and that infers and verifies information flows in a software unit,
2. to express desired information flows in terms of security policies which can be verified *a posteriori*,
3. to prove the soundness of the model.

Our solutions rely on static analysis, abstract interpretation [CC77] and points-to analysis. Points-to analysis [WR99, Hod07] computes how references point one to another in a program. It is usually used for program optimization [GHSR06], but it is not sufficient for enforcing information flow, as its abstract domain is limited to references and it does not take into account primitive values. We extend points-to abstract domain with primitive values and, beside reference assignments, we take also into account primitive assignments and implicit flows. Hence, our model is more general than points-to analysis and points-to information can be easily recovered from our representation.

All these programming techniques are used in other contexts such as program slicing [XQZ$^+$05]. Program slicing models compute dependencies between program statements and data in order to eliminate statements that do not influence certain behaviours of the program. These models can be exploited to verify safe information flow in programs, as it has been done in [HKS06], but the analysis is more complex than classical information flow analysis and points-to analysis, since it performs unnecessary computations (*i.e.*, dependencies between program statements).

We now detail the contributions of this thesis and how we achieve the goals and the steps mentioned above.

2.5.1 An information flow analysis for embedded systems

In Chapter 3 we present a technique for enforcing information flow model in open systems. To adapt to the dynamic downloading from untrusted sources and throught insecure channels, we certify Java assembly language (JVM bytecode), and the verification is performed at loading time. Due to the limited resources of open systems, we split the verification process in two steps: *(1)* an external analysis, that could be performed on any computer and which computes, for each method, a flow signature containing all possible flows within the method, and proof elements which are shipped with the code, and *(2)* an embedded analysis, which certifies the flow signature, using the proof, at loading time.

The flow signature is computed using abstract interpretation and type inference techniques. Hence, we have designed for the external analysis an automatic type inference system for object-oriented languages. Our system does both inter-procedural and intra-procedural inference, in contrast with previous work on information flow inference for object-oriented languages which supports only intra-procedural inference [ML97]. Moreover, we perform modular inference, *e.g.*, each flow signature is computed only once. The modularity is essential in our context; this means that loading new classes does not require to reanalyze the entire system. Moreover, modularity allows us to add support for overriding.

2.5.2 A language for defining information flow policies

One of the main challenges in current information flow research area is to define realistic security policies that faithfully describe the desired information flow within a system. In the literature, information flow security policies are often limited to non-interference [GM82], where public outputs cannot depend on secret inputs. Policies defined with such relation are too restrictive, and not the desired policies in most of the cases, especially in open multiapplication systems [Gir99]. Moreover, policies are mixed with the code in a coherent set, hence redefining new policies requires reanalyzing the entire system. In order to escape from non-interference strictness and to separate policies and code, we define in Chapter 3.6 a domain specific language, which describes the allowed flow of information between applications. Programs are certified, at loading time, by verifying that the flow signatures (computed by the type inference system) respect the desired security policies. Hence the security policies verification is an extension of the embedded information flow model: we first compute the flow signature, and only after we define and verify the security policies. Changing the security policy does not require to reanalyze the entire system.

2.5.3 Applying information flow in practice

Even if information flow security is a well studied area, it has failed to show its usefulness in practice. The real challenge is not to define new information flow models, but to apply the results in practice and, in our case, on small open systems. In Chapter 4 we show how our model can be successfully applied in practice by: *(1)* showing experimental results, *(2)* integrating it in a general verification framework, combining design and software validation, and *(3)* applying it to some particular cases of Java applications such as J2ME MIDlets.

First, we briefly present the tools we developed that implement the models presented in the previous chapter. We discuss experimental results and their performances, both offboard and onboard, and then we compare them to other existing information flow verification tools, such as *Jif* and *SecJ*.

Then we make the information flow model even more practical by showing how it can be used in an integrated verification framework. Information flow leaks can appear not only in malicious software, but also due to a bad design of the software unit or a bad implementation. Hence, a continuous verification during the entire life cycle of the application is highly desired. In Section 4.2, we aim at providing a development framework which performs software verification and validation both at design time and implementation time. Hence, we combine our analysis tool with the USIE (User-system interaction effect) model [LT04] designed at the University of Victoria, Canada. The USIE model is used to capture and check security events at design time based on diagrams derived from sequence diagrams.

Our information flow tools are dedicated to standard Java applications running on small systems. By standard Java applications we mean applications storing sensitive data in their fields and which can directly collaborate with other applications within the same system. Since this is not the case for all small platforms supporting Java, our tools cannot be applied to all kinds of Java applications. Our goal is to have a tool which requires only a few adjustments in order to be adapted to other kind of Java applications, and that this adjustments can be done, in most of the cases, by the end users. To verify these claims, we show in Section 4.3 how our analysis can be applied in the context of J2SE and Java MIDlets. Moreover, the work has been done by an engineer who had no knowledge about information flow security and about our analysis. The results were satisfactory: the engineer has successfully adjusted our tool STAN to MIDlet in a short period of time and only with a few modifications.

2.5.4 A sound dependency analysis using abstract memory graphs

A complete information flow framework should address both practical and theoretical aspects. While before we concentrated on practical aspects, in Chapter 5 we tackle information flow problem from a theoretical point of view. We define a formal framework in which information flows in a method are modeled by means of abstract memory graphs. An abstract memory graph (or an

AMG) is a points-to graph extended with primitive values and flows arising from implicit flows. Throughout the remainder of this thesis, AMG will always stand for "abstract memory graph". We keep the practical side of information flow by separating analysis and security policies: this graph is computed independently of any security policies, which can be applied later by labeling the edges and nodes of the AMG. This leads to a more general analysis than information flow; many program analyses such as points-to, purity analysis can be recovered from our model.

As the embedded model, the analysis certifies JVM bytecode in a modular way: each AMG is computed only once. In Chapter 6 we prove the soundness of this approach for both intra-procedural and inter-procedural analysis, by stating a non-interference theorem.

This analysis is more general than the information flow model for embedded systems, which can be recovered as an approximation of the AMGs. A natural continuation would be to formally prove that the embedded model is a correct approximation of the dependency model. We discuss this in perspectives.

3 An information flow analysis for open systems

Contents

In this chapter we present a model for checking secure information flow in Java-enabled, open, multiapplication, small systems. We keep the main features of the Java Virtual Machine including support for dynamic class loading and overriding. Moreover, the model supports collaboration policies which can be specified and verified *a posteriori*.

We use a technique in the style of Necula's proof carrying code (PCC) to perform the onboard verification [RR98, Nec97]. This verifier scheme is an extension of the the Java bytecode lightweight verification [RR98]. The idea is to split the verification process in two parts. An offboard part (*the prover*), that computes a certificate and some "proof" indicating that the code is correct with respect to the security policy and an onboard part (*the verifier*), that uses the proof to certify the correctness of downloaded code.

The verifier is implemented as a user-defined ClassLoader, which can be successfully used on any Java Virtual Machine. Experimental results show that our verifier could be efficiently used for our target systems and hence, it can be successfully applied in practice.

3.1 Preliminaries

In this section, we briefly introduce the Java Virtual Machine (JVM) language we use and we discuss information flow issues and non-interference for small open systems. We conclude this section with a description of our approach.

3.1.1 Notations and instructions set

Our tool[1] supports the complete JVM specification [LY99], but in this document, due to limited space and in order to facilitate the reading, we focus on a general and representative subset of the JVM, depicted in Figure 3.1. The considered instructions retain the full power of expression of the JVM language and the main features of Java are supported (objects, method invocation, interfaces, static fields/methods, arrays, polymorphism and overriding, dynamic class loading). The `invoke` instruction corresponds to a virtual invocation (*i.e.*, `invokevirtual` in Java); all types of invocation in Java (*e.g.*, `invokeinterface`, `invokestatic`) are a simplified version of virtual invocation, and hence they are supported by our model. Our model supports handled exceptions, multi-threading and synchronization; you can notice that instructions dealing with such cases (`athrow`, `monitorenter`, `monitorexit`) are not included in our initial instruction set. Nevertheless, we discuss them extensively and add support later in this chapter.

We consider a set of class names Class, a set of methods names Method and a set Field of fields names. For two classes B, C, $B \leq C$ if and only if $B = C$ or B is a super class of C. We denote by $\mathcal{T}(o) \in \mathit{Class}$ the type of an object o and by $o.f$ the field f of o.

3.1.2 Information flow in open systems

In classical information flow analysis [Mye99a, BPR07], the source of information flow is large: security levels are applied to methods parameters, return value, local variables etc. Recently, in [ST07] authors state that the only real source of information flow are IO channels, thus policies must be specified for input/output streams.

In our interpretation of Java, we consider that confidential data in small open systems (*e.g.*, PIN code, cryptographic key) typically resides in instance fields. The kind of information flow that we prevent are leaks from high-security instance fields to low-security instance fields. We are concerned only with security for heap contents and we allow programs to manipulate sensitive data and to temporary store it on the operand stack and local variables. This approach to security is similar to the one taken in [HP06].

In our vision, we consider IO channels as a possible source of information, and also a possible destination. Moreover, IO channels are a public, observable source, thus writing confidential data to an IO output channel is insecure; in order to send sensitive information from object fields to output

[1]`http://www.lifl.fr/~ghindici/STAN`

`prim` *op*	primitive operation taking two operands, pushing the result on the stack
`pop`	pop the top of the stack
`bipush` n	push the primitive value n on the stack
`aconst_null`	push null on the stack
`new` C	creates new object of type C in the memory
α`newarray` C	creates new array of primitives or of objects of type C in the memory
`goto` a	jump to address a
`ifeq` a	jump to address a if the top of the stack is equal to zero
`ifnull` a	jump to address a if the reference on top of the stack is null
α`load` x	push the content of the local variable x on the stack
α`store` x	pop the top of the stack and store it into the local variable x
`getfield` $f_{C'}$	load the field $f_{C'}$ of the top of the stack on the stack
`putfield` $f_{C'}$	store the top of the stack in the field $f_{C'}$ of an object on the stack
`getstatic` $f_{C'}$	load the static field $f_{C'}$ on the stack
`putstatic` $f_{C'}$	store the top of the stack in the static field $f_{C'}$
α`aload`	pop the index and the array reference from the stack and push the element at index
α`astore`	pop the element, index and array reference from the stack and store the element in array, at index
`arraylength`	pop the array and push its length on the stack
`invoke` $m_{C'}$	virtual invocation of method $m_{C'}$
α`return`	return an object or primitive value and exit the method

α denotes the type: `i` for primitive types, `a` for references, or nothing (*i.e.*, `return`)

Figure 3.1: Instruction set

channels, declassification [SS05] techniques can be applied, by allowing release of confidential data only in certain methods.

Let us define the non-interference property of a Java software unit:

Non-interference for Java Public instance fields do not provide any information about secret instance fields, after executing the software unit.

In this context, we define three types of interference :

- **interference through inference**: information about secret fields can be inferred from public fields (through implicit flow),
- **interference through copy**: information about values of secret fields can be read from public fields (through explicit assignment),
- **interference through aliasing**: value of secret fields can be accessed through a reference stored in a public field.

From an *interference through copy* we can obtain at least the same amount of information that can be obtained from an *interference through inference*. In the same way, the *interference through aliasing* provides at least the same information and privileges as the *interference through copy*. As a conclusion, we consider that the amount of information leaked by an *interference through aliasing* is bigger than the one leaked by an *interference through copy*; similarly, the amount of information leaked by an *interference through copy* is bigger than the one leaked by an *interference through inference*.

Finally, the property of **non-interference** defined before must be ensured while a new software unit is deployed both for the **consistency** and **incrementality** of the open system.

Traditional non-interference enforcement relies on the call graph of the system of software units. Obviously, this approach fails on a system which allows dynamic loading of software units. Our aim is to allow dynamic downloading of new software units, while keeping the system safe. In other

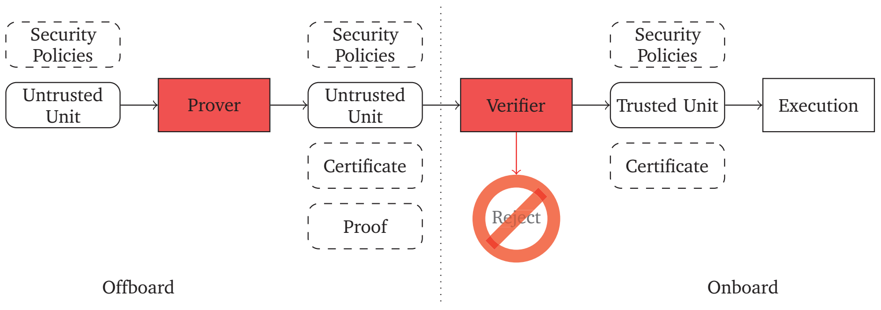


Figure 3.2: Lightweight information flow certification

terms, to certify the **non-interference** property of already installed software units (**consistency**) and newly loaded software unit (**incrementality**), without reanalyzing the entire system.

In order to deal with openness, we perform a compositional analysis, computing for each method a stand-alone flow signature. The flow signature of a method is independent of the context under which the method is called. It contains the flows, potentially generated by the execution of the method. One "type" is associated with every data source reflecting the flows generated by the method between this source and the others. Based on the knowledge of the flows, a software unit can verify its compliance with its own security policy. Thus, flows inside methods are detected by traditional static analysis while flows generated by interactions between methods are detected by composition of the methods signatures.

3.1.3 Our approach

The only place where security can be reasonably guaranteed for us is the Virtual Machine, while loading a new class. The certification cannot be performed before because the deployment context is unknown at compilation time and it would be harmful to be performed later as the execution time of a program would be penalized. Unfortunately, performing certification at loading time in a Virtual machine forces the developer to deploy its software unit in order to test its correctness. It is necessary to provide software tools which allow the developper to test the correctness of its units at compilation time.

As mentioned above, our goal is to support all Java features (especially inheritance and dynamic class loading) and to adapt our analysis to mobile code and open systems. The main challenge is to adapt the technique to the limited resources of open systems.

In the context of small systems, a technique known as "Lightweight bytecode verification" has been developed in [RR98] for Java bytecode type verification. This technique, closely related to proof-carrying code [Nec97], is interesting because:

1. it provides the developer with tools which are supposed to help him/her test the security of his/her software unit before loading it in the embedded environment,

2. it allows an open system to verify code received from an untrusted source without relying on a third party even if it does not have enough power to compute the proof itself.

It relies on the simple idea that it is easier to verify a result already computed. It consists of two phases, as depicted in Figure 3.2:

the offboard phase (on the producer side) which is assumed to have access to infinite resources (typically a personal computer compared to a small device), which computes the type correctness and annotates the bytecode with some proof elements,

the onboard phase (on the consumer side) which verifies, at loading time, the annotations obtained during the first phase. The annotations are embedded within the code and verified. The verification operation is linear in the code size and uses constant memory.

The first phase is performed by a *prover* while a *verifier* embedded in the JVM ClassLoader certifies the second phase. This technique has been developed for type checking and it relies on the lattice structure of types and on unification operations on this lattice. To use it in our context, we extend this technique by proposing in the next section an encoding of the flow of information in the form of a lattice. Moreover, we must deal with type inference and onboard certificate management.

We detect flows arising from assignments and implicit flows; we do not consider covert channels such as termination, timing, power channels, etc.

3.1.4 Structure of the presentation

We structure the presentation in the following way: in Section 3.2, we propose an information flow model which takes into account constraints of small open systems. We identify the main sources of information flow and the abstract domain, we define the security lattice, the flow relation and the flow signature of a method.

In Section 3.3, we detail how the information flow certificate and proof is computed during the offboard step (by the prover). We first present the intra-procedural analysis, and after we add support for method invocation. We also discuss some particular cases (exceptions, threads, synchronization) and we show an analysis example on the LoyaltyCard.

The certificate computed by the prover is verified at loading time by the verifier. In Section, 3.4 we give solutions to challenges raised by limited resources and dynamic class loading (certification and proof encoding, implicit flow verification, certificate management). In order to make our analysis highly portable, we implement the verifier as a user-defined class loader, and we present how we deal with virtual machines running a hierarchy of heterogeneous class loaders.

We continue by adding support, through contracts, for polymorphism and open class hierarchy in Section 3.5. Contracts describe the required behaviour of pieces of code not yet loaded in terms of information flow, more exactly the maximum flow of information that can be generated by the execution of a virtual method. Contracts can be computed offboard starting from a class hierarchy, or they can be manually defined. In order to be accepted onboard, newly loaded methods must respect the contracts of the class hierarchy to which they belong.

Finally, in Section 3.6 we extend our model with support for collaboration policies using a domain specific language which refines non-interference. Security policies are essential to finely express allowed information flows in an open system. To make them practical, policies and source bytecode are separated; hence policies can be defined *a posteriori* and verified only at loading time. Information flow verification and policies enforcement are not mixed.

3.2 Information flow model

In this section, we formalize an information flow framework adapted to small systems. As the model must be embedded with the code, the challenge is to express information flows in a method in a concise, but yet expressive manner.

We first identify sources of confidential data, which, in small systems, typically reside in instance fields. To express secrecy, we define a security lattice with two security levels (public and secret), with which class fields are labeled.

In order to keep the model as compact as possible, we perform a field independent, but security level sensitive analysis. Thus, the abstract domain is restricted to parameters and other abstractions for information flow sources (static fields, exceptions, return value of the method, etc.).

The support for openness and dynamic class loading is achieved by performing a compositional analysis. For each method we compute a context-insensitive *flow signature* which contains all potential flows between abstract values generated by the execution of the method. The definition of the flow relation between abstract values allows for a compact representation of flow signatures: one byte is sufficient to encode the possible flows between two abstract values.

Non-interference is ensured if flow signatures do not contain any flows from confidential data to a public output.

3.2.1 Security lattice

To keep the system simple and suited to small systems, we define a security lattice composed of two security levels:

$$L = \{s, p\},$$

where s stands for secret, high level and p stands for public, low security level. The order relation $\sqsubseteq$ between elements in L is defined as follows: $p \sqsubseteq s$. Using this order relation, we define the security lattice L^s of security levels. Nevertheless, adapting the present analysis to a more significant security lattice is straightforward but not adapted to constraint devices. Using a more complex lattice will result in more complex analysis and a considerable overhead (in time and mostly in memory) for desktop computers and out of reach of small systems.

Security levels are associated to information flow sources, thus objects fields, and they should not be confused with Java modifiers (`private`, `public`, `protected`). While Java modifiers express accessibility for the Java language, the s defined above expresses secrecy, the fact that the information must not be made accessible through information flow to unauthorized parties. The default security level of object fields is p, but restrictions on classes and their fields can be specified in an external file (*e.g.*, an XML file). We denote by $\mathcal{L}(C, f)$ the level associated with field f in a class C.

Tracing the behaviour of each field is expensive and memory consuming, thus this is not adequate in the domain of mobile code and small open systems. To reduce the size of information flow annotations, we perform a *field independent* but *security level sensitive* analysis. In a field independent analysis, all the fields in a structure are modeled as having the same location; thus, a write into one field writes to all the fields in the structure.

The *security level sensitive* analysis draws a distinction between fields of the same object having different security levels. Hence, we define:

Security level sensitive analysis All the fields of an object having the same security level are modeled as having the same location.

Thus, considering the security levels $L = \{s, p\}$, an object o is modeled as being made of two parts (locations): a secret part (denoted by o^s), for fields with security levels s and a public part (denoted by o^p), for fields with security level p. Our approach is more general than a fully *field independent* analysis, but still less precise than *field sensitive* approaches [GSRT07] that track the individual fields of individual pointers. A more precise analysis will produce more precise results, *i.e.*, less false flows, but abstract domain and the number of information flows will increase significantly, which will be more difficult, if not impossible, to embed. For example, if we refine the analysis and track individual fields of depth one, and if we consider an object o having three object fields f_1, f_2, f_3, than the abstract domain multiplies by three: it will contain the secret and public part of each field $(o.f_1^s, o.f_1^p, o.f_2^s, o.f_2^p, o.f_3^s, o.f_3^p)$. The abstract domain, and hence the complexity of the analysis, depends on the depth of the unfolded tree of fields and on the average number of fields of objects. A more compact tree (as in our case) leads to a less precise, but less complex analysis.

To deal with security levels at different field depths, we use the following convention:

The secret part o^s of an object o contains all the field access paths that contain at least one field having the security level s:

$$o^s = \{o.f_1 \dots f_n \mid \exists 0 < i \leq n, \mathcal{L}(\mathcal{T}(o.f_1 \dots f_{i-1}), f_i) = s\}, \tag{3.2.1}$$

The public part o^p of an object o refers to all fields of o that contain only fields with security level p on their access path:

$$o^p = \{o.f_1 \dots f_n \mid \forall 0 < i \leq n, \mathcal{L}(\mathcal{T}(o.f_1 \dots f_{i-1}), f_i) = p\}. \tag{3.2.2}$$

3.2.2 The abstract domain

Sensitive data are stored in object fields, while objects are made accessible to a method through parameters, objects allocated inside the method or objects returned by invoked methods. We use the *object allocation site model*: all objects created/returned at the same program statement have the same abstraction. Let P be the abstract domain for parameters of a method m (they are denoted by $p_0, p_1, \dots$). We denote by New the set of abstract values modeling the objects created by the execution of the method; $n_i \in New$ is the abstraction of objects created at instruction i. The set Ret models values returned by `invoke` statements in m: $r_i \in Ret$ denotes the return value of the method invoked at instruction i. Besides parameters and newly created objects, we identify the following abstract values, that might interfer in the information flow process:

- the return value of the current method, denoted by the abstract value R,
- input/output channels; all the channels are abstracted by a single value, IO,
- static fields; all static fields are modeled as the fields of a single object, denoted by the abstract value $Static$,
- exceptions; all thrown values flow to the abstract value Ex,
- constants, pushed on the stack by the `bipush` instruction, and defined by the abstract value $Const$,
- an abstract value `null` for null reference.

Hence, **the abstract domain of a method** m is defined as

$$\ddot{\Sigma}_m = P \cup New \cup Ret \cup \{R, IO, Static, Ex, Const, \mathtt{null}\}.$$

In order to unify the model, we associate security levels to all abstract values, including those not abstracting objects. Hence, the input/output channels IO, the static world $Static$, the exceptions Ex, the constants $Const$ and `null` have the default security level p, as all sensitive data flowing to them potentially leaks to unauthorized parties. Parameters of primitive type also have the security level p. If a primitive instance field having the security level s is passed as parameter to m, its security level is taken into consideration when applying the context call to m.

We can now enrich the abstract domain with security levels and obtain the set

$$\begin{aligned}\ddot{\Sigma}_m^L = \quad & (P_{|Obj} \cup Ret_{|Obj} \cup New) \times L \cup \\ & (P_{|Val} \cup Ret_{|Val} \cup \{R, Static, Ex, IO, Const, \mathtt{null}\}) \times \{p\}\end{aligned}$$

where $A_{|Obj}$ represents the set A restricted to objects, $A_{|Val}$ the set A restricted to primitive values and a^l is an equivalent notation for $(a, l) \in \ddot{\Sigma}_m \times L$. Moreover, we extend this notation to sets of elements: $A^l = \{a^l \mid \forall a \in A\}$, with $A \subseteq \ddot{\Sigma}_m$ and $l \in L$.

(a)

```
class A {
  int f;
  B g;
}

class B {
  int h;
}
```

(b)

```
void m(A o1, A o2, A o3){
   o1.g = o2.g;
   o2.g.h = o3.f;
}


0: aload 1
1: aload 2
2: getfield A.g
3: putfield A.g
4: aload 2
5: getfield A.g
6: aload 3
7: getfield A.f
8: putfield B.h
```

(c)

```
void m'(A o1, A o2, A o3){
   o1.f = o2.f;
   o2.f = o3.f;
}


0: aload 1
1: aload 2
2: getfield A.f
3: putfield A.f
4: aload 2
5: getfield A.g
6: aload 3
7: getfield A.f
8: putfield B.h
```

Figure 3.3: Flow propagation example

3.2.3 Flow relation

Data propagation example

In Section 3.1.2, page 24, we distinguish three types of interference: interference through inference, interference through copy and interference through aliasing. To show the necessity of this distinction, let us consider the example in Figure 3.3, where `o1`, `o2` and `o3` are three objects of type `A`, stored in local variables `1`, `2` and `3` respectively. For simplicity, we assume that all fields have the security level p. The first assignment in Figure 3.3b, `o1.g = o2.g`, generates an *interference through aliasing* between `o1` and `o2` (we recall that our analysis is field independent and all fields of an object have the same abstract location). Thus, subsequent changes to `o2` also affect `o1`. The second assignment, `o2.g.h = o3.f`, not only creates an interference between `o3` and `o2`, but also an interference between `o3` and `o1`, due to an alias between `o2` and `o1`.

Let us now consider the second example in Figure 3.3c. The first assignment creates an interference from `o2` to `o1`, but this time it is an *interference through copy*, as the field `f` is of primitive type. In this case, any subsequent modifications to `o2` do not affect `o1`. Thus the second assignment creates only an interference between `o3` and `o2`.

Typing information flows

The example above shows that, in order to correctly propagate data, flows must be typed. This choice is imposed by the approximation of the field independent analysis. In a field sensitive analysis, this distinction is not mandatory, as every field of every object is tracked independently, and the type of flow is given by the type of the field.

We now define the flow relation:

Flow relation There is a flow from an input a to an output b, denoted by $b \rightarrow a$, if an observer of b can learn information about a.

The analysis is an extension of a points-to analysis: besides aliases between objects, we also take into consideration primitive assignments and implicit flows. Because alias may lead to further data propagation while primitive assignments do not, we type the flow relation: a flow is denoted by $b \xrightarrow{t} a$, with $t \in \mathcal{F}$, where

$$\mathcal{F} = \{\mathbf{r}, \mathbf{v}, \mathbf{i}\}$$

denotes the set of possible flows, which correspond to the three forms of interference defined in Section 3.1.2:

- *reference flows* ($\mathtt{r}$), generate *interference through aliasing* and denote aliases that may lead to further data transfers,
- *value flows* ($\mathtt{v}$), generate *interference through copy* and represent data transfer of primitive type,
- *implicit flows* ($\mathtt{i}$), generate *interference through inference* and stand for flows arising from the control structure of the program.

In Section 3.1.2, we show that the interference through aliasing is stronger than the interference through copy, which is stronger than the interference through inference. Hence, we can define an order relation on the type of flows:

Order relation on type of flows $\mathtt{i} \sqsubseteq \mathtt{v} \sqsubseteq \mathtt{r}$.

Moreover, if we take into consideration the security levels and the fact that our model is security level sensitive, a flow from b to a is a triple (l_1, l_2, ϕ), where $l_1 \in L$ represents the security level of a, $l_2 \in L$ the security level of b and $\phi \in \mathcal{F}$ the type of the flow. There can be several flows (triples) between a and b. The example in Figure 3.3c generates a flow $(p, p, \mathtt{v})$ between `o1` and `o2`, meaning that the public part of `o2` flows to the public part of `o1` through a primitive assignment, and the flow $(p, p, \mathtt{v})$ between `o2` and `o3`.

Let $\Delta = L^2 \times \mathcal{F}$ be the set of all possible flows between two abstract values a and b. For convenience, we denote a flow (l_1, l_2, ϕ) by

$$a^{l_1} \xrightarrow{\phi} b^{l_2},$$

and it can be read:

- either *from the part l_1 of a (or from a field having the security level l_1 of a) we can learn information about the part l_2 of b (or about a field having the security level l_2 of b) through a flow of type ϕ*,
- or *the part l_2 of b (or a field having the security level l_2 of b) flows to the part l_1 of a (or to a field having the security level l_1 of a) through a flow of type ϕ.*

For example, the set $\{(s, s, \mathtt{v})\}$ is denoted by $a^s \xrightarrow{\mathtt{v}} b^s$ and represents an interference through copy (value flow) between the secret part of a and the secret part of b. The arrow is about the path to follow to get the information and not about the movement of the information.

Order relation and lattice of set of flows

Based on the order relation between types of flows, we define an *order relation on* Δ: given two flows $\vartheta', \vartheta'' \in \Delta$ such that $\vartheta' = (l_1', l_2', \phi')$ and $\vartheta'' = (l_1'', l_2'', \phi'')$, then $\vartheta' \leq \vartheta''$ if and only if $l_1' = l_1''$, $l_2' = l_2''$ and $\phi' \sqsubseteq \phi''$.

There can be more than one flow between two abstract values, hence we express the flow relation using *set of flows*, with values in $\wp(\Delta)$, where $\wp(\Delta)$ designates subsets of Δ. For example, the set $\{(p, s, \mathtt{r}), (s, s, \mathtt{r})\}$ expresses two flows between a and b: $a^p \xrightarrow{\mathtt{r}} b^s$ and $a^s \xrightarrow{\mathtt{r}} b^s$. For convenience, we denote this flow by $a^{p,s} \xrightarrow{\mathtt{r}} b^s$. For two sets A and B, $A \xrightarrow{\phi} B$ denotes that every element of A is related to every element of B through a flow of type ϕ.

Using the order relation on Δ, we can define a partial order relation on $\wp(\Delta)$:

$$\begin{tikzcd}
 & & a^{p,s} \xrightarrow{\mathtt{r}} b^{p,s} \arrow[dl, dash] \arrow[dr, dash] & & \\
 & a^{p,s} \xrightarrow{\mathtt{r}} b^{s} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & & a^{p} \xrightarrow{\mathtt{r}} b^{p,s} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
a^{s} \xrightarrow{\mathtt{r}} b^{s} \arrow[d, dash] & a^{p,s} \xrightarrow{\mathtt{v}} b^{s} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & a^{p} \xrightarrow{\mathtt{r}} b^{s} \arrow[d, dash] & a^{p} \xrightarrow{\mathtt{v}} b^{p,s} \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & a^{p} \xrightarrow{\mathtt{r}} b^{p} \arrow[d, dash] \\
a^{s} \xrightarrow{\mathtt{v}} b^{s} \arrow[d, dash] & a^{p,s} \xrightarrow{\mathtt{i}} b^{s} \arrow[dl, dash] \arrow[dr, dash] & a^{p} \xrightarrow{\mathtt{v}} b^{s} \arrow[d, dash] & a^{p} \xrightarrow{\mathtt{i}} b^{p,s} \arrow[dl, dash] \arrow[dr, dash] & a^{p} \xrightarrow{\mathtt{v}} b^{p} \arrow[d, dash] \\
a^{s} \xrightarrow{\mathtt{i}} b^{s} & & a^{p} \xrightarrow{\mathtt{i}} b^{s} & & a^{p} \xrightarrow{\mathtt{i}} b^{p}
\end{tikzcd}$$

Figure 3.4: Extract of the lattice of flows

Order relation on sets of flows Given two sets of flows $\theta_1, \theta_2 \in \wp(\Delta)$, $\theta_1 \leq \theta_2$ if for any flow $\vartheta_1 \in \theta_1$ there exists a flow $\vartheta_2 \in \theta_2$ such that $\vartheta_1 \leq \vartheta_2$.

This order relation allows us to organise sets of flows as a lattice (Figure 3.4 shows an extract of it):

Lattice of sets of flows We define by $\Theta = (\wp(\Delta), \leq)$ the lattice of set of flows between two abstract values. The join of two elements is the element generated by their union, and the meet of two elements is their intersection. The bottom of the lattice is represented by an empty set, meaning that there is no flow of information, while the top of the lattice is represented by $\{a^{p,s} \xrightarrow{\mathtt{r}} b^{p,s}\}$.

We define an equivalence relation, $\sim$ on Θ, as follows:

$$\theta_1 \sim \theta_2 \text{ iff } \theta_1 \leq \theta_2 \text{ and } \theta_2 \leq \theta_1.$$

Informally, two sets of flows are equivalent if for any flow, from one of the sets, there exists a bigger flow (according to the order relation on flows), in the other set. Thanks to the equivalence relation, we can obtain a compact representation of the lattice of sets of flows, which is essential in the context of small systems, with limited resources.

For example, the sets $\theta_1 = \{a^s \xrightarrow{\mathtt{r}} b^s\}$ and $\theta_2 = \{a^s \xrightarrow{\mathtt{r}} b^s, a^s \xrightarrow{\mathtt{v}} b^s\}$ are equivalent ($\theta_1 \sim \theta_2$), as $a^s \xrightarrow{\mathtt{v}} b^s \leq a^s \xrightarrow{\mathtt{r}} b^s$ and $a^s \xrightarrow{\mathtt{r}} b^s \leq a^s \xrightarrow{\mathtt{r}} b^s$. Moreover, the equivalence class of $\{a^s \xrightarrow{\mathtt{r}} b^s\}$ is $[\{a^s \xrightarrow{\mathtt{r}} b^s\}] = \{\{a^s \xrightarrow{\mathtt{r}} b^s\}, \{a^s \xrightarrow{\mathtt{r}} b^s, a^s \xrightarrow{\mathtt{v}} b^s\}, \{a^s \xrightarrow{\mathtt{r}} b^s, a^s \xrightarrow{\mathtt{i}} b^s\}, \{a^s \xrightarrow{\mathtt{r}} b^s, a^s \xrightarrow{\mathtt{v}} b^s, a^s \xrightarrow{\mathtt{i}} b^s\}\}$.

There are 12 elements in Δ (the cardinality of Δ is $|\Delta| = 12$), and hence we can have 2^{12} possible flows between two abstract values ($|\wp(\Delta)| = 2^{12}$). Some flows are equivalent according to relation $\sim$ relation, hence between two abstract values a and b we can have at most 256 distinct sets of possible flows; in other words, there are 256 equivalence classes ($|\wp(\Delta)/\sim| = 2^8$). This result is very important for a constraint system, as it allows us to encode the flows between two abstract values on a single byte. Moreover, the binary encoding allows to manipulate flows using simple bitwise logical operations. For example, the join of two sets of flows $\theta_1, \theta_2 \in \Theta$ corresponds to the bitwise OR operation.

Finally, in this subsection we have defined:

- an order relation between sets of flows,
- and a lattice of flows, which allows us to use the lightweight verification technique in our context.

3.2.4 The flow signature of a method

Based on the definition of the abstract domain and of the flow relation between abstract values, we can finally define the flow signature of a method at the set of flows potentially generated by the execution of a method. A flow signature carries relevant information for a later use of the method.

The flow signature of a method m The flow signature $\ddot{S}_m$ of method m contains flows $(a, b, \vartheta) \in \ddot{\Sigma}_m \times \ddot{\Sigma}_m \times \Theta$ such that the execution of m potentially generates a flow of type ϑ from b to a.

We denote by $\ddot{\mathcal{S}}_m$ the domain of flow signatures for a method m. We extend the order relation between flows in Θ to an order relation for flow signatures in $\ddot{\mathcal{S}}_m$.

Order relation on flow signatures Given two flow signatures $\ddot{S}'_m, \ddot{S}''_m \in \ddot{\mathcal{S}}_m$, $\ddot{S}'_m$ is *smaller than* $\ddot{S}''_m$, denoted by $\ddot{S}'_m \sqsubseteq \ddot{S}''_m$, if for all flows $(a, b, \vartheta') \in \ddot{S}'_m$ there exists a flow $(a, b, \vartheta'') \in \ddot{S}''_m$ such that $\vartheta' \leq \vartheta''$.

This order relation allows us to define a **lattice of flow signatures** $\Lambda = \wp(\ddot{\mathcal{S}}_m)$. The lattice of flow signatures is a natural extension of lattice of flows. The union of two flow signatures $\ddot{S}'_m \sqcup \ddot{S}''_m$ is the least upper bound flow signature between $\ddot{S}'_m$ and $\ddot{S}''_m$ according to the lattice Λ.

Certain abstract values defined above (New, Ret, $Const$) are locally defined inside a method and are not relevant outside. Thus, the global result must be restricted to the values that survive at the end of the method:

$$\Sigma_m = P \cup \{R, Static, Ex, IO\}.$$

The *final flow signature* of m contains only flows between values in Σ_m:

The final flow signature of a method m The final flow signature S_m of method m contains flows $(a, b, \vartheta) \in \Sigma_m \times \Sigma_m \times \Theta$ such that the execution of m potentially generates a flow of type ϑ from b to a.

The notation $\mathcal{S}_m$ gives the domain of final flow signatures. Throughout the remainder of this thesis, the term *flow signature* of a method m will designate the final flow signature of m.

3.2.5 Enforcing non-interference

The flow signature of a method allows us to verify the non-interference [GM82]: a program is secure w.r.t. non-interference if there is no flow of information from an abstract value with security level s to an abstract value labeled with p. In our framework, a method is secure if for any flow $a^{l_1} \xrightarrow{\phi} b^{l_2} \in \Delta$, we have $l_2 \sqsubseteq l_1$, according to the ordering relation on the lattice of security levels L^s.

Considering $L = \{s, p\}$, a method is secure if there is no flow from an s value to a p value. Hence, we can define non-interference w.r.t. to the flow signature of a method:

Non-interference for flow signatures The flow signature of a method, S_m, is secure w.r.t. to non-interference if it does not contain flows of type $a^p \xrightarrow{\phi} b^s$, except the case when a represents the return value of the method, R. The verification of flows of confidential data to return values is postponed until the method is invoked.

3.3 Information flow prover

Dynamic enforcement of information flow security policies had little attention from the research community, as implicit flows, arising from the execution or non-execution of a branch of the program, cannot be always detected. Unsatisfiable for a normal system, the dynamic enforcement is even less adapted to constraint systems, as it introduces a runtime overhead unjustified for the end user. Hence, static enforcement of information flow policies is the obvious choice, as it has the advantage of avoiding overheads, and detecting implicit flows.

In the previous section, we describe the behaviour of a method in terms of information flow using flow signatures. The flow signature contains all flows potentially generated by the execution of the method. To compute methods signatures, we perform for each method an intra-method static abstract interpretation relying on a classical program semantics composed of a set of transformation rules. The control flow structure of the Java bytecode dictates an iteration on the set of instructions for each method.

The existence of recursive and inter-dependent methods dictates an incremental inter-method analysis, starting with the set of empty signatures and iterating on a set of methods until a fixed-point has been reached.

Hence, computing flow signatures onboard raises several problems:

- the fixed point computation and the iterations (on the set of instructions and on the set of methods) requires many resources,

- in an open system, the software units and classes loaded are not available and are not known from the start.

In order to overcome these limitations, we split the verification process in two steps: *(1)* an external analysis, which computes flow signatures and some proof elements, and *(2)* an embedded analysis which certifies at loading time, using the proof elements, the flow signatures.

Moreover, in order to comply with the Java paradigm of dynamic class loading and to support open systems, the analysis is compositional, context-insensitive and it does not rely on the call graph: the interpretation of an invoke bytecode consists of applying the flow signature of the called method to the flow signature of the calling method. Such an approach has two main advantages:

1. the analysis of new classes does not require re-analyzing old code,

2. a class is verified using already certified signatures, and not with signatures used during compilation.

In this section we describe the external analysis, performed by a *prover*. In the first part we present the intra-procedural analysis, which concentrates on correctly computing flow signatures for the JVM language without taking into consideration method invocation. We discuss implicit flows and give the abstract semantics of Java bytecode.

In the second part, we add support for method invocation while presenting the inter-procedural analysis. Finally, we discuss particularly cases of the JVM, such as exceptions and threads, and we detail the analysis on an example from the LoyaltyCard.

3.3.1 Intra-procedural analysis

We now consider the analysis of a method without method call. We perform a flow-sensitive static abstract interpretation of JVM bytecode, which computes, for each method, its flow signature.

The information flow analysis is an alias analysis extended to take into consideration flows of primitive fields and implicit flows. Classical points-to analysis can be recovered as an abstract interpretation of our information flow analysis by eliminating unnecessary flows.

Flow sensitivity

In our framework, we perform a flow-sensitive analysis, as we follow all the control flow paths in a program and compute, through static abstract interpretation, for each program point, a flow signature. On the other side, flow-insensitive analysis is independent of the control flow encountered, as they do not take into consideration the order in which the instructions are executed.
For example, the code

```
m(A o1, A o2, int a, int b){
   A l = o1;
   l.f = a;
   l = b;
   l.f = b;
}
```

creates two flows in a flow-sensitive analysis ($\texttt{o}_1 \rightarrow \texttt{a}$ and $\texttt{o}_2 \rightarrow \texttt{b}$) and four flows in flow insensitive analysis ($\texttt{o}_1 \rightarrow \texttt{a}$, $\texttt{o}_2 \rightarrow \texttt{b}$, $\texttt{o}_1 \rightarrow \texttt{b}$, and $\texttt{o}_2 \rightarrow \texttt{a}$).

The flow-sensitive approach give us a more precise follow-up for flow signatures but it is also considerably more expensive in terms of computational time and space. According to our experience the analysis time is actually expensive. Nevertheless, given our two step approach, only the offboard analysis will suffer because of the the complexity of flow-sensitive analysis. For the onboard phase, verification relies on the proof elements and is linear in time and space, hence it does not depend on the control flow.

Thus, using the flow-sensitive analysis, we obtain more precise flow signatures without introducing any overhead for the onboard verification.

Control dependency regions

To determine implicit flows and thus the scope of conditional instructions, we use existing techniques [LT79], using the control flow graph of a program, postdominators and immediate postdominators, which can be easily applied, as has already been done in [BPR07, BFLM04]. The low complexity of these algorithms is adequate for the embedded verification of the implicit flows. In other work on information flow [Mye99a], and especially in type-based systems, a special label pc holding the security level of the program counter is being used.

Considering the instruction list P_m of a method m, the intra-method control flow graph CF_m is defined as usual: CF_m is a graph with vertices in $P_m \cup \{exit\}$ and for each $i \in P_m$ edges from i to each element in $succ(i)$, where $succ(i)$ is defined according to the bytecode semantics as:

- $succ(exit) = \emptyset$,
- $succ(i) = \{exit\}$ if $P_m[i] = \texttt{areturn}$,
- $succ(i) = \{a\}$ if $P_m[i] = \texttt{goto}\ \ a$,
- $succ(i) = \{a, i+1\}$ if $P_m[i] = \texttt{ifeq}\ \ a$,
- $succ(i) = \{i+1\}$ otherwise.

We introduced a special node $exit$, with no successors, as the exit point of the method. We also use the notation $pred(i) = \{j \mid i \in succ(j)\}$.

We use the notion of postdominance as defined in [Bal93]: in a control flow graph CF_m of a method m, a node n' postdominates a node n if $n' \neq n$ and n' belongs to every path from n to $exit$. We denote $PD(n)$ the set of postdominators of n. The immediate postdominator of n, $ipd(n) \in PD(n)$ satisfies that $\forall n'' \in PD(n)$, if $n'' \neq ipd(n)$, then $n'' \in PD(ipd(n))$.

If instruction i is a conditional instruction ($P_m[i] = \texttt{ifeq}\ \ a$ or $P_m[i] = \texttt{ifnull}\ \ a$), $ipd(i)$ is the first instruction that will be executed independently from the conditions tested by i, thus it is the

```
void discount(Hertz h) {
  int level = h.getLevel();
  if(level == GOLD)
    discount = 20;
}
```

```
 0: aload 1
 1: invoke getLevel
 2: istore 2
 3: iload 2
 4: ifeq 6
 5: goto 9
 6: aload 0
 7: bipush 20
 8: putfield
 9: return
```

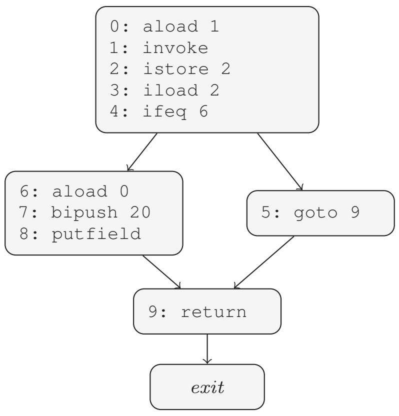


Figure 3.5: Control flow graph example

first instruction belonging to all the branches originated by the conditional instruction. We define the *control dependency region* of a conditional instruction i as the set of instructions executed under its condition:

$$cdr(i) = Reach_{CF_m}(i) \setminus Reach_{CF_m}(ipd(i)),$$

where $Reach_G(u)$ denotes the set of vertices reached by the vertex u in a graph G, with $u \in Reach_G(u)$. Informally, all instructions in a path from i to $ipd(i)$ belong to the dependency region of i.

We compute a function $cxt : P_m \rightarrow \wp(P_m)$ representing the context (the set of conditional bytecodes) under which each instruction is executed: $cxt(i) = \{j \mid i \neq j \wedge i \in cdr(j)\}$. Instruction i may or may not be executed, depending on the condition tested by instructions in $cxt(i)$.

Figure 3.5 depicts the control flow graph of method `discount`. The execution of instructions `5, 6, 7` and `8` depends on the conditional instruction `4`, thus they belong to the control dependency region of `4`: $cdr(4) = \{5, 6, 7, 8\}$. The computation of cxt function is straightforward: $cxt(5) = cxt(6) = cxt(7) = cxt(8) = \{4\}$.

We will denote by Γ_i the set of abstract values tested by instructions in $cxt(i)$. Hence, all abstract values manipulated by instruction i depend implicitly on values in Γ_i.

Abstract execution state

The intra-method analysis consists in an abstract interpretation of the method instructions (Java bytecode). We use a classical operational semantics, which we will only sketch here. The JVM behaviour is simulated using the abstract values $\ddot{\Sigma}^L_m$. For a method m of a class C, n_m denotes the number of arguments of the method, P_m the instruction list and χ_m the local variables.

An abstract state of the JVM is a frame $(\rho, s, \ddot{S})$ where s denotes the local operand stack with values in $\wp(\ddot{\Sigma}^L_m \times \mathcal{F})$, $\rho : \chi_m \rightarrow \wp(\ddot{\Sigma}^L_m \times \mathcal{F})$ the local variables, and $\ddot{S} \in \ddot{\mathcal{S}}_m$ the flow signature.

Building the flow signature requires to approximate local variables and stack contents, and to deal with implicit flow, we must know the conditions under which the local variables and the stack are modified. Thus, elements from stack and local variables have the form (a, ϕ), where $a \in \ddot{\Sigma}^L_m$ and $\phi \in \mathcal{F}$. Let $\mathcal{Q}$ be the property space of frames in a method.

We consider the function Q which associates an abstract state to every instruction; given a method m and $i \in P_m$, Q_i denotes the state associated with instruction i and represents *the state before* *executing the instruction*. If $Q_i = (\rho_i, s_i, \ddot{S}_i)$, then the flow signature $\ddot{S}_i$ is the set of flows realized along an execution path from the entrance of the method to the program point i.

Our algorithm is a forward data flow may-analysis [NNH99], as the computed frame Q_i for a given program point $i \in P_m$ is the union of frames created by all the execution paths reaching that point.

Union relation on abstract states Given two states $Q' = (\rho', s', \ddot{S}')$ and $Q'' = (\rho'', s'', \ddot{S}'')$ from the property space $\mathcal{Q}$, their union $Q' \sqcup Q''$ is defined as follows:

$$Q' \sqcup Q'' = (\rho' \sqcup \rho'', s' \sqcup s'', \ddot{S}' \sqcup \ddot{S}'').$$

We assume the code was previously checked by the JVM bytecode verifier and so it is well typed. This implies that the height of the stack is statically known at each instruction point, meaning that the merge operation is always performed only on stacks having the same heights. The union of two stacks and the union of local variables sets is an element by element operation. The result of $\ddot{S}' \sqcup \ddot{S}''$ is the least upper bound signature between $\ddot{S}'$ and $\ddot{S}''$ according to the lattice of flow signatures Λ.

Moreover, based on the order relation for flow signatures, we can define an order relation on abstract states:

Order relation on abstract states Given two abstract states $Q', Q'' \in \mathcal{Q}$, Q' is *smaller than* Q'', denoted by $Q' \sqsubseteq Q''$, if $s' \sqsubseteq s''$ (the stack s'' contains at least all elements in s'), $\rho' \sqsubseteq \rho''$ and $\ddot{S}' \leq \ddot{S}''$.

Abstract semantics and algorithm

The abstract interpretation relies on a set of transformation rules. The most significant bytecodes and their rules are presented in Figure 3.6. The concatenation $n :: s$ denotes a stack having n on top, f_C designates the field f of the class C, while $f' = f[x \mapsto e]$ is a function which agrees with f on all elements except for x.

Algorithm 3.1 $analyse(m)$: Intra-procedural analysis of a method m

```
compute $cxt(i), \forall i \in P_m$
$Q_0 = (\{0 \mapsto pr_0, \ldots, n_m \mapsto pr_{n_m}\}, \epsilon, \emptyset)$, with $pr_j = (p_j^{p,s}, \mathbf{r})$ if $p_j \in P_{|Obj}$
                                          or $pr_j = (p_j^{p}, \mathbf{v})$ if $p_j \in P \setminus P_{|Obj}$
$Q_i = (\{0 \mapsto \emptyset, \ldots, n_m \mapsto \emptyset\}, s^i, \emptyset), \forall i \in P_m \setminus \{0\}$
$PC = \{0\}$
while $PC \neq \emptyset$ do
    extract $i$ from $PC$
    $\Gamma_i = \{e \mid Q_j = (\rho, (e, \phi) :: s, \ddot{S}) \ \wedge \ j \in cxt(i))\}$
    apply the embedded semantics of $P_m[i]$
    for all $j$ in $succ(i)$ do
        if $Q_j$ has changed then
            $PC = PC \cup \{j\}$
        end if
    end for
end while
return $S_{exit}$
```

The algorithm, depicted by Algorithm 3.1, starts with computing the control dependency regions and the context of each instruction and populating the initial states. In the initial abstract state Q_0, the method parameters must be mapped to the local variables, while the stack and the flow

$$\frac{P_m[i] = \texttt{prim}\ op \qquad Q_i = (\rho, v_1 :: v_2 :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, v_1 \cup v_2 \cup TV_{\Gamma_i} :: s, \ddot{S})} \qquad \frac{P_m[i] = \texttt{pop} \qquad Q_i = (\rho, v :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, s, \ddot{S})}$$

$$\frac{P_m[i] = \texttt{bipush}\ \ n \qquad Q_i = (\rho, s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, \{(Const^p, \mathbf{v})\} \cup TV_{\Gamma_i} :: s, \ddot{S})} \qquad \frac{P_m[i] = \texttt{aconst\_null} \qquad Q_i = (\rho, s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, \{(\texttt{null}^p, \mathbf{v})\} \cup TV_{\Gamma_i} :: s, \ddot{S})}$$

$$\frac{P_m[i] = \texttt{ifeq}\ \ a \qquad Q_i = (\rho, v :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, s, \ddot{S}) \qquad Q_a = Q_a \sqcup (\rho, s, \ddot{S})} \qquad \frac{P_m[i] = \texttt{ifnull}\ \ a \qquad Q_i = (\rho, v :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, s, \ddot{S}) \qquad Q_a = Q_a \sqcup (\rho, s, \ddot{S})}$$

$$\frac{P_m[i] = \texttt{new}\ \ C \qquad Q_i = (\rho, s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, \{(n_i^{s,p}, \mathbf{r})\} \cup TV_{\Gamma_i} :: s, \ddot{S})} \qquad \frac{P_m[i] = \alpha\texttt{newarray}\ \ C \qquad Q_i = (\rho, s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, \{(n_i^{s,p}, \mathbf{r})\} \cup TV_{\Gamma_i} :: s, \ddot{S})}$$

$$\frac{P_m[i] = \alpha\texttt{load}\ \ x \qquad Q_i = (\rho, s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, \rho(x) \cup TV_{\Gamma_i} :: s, \ddot{S})} \qquad \frac{P_m[i] = \alpha\texttt{store}\ \ x \qquad Q_i = (\rho, v :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho[x \mapsto v \cup TV_{\Gamma_i}], s, \ddot{S})}$$

$$\frac{P_m[i] = \texttt{getfield}\ \ f_{C'} \qquad Q_i = (\rho, v :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, (v_{|f_{C'}}, \mathcal{T}(f_{C'})) \cup v_{|\mathbf{v}} \cup v_{|\mathbf{i}} \cup TV_{\Gamma_i} :: s, \ddot{S})}$$

$$\frac{P_m[i] = \texttt{putfield}\ \ f_{C'} \qquad Q_i = (\rho, v :: u :: s, \ddot{S}) \qquad \phi' = \phi \sqcap \mathcal{T}_{\mathcal{F}}(f_{C'})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, s, closure(\ddot{S} \sqcup \{u_{|f_{C'}} \xrightarrow{\phi'} e \mid (e, \phi) \in v\} \sqcup \{u_{|f_{C'}} \xrightarrow{\mathbf{i}} V_{\mathbf{i}}^u \cup \Gamma_i\}))}$$

$$\frac{P_m[i] = \texttt{getstatic}\ \ f_{C'} \qquad Q_i = (\rho, s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, \{(Static^p, \mathcal{T}_{\mathcal{F}}(f_{C'}))\} \cup TV_{\Gamma_i} :: s, \ddot{S})}$$

$$\frac{P_m[i] = \texttt{putstatic}\ \ f_{C'} \qquad Q_i = (\rho, v :: s, \ddot{S}) \qquad \phi' = \phi \sqcap \mathcal{T}_{\mathcal{F}}(f_{C'})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, s, closure(\ddot{S} \sqcup \{Static^p \xrightarrow{\phi'} e \mid (e, \phi) \in v\} \sqcup \{Static^p \xrightarrow{\mathbf{i}} \Gamma_i\}))}$$

$$\frac{P_m[i] = \alpha\texttt{aload} \qquad Q_i = (\rho, v :: u :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, u \cup v \cup TV_{\Gamma_i} :: s, \ddot{S})}$$

$$\frac{P_m[i] = \alpha\texttt{astore} \qquad Q_i = (\rho, v :: w :: u :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, s, closure(\ddot{S} \sqcup \{V_{\mathbf{r}}^u \xrightarrow{\phi} e' \mid (e', \phi) \in v\} \sqcup \{V_{\mathbf{r}}^u \xrightarrow{\mathbf{i}} V_{\mathbf{i}}^u \cup \Gamma_i\} \sqcup \{V_{\mathbf{r}}^v \xrightarrow{\mathbf{i}} V_{\phi}^w\}))}$$

$$\frac{P_m[i] = \texttt{goto}\ \ a \qquad Q_i = (\rho, s, \ddot{S})}{Q_a = Q_a \sqcup (\rho, s, \ddot{S})} \qquad \frac{P_m[i] = \texttt{arraylength} \qquad Q_i = (\rho, u :: s, \ddot{S})}{Q_a = Q_a \sqcup (\rho, u \cup TV_{\Gamma_i} :: s, \ddot{S})}$$

with $\phi, \phi' \in \mathcal{F}$ $\quad V_{\phi}^u = \{e \mid (e, \phi) \in u \wedge e \neq \texttt{null}^p\}$ $\quad u_{|\phi} = \{(e, \phi) \mid (e, \phi) \in u\}$ $\quad TV_{\Gamma_i} = \{(e, \mathbf{i}) \mid e \in \Gamma_i\}$
$u_{|f_{C'}} = \{approx(e, f_{C'}) \mid e \in V_{\mathbf{r}}^u\}$ $\quad \mathcal{T}_{\mathcal{F}}(f_{C'})$ *gives the type of field* $f_{C'}$ *(**v** or **r**) with respect to* $\mathcal{F}$.

Figure 3.6: A subset of information flow transformation rules

signature are empty; ϵ denotes the empty stack and $pr_j \in \ddot{\Sigma}^L_m$ stands for the jth parameter of the method. The rest of the states [2] Q_i are initialized with a stack and local variable array populated with empty sets of elements and with an empty signature $\ddot{S} = \emptyset$ (we maintain the JVM property stating that the stack in a program point has always the same height, as s^i denotes a stack of empty elements and its height is equal to the statically computed height at instruction i). Next, the algorithm consists of applying the rule corresponding to each instruction and performing a fixed point iteration on the instructions set of the method, starting with the instruction at $P_m[0]$. The set PC holds the intructions to be executed (it is initialized with $\{0\}$). The successors of the current instruction i are going to be executed (and added to PC) only if their abstract state has changed.

The transformation rules are monotone as *(1)* we do not perform strong updates on flow-signatures (*i.e.*, flows are added and never deleted) and we perform weak updates on local variables and operand stack (*i.e.*, local variable updates erase previous content) and *(2)*, at each step, we make the union between the already computed state and the current state. The may-analysis, weak updates, the monotonicity of transformation rules, the finite set of abstract values and the finite lattice of flow signatures ensure that there is a the fixed point and that it is reached by the analysis.

The frame Q_{exit} represents the state reached when exiting the method (*e.g.*, after the execution of the `return` bytecode). If $Q_{exit} = (\rho_{exit}, s_{exit}, \ddot{S}_{exit})$, the *final flow signature* of a method m, S_m, is the restriction of signature $\ddot{S}_{exit}$ to elements in Σ_m.

Transformation rules examples

To reflect the impact of control regions on stack and local variables array, every instruction modifying the last two (push for stack or store for local variables array) takes into consideration the values in the context. For example, the instructions `new` and `bipush` push new abstract values on the stack as well as the context under which the operation takes place, meaning elements of type $(e, \mathtt{i})$ with $e \in \Gamma_i$. The α`store` bytecode stores the top of the stack in the local variables array, as well as elements $(e, \mathtt{i})$ with $e \in \Gamma_i$.

Considering that v is the element on top of the stack before execution, the instruction `getfield` performs the following operations:

- pushes the location of $v.f$ on the stack, if v abstracts an object,
- keeps v on the stack if it abstracts a primitive value or arising from implicit flow ($v_{|\mathtt{v}}$ and $v_{|\mathtt{i}}$),
- and pushes the values $(e, \mathtt{i})$ in the context ($e \in \Gamma_i$), as the execution depends on values in Γ_i.

The most significant bytecode is `putfield`, as it generates information flows:

- from e such that $(e, \phi') \in v$ to object abstractions in u (denoted by $u_{|f_{C'}}$). The type of flow is the least upper bound corner between ϕ and the type of the field, according to the lattice of flows, Θ,
- from e such that $(e, \mathtt{i}) \in u$ to object abstractions in u; The presence of elements like $(e, \mathtt{i})$ in u signifies that the abstract values in u depend on e, most likely because the stack was modified under the condition e,
- from abstract values in Γ_i to abstract values in $u_{|f_{C'}}$.

Our analysis is field-independent but security level sensitive, as described in Section 3.2.1: all the fields of an object having the same security level are modeled as having the same location. Hence, given an abstraction of an object and a field f, we must compute its abstract location, according to

[2]To optimize, our implementation stores only the states associated to jump points.

Equation 3.2.2. For $u \in \ddot{\Sigma}^L_m$ and a field f, the function $approx_{\mathcal{L}} : \ddot{\Sigma}^L_m \times Fields \rightarrow \ddot{\Sigma}^L_m$ computes the abstract location of $u.f$ according to Equation 3.2.2 and to the security level of field f, defined by $\mathcal{L}$:

$$approx_{\mathcal{L}}(u^l, f_C) = \begin{cases} u^l & \text{if } \mathcal{L}(f, C) = p \\ u^s & \text{if } \mathcal{L}(f, C) \neq p \end{cases}$$

For simplification, we may also apply $approx$ on a set $U \subseteq \ddot{\Sigma}^L_m$ where $approx_{\mathcal{L}}(U, f_C) = \{approx_{\mathcal{L}}(u, f_C) \mid u \in U\}$.

By $closure$ we denote a function that computes the transitive closure on the set of flows with propagation of the p and s security levels, and of the type of flow ($\mathbf{r}$, $\mathbf{v}$ or $\mathbf{i}$). The closure of a flow signature must be computed after every flow creation, as the order in which flows are generated and the type of flow influence the propagation. For the example in Figure 3.3b, the function $closure$, called after adding the flow $p^p_2 \xrightarrow{\mathbf{v}} p^p_3$, will generate, due to aliasing between o_1 and o_2 ($p^p_1 \xrightarrow{\mathbf{r}} p^p_2$), the flow $p^p_1 \xrightarrow{\mathbf{v}} p^p_3$.

The function $closure$ also handles flows potentially generated by encapsulation and by our field-independent approach, which models all the fields in a structure as having the same abstract location. Let us consider the example below:

```
 static void m(A[] o, A o1, B o2){
    o[1] = o1;
    o[1].g = o2;
}
```

The first assignment generates a flow from `o1` to `o` ($p^p_0 \xrightarrow{\mathbf{r}} p^p_1$), the second assignment generates a flow from `o2` to `o` ($p^p_0 \xrightarrow{\mathbf{r}} p^p_2$), as well as a flow from `o2` to `o1` ($p^p_1 \xrightarrow{\mathbf{r}} p^p_2$) as `o2` is stored in the field `g` of `o1`. These kind of flows are detected by function $closure$, which creates flows from all abstract values pointed by `o` (abstract values e such that there exists a flow $p^{p,s}_0 \xrightarrow{\mathbf{r}} e$) to `o2`.

3.3.2 Inter-procedural analysis

We saw how a simple method is analyzed, without any method call. Now we add support for method invocation and take into consideration the analysis of a group of classes. We assume that the method to be invoked is statically known; we add support for virtual invocation while we discuss open world issues in Section 3.5.

Composing flow signatures

In an open and dynamic loading context, the target methods are not available, and can vary in time. Hence, a compositional analysis is mandatory in such a context. Moreover, as in an open world the call graph is not available, the flow signatures we compute are context-insensitive, meaning that they are independent on the context under which the method might be invoked.

Invoking a method `m'` in `m` consists in *(1)* mapping the arguments to parameters of `m'` and *(2)* transposing flows between parameters in `m'` in flows between mapped values in `m`. This operation is performed by the function $apply$. The mapping must take into consideration the p and s security levels, as well as the conditions under which the local variables array and stack have been modified. The semantics rules for `invoke` and α`return` bytecode are depicted in Figure 3.7. Note that `return` bytecode generates flows between abstract value R and elements in the execution context (Γ_i), while `areturn` also generates flows between R and elements on top of the stack. Semantics rule for `ireturn` is similar to `areturn`, only that the type of flow , ϕ' is computed as $\phi' = \phi \sqcap \mathbf{v}$

$$\frac{P_m[i] = \texttt{invoke}\ \ m'_{C'} \qquad Q_i = (\rho, v_{n_m} :: .. :: v_1 :: u :: s, \ddot{S}) \qquad S_{m'} = \mathcal{R}(m')}{Q_{i+1} = Q_{i+1} \sqcup (\rho, \{(r_i^{p,s}, \mathbf{r})\} :: s, closure(\ddot{S} \sqcup apply(\ddot{S}, S_{m'}, u, v_1, .., v_{n_m}))}$$

$$\frac{P_m[i] = \texttt{areturn} \qquad Q_i = (\rho, v :: s, \ddot{S}) \qquad \phi' = \phi \sqcap \mathbf{r}}{Q_{exit} = Q_{exit} \sqcup (\emptyset, \epsilon, closure(\ddot{S} \sqcup \{R^p \xrightarrow{\phi'} e \mid (e, \phi) \in v\} \sqcup \{R^p \xrightarrow{\mathbf{i}} \Gamma_i\})}$$

$$\frac{P_m[i] = \texttt{return} \qquad Q_i = (\rho, v :: s, \ddot{S})}{Q_{exit} = Q_{exit} \sqcup (\emptyset, \epsilon, closure(\ddot{S} \sqcup \{R^p \xrightarrow{\mathbf{i}} \Gamma_i\})}$$

$\mathcal{R}$ associates a flow signature to each method. $\qquad \phi, \phi' \in \mathcal{F}$

Figure 3.7: Information flow transformation rules for inter-procedural analysis.

Algorithm

The flow signature of a method depends on the flow signatures of invoked methods. The flow signature of the calling method is correct only when the flow signatures of invoked methods are correct. If there are no mutually recursive methods, we can define an order on methods and the analysis could be achieved in only one step. But, in general, this is not possible because recursive methods or inter-dependent methods are frequent in a software unit. That is why we must perform an external analysis in many iterations. The flow signatures are computed simultaneously for a set of classes (*e.g.*, an API, an application, a JAR file, a bundle), which permits more accurate results and a strict control on overriding.

Algorithm 3.2 $analyse(Cl)$: Inter-procedural analysis of a set of classes Cl

```
$S_m = \emptyset, \forall m$
$change$ = true
while $change$ == true do
    $change$ = false
    for all $C \in Cl$ do
        for all $m \in C$ do
            $S = analyse(m)$
            if $S \neq S_m$ then
                $S_m = S$
                $change$ = true
            end if
        end for
    end for
end while
```

The analysis, depicted by Algorithm 3.2 is incremental: at the beginning, the flow signature associated with each method m is empty $S_m = \emptyset$; during the analysis of Java bytecode, the newly-found flows are added to the existing signatures. We perform weak updates: flows are added to signatures, but never deleted. The algorithm is iterated for the entire set of analyzed classes [3], until all the methods have the correct flow signatures, which means until a fixed point has been reached. A fixed point exists because transformation rules are monotone (flows are only added and never

[3]To optimize, our implementation reduces at each step the set of methods on which we must iterate; only methods that have not reach their "final" flow signature are kept.

```
class Ex extends Exception{
   int f;
   ...
}

class A {
   int secret;
   int public;
   ...
}
```

(a) Explicit flow

```
class A {
   ...
1:   Ex e;
2:   try{
3:      e = new Ex();
4:      e.f = secret;
5:      throw e;
6:   } catch(Ex e1) {
7:      public = e1.f;
8:   }
   ...
}
```

(b) Implicit flow

```
class A {
   ...
1:   try{
2:      if(secret)
3:         throw new Ex();
4:   } catch(Ex e) {
5:      public = 1;
6:   }
   ...
}
```

Figure 3.8: Information flow through exceptions

deleted), and the flow signatures form a finite lattice.

To have correct and precise results, our algorithm permits to annotate manually native methods and inserting their flow signatures in the dictionary, before the offline analysis is executed. Native methods without annotations are considered insecure. In this case, a default flow signature, corresponding to the most pessimistic case, where everything flows to everything, is associated with them. This creates a loss of precision but prevents any leakage. We detect all the leaks through I/O, as native methods are the only place where they may appear. The result of the analysis is correct with respect to manually annotated methods.

Manually annotating methods signatures also allows to express expected behaviours for different methods. When writing abstract or interface methods, adding constraints for methods implementing them can be easily done through manual annotation. This not only imposes restrictions on third-party code, but also ensures that only code meeting established constraints will be allowed to interact with existing software units.

3.3.3 Particular cases

Exceptions

Problems with exceptions Exceptions represent a source of information leak in Java programs and must be treated carefully. An exception disrupts the normal flow of instructions and jumps in the current method according to the `try-catch` statement or terminates the execution of the method and returns itself to the caller. Exceptions can be raised explicitly by the `athrow` bytecode.

An exception raised during the execution of a program can disclose private information to an attacker, both through explicit flows and implicit flows. Explicit flows occur when the thrown object contains sensitive data and it is "sent" to the statement where the exception is catched. This situation is depicted in Figure 3.8a, where the exception `e` is used as a buffer to store secret field `secret` in the public field `public`. Implicit flows occur if the exception is thrown in one of the branches of an conditional instruction. For example, in Figure 3.8b, the fact that the exception is or not thrown reveals information about `secret`.

Exceptions can either be caught within the method raising the exception, using a `try-catch` statement (as we have seen in example in Figure 3.8), or not caught (unchecked exceptions) and the leak of information is thrown to the caller. For example, if lines `2`, `6-8` in Figure 3.8a are not present, then the exception is unchecked.

Treating exceptions is even more difficult, apart from exceptions thrown explicitly using the `athrow` bytecode, since the JVM also supports runtime exceptions. Runtime exceptions are the result of a programming problem, *e.g.*, arithmetic exceptions (such as a division by zero), pointer exceptions (such as trying to access an object through a null reference) and indexing exceptions (such as attempting to access an array element through an index that is too large or too small). For example, the assignment `x = y/z` can throw a division by zero exception if the runtime value of `z` is 0. Moreover, if `z` is a secret value, the raised exception will disclose its value. Even if their number is more significant, runtime exceptions generate only the weakest form of interference (implicit flows). For example `NullPointerException` is raised when a reference is null, hence no explicit flow can be transmitted, but only an implicit one (*e.g.*, the reference is null because the sensitive data is equal to 1).

Simplifying assumptions Exceptions, and runtime exceptions in particular, are a result of a programming problem; when it deals with an exception, the software tries to recover from an abnormal execution state to a normal execution state. The answer to an exception should not be at application level, and the exceptions treatment by an application should not interfer with public and secret data manipulated by the software unit. Hence, we make the assumption that there is no distinction between exceptions treatment: we use an abstract value Ex which stands for all thrown objects. Throwing an object as exception makes the object available for public use.

Solution Treating the catched exceptions statically is straightforward: an edge from the instruction throwing the exception (`athrow`) to the matching exception handler is added to the control flow graph of the method. There can be more than one exception handler matching the thrown exception. The order in which the exception handlers of a method are searched for a match is important and it depends on the control flow and on the runtime type of the exception. As the runtime type is not statically known, then an edge from the intruction throwing the exception to each handler is added.

As said before, we consider that exceptions put all thrown object at the disposal of the entire system, through an abstract value name Ex. Hence, throwing an object stored in the abstract value u (with the `athrow` bytecode) results in a flow $Ex^p \xrightarrow{t} u$, as well as in a flow from Ex to elements in the context of the instruction:

$$\frac{P_m[i] = \texttt{athrow} \qquad Q_i = (\rho, \{(u,t)\} :: s, \ddot{S}) \qquad j \in \mathit{handler}(\mathcal{T}(u), i)}{Q_j = Q_j \sqcup (\rho, \{(u,t)\} :: \epsilon, \mathit{closure}(\ddot{S} \sqcup \{Ex^p \xrightarrow{t} u\} \sqcup \{Ex^p \xrightarrow{\mathtt{i}} \Gamma_i\}))}$$

where $handler$ returns the list of handlers matching the thrown exception. Moreover, using the transitive closure, all abstract values that flow to u will flow to Ex, too.

Handling unchecked exceptions could be done in a similar way: an edge from the instruction raising the exception to the exit point of the program must be added to the control flow graph, and a flow from Ex to the abstract value generating the exception (for example, the assignment `x = y/z` generates an exception if `z` is equal to zero and hence leads to a flow from Ex to the abstraction of `z`). Unfortunately, the number of bytecode raising runtime exceptions is significant, thus many instructions will have as immediate postdominant the $exit$ node. This will lead to many flows of information. Moreover, such behaviour should not occur in a small open system. Thus, our model accounts for catched exceptions, but does not handle unchecked exceptions. This limitation applies to all the other information flow systems for Java [Mye99a, CF07].

Threads and synchronization

Multithreading, which consists of simultaneous manipulation of shared objects, represents a threat for confidentiality of shared secrets as (*1*) flows can be missed by the analysis (*e.g.*, a thread A can write sensitive data in a shared object and a thread B can read it) and (*2*) information can be inferred from synchronization (*e.g.*, some piece of code is synchronized in a branch of a conditional instruction of a thread A, then a thread B trying to synchronize its execution may infer information about the condition tested by A).

Object sharing There are three ways in which objects can be shared between threads:

1. through a static field,
2. through the *sharable* service offered for example by JAVACARD. In this case, a sharable object must be registered using the API, and a card applet can obtain the object through the `getShareable()` method,
3. through the field of a thread set by the thread which created it.

In the first case, storing a sensitive data in a shared object (a static field) will lead to a flow from *Static* to the shared object, hence from security level p to security level s; this flow is detected by our analysis.

In the second case, the API is either implemented natively and our analysis is not able to compute the flow signatures, hence we can declassify them manually, or is implemented in Java using a sort of static hashset, hence using the API will "store" the sharable object in a static field. This case is also correctly handled by our analysis.

The third possibility to share objects proves to be more complicated. Threads are implemented as a part of the system, the JVM, and are accessible thought API methods. Hence, we make the following assumption:

Thread-safety policy All objects of type `Thread` are static objects.

We need to redefine the abstract semantics of `new`, such that, when a thread is created, our analysis attaches it to the abstract value *Static*:

$$\frac{P_m[i] = \texttt{new}\ C \qquad Q_i = (\rho, s, \ddot{S}) \qquad \mathcal{T}(C) \leq \mathit{Thread}}{Q_{i+1} = Q_{i+1} \sqcup (\rho, \{(n_i^{s,p}, \mathbf{r})\} \cup \mathit{TV}_{\Gamma_i} :: s, \mathit{closure}(\ddot{S} \sqcup \{\mathit{Static}^p \xrightarrow{\mathbf{r}} n_i^{s,p}\} \sqcup \{\mathit{Static}^p \xrightarrow{\mathbf{i}} \Gamma_i\}))}$$

All subsequent computations involving threads (*e.g.*, storing a secret value to the field of a thread) will generate, through the transitive closure, flows to *Static*. As a thread is a static object, the following corollary is obvious:

Thread-safety corollary All the fields of a class extending `Thread` are public fields.

Thread-safety is ensured by the abstract value *Static* modeling static fields: sensitive data flowing to static fields and susceptible to be accessed through different threads is considered as leaked. We now show how thread-safety is ensured by this property, both for problems arising from *thread* flows and from synchronization.

Thread flows An example of thread flows is showed in Figure 3.9. `ThreadA` creates a thread `ThreadB` (at line `1` in Figure 3.9a) and shares the object stored in field `f` (at line `2`). Later, at line `i`, `ThreadA` stores a sensitive data in the field `s` of `f`. Let us suppose that `ThreadB` reads the field `f.s` and stores it in a public field `p` after `ThreadA` has stored the sensitive data (line `j` in Figure 3.9b). Hence, a flow from sensitive data stored in field `secret` to the field `p` of `ThreadB` occurs. Our analysis does not account directly for such flows.

Generally speaking, considering that o is a shared object, and `ThreadA` creates flows of type $o \rightarrow o_1$ while `ThreadB` creates flows of type $o_2 \rightarrow o$, as depicted in Figure 3.10, then flows $o_2 \rightarrow o_1$ are not directly visible in our analysis. The illegal flow arises when o_1 contains sensitive data. But our thread-safety policy saves us, as the sharing object is attached to the thread, and hence to *Static*: $\mathit{Static} \rightarrow o$. By transitive closure, we have $\mathit{Static} \rightarrow o_1$. The illegal flow will be detected when analysis `ThreadA` and finding the flow $\mathit{Static}^p \rightarrow o_1^s$. Hence, our analysis only fails to show flows between data that is attached to *Static*, which is already considered as illegal.

Notice that the thread `t` created in `ThreadA.run` is local to the method, hence all flows to `t` are not present in the final flow signature of `ThreadA.run`. Nevertheless, as `t` flows to *Static*, all values flowing to `t` also flow to *Static*, and they are not lost.

(a) ThreadA

```
class ThreadA extends Thread{
   A f;
   int secret;
   public void run(){
   1:  t = new ThreadB();
   2:  t.f = f;
   3:  t.start();
       ...
   i:  f.s = this.secret;
       ...
   }
   ...
}
```

(b) ThreadB

```
class A {
   int s;
}

class ThreadB extends Thread{
   A f;
   int p;
   public void run(){
       ...
   j:  this.p = f.s;
   }
   ...
}
```

Figure 3.9: Multithreading example: flows detected indirectly by the analysis

ThreadA	$o_1 \rightarrow o$
ThreadB	$o \rightarrow o_2$
Thread flows	$o_1 \rightarrow o_2$

Figure 3.10: Multithreading: thread flows

Synchronization through monitors The mechanism that Java uses to support synchronization is the monitor. A monitor is basically a guardian in that it watches over a sequence of code, making sure only one thread at a time executes the code. Each monitor is associated with an object reference. When a thread arrives at the first instruction in a block of code that is under the watch of a monitor, the thread must obtain a lock on the referenced object; this is implemented by the `monitorenter` bytecode or by a flag `synchronized` set in a method descriptor. The thread is not allowed to execute the code until it obtains the lock. Once it has obtained the lock, the thread enters the block of protected code. When the thread leaves the block, it releases the lock on the associated object, by using the `monitorexit` bytecode. In the case of `synchronized`, the action usually performed by `monitorenter` is performed by the `invoke` bytecode on the first argument, `this`, or on the class for `invokestatic`.

In the example in Figure 3.11, `ThreadA` locks the monitor in a branch of a conditional instruction (line `i` in Figure 3.11a). The thread `ThreadB` tries to access the monitor (line `j` in Figure 3.11b) and it may infer information about the condition tested by `ThreadA` if the monitor is locked.

In order to detect such leaks, we add semantics rules for `monitorenter` and `monitorexit` bytecode:

`monitorenter`	pop the top of the stack and acquire the lock associated with this object
`monitorexit`	pop the top of the stack and release the lock associated with this object

$$\frac{P_m[i] = \texttt{monitorenter} \qquad Q_i = (\rho, (u,t) :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, s, closure(\ddot{S} \sqcup \{u \xrightarrow{\mathtt{i}} \Gamma_i\}))} \qquad \frac{P_m[i] = \texttt{monitorexit} \qquad Q_i = (\rho, (u,t) :: s, \ddot{S})}{Q_{i+1} = Q_{i+1} \sqcup (\rho, s, \ddot{S})}$$

The semantics rule for `monitorenter` adds an implicit flow from the monitor (u) to the abstract value in the context (Γ_i). In the example in Figure 3.11a, a flow from the abstraction of `this.f` to the abstraction of `this.secret` is added. As the monitors are shared objects, they are of type `Thread` they flow to an object of type `Thread`, which flows to *Static*. Then, monitors also flow to *Static*. Hence, our analysis will detect implicit flows arising from synchronization when analysis the thread which manipulates the sensitive data.

(a) ThreadA

```
class ThreadA extends Thread{
   A f;
   int secret;
   public void run(){
   1:  t = new ThreadB();
   2:  t.f = f;
   3:  t.start();
       ...
   i:  if(this.secret)
   i+1:   synchronize(this.f){
             ...
       ...
       }
   }
   ...
}
```

(b) ThreadB

```
class A {
   int s;
}

class ThreadB extends Thread{
   A f;
   int p;
   public void run(){
       ...
   j:  synchronize(f) {
   j+1:   this.p = 1;
       }
       ...
   }
   ...
}
```

Figure 3.11: Multithreading example: implicit flow through monitor observation

$FlyFrance:$	S_{update}	$=$	$\{p_1^s \xrightarrow{\mathtt{v}} p_0^s\}$
$MHz:$	S_{update}	$=$	$\{p_0^s \xrightarrow{\mathtt{v}} p_1^p\}$
$MHz:$	$S_{getLevel}$	$=$	$\{R^p \xrightarrow{\mathtt{i}} p_0^{p,s}, R^p \xrightarrow{\mathtt{i}} Static^p\}$
$FlyMaroc:$	$S_{discount}$	$=$	$\{p_0^{p,s} \xrightarrow{\mathtt{i}} p_1^{p,s}, p_0^{p,s} \xrightarrow{\mathtt{i}} Static^p\}$

Figure 3.12: Flow signatures for LoyaltyCard

Synchronization through `wait()` and `notify()` Another way to protect against concurrent access problems and to communicate between threads is to use the `wait()` and `notify()` primitives defined in class `Object`. In addition to having a lock that can be grabbed and released, each object has a system that allows it to pause or wait whilst another thread takes over the lock. To ensure that the current code owns the monitor, wait and notify must be placed within synchronized code.

From the point of view of software security, `wait()` and `notify()` represent another way to leak information. From our point of view, the safety comes from the fact that these instructions must be placed inside synchronized code, hence between a `monitorenter` and `monitorexit`. Implicit flows that may occur due to this synchronization mechanism are detected in the same way as for monitors (in fact, monitors include `wait()` and `notify()`).

Conclusion Due to the thread-safety policy, our analysis accounts for leaks arising from multi-threading environments, but with a price to pay. As a consequence of our thread-safety management policy, we forbid any exchange of sensitive data between threads. Moreover, all secrets attached to a thread leak through the abstraction of static variables, *Static*.

3.3.4 LoyaltyCard Example

Let us reconsider the LoyaltyCard example, explained in Section 2.3.3. Confidential data (loyalty points of FlyFrance) reside in fields `FlyFrance.miles` and `MHz.ppoints`. Thus, the information flow analysis is performed with security levels: $\mathcal{L}(FlyFrance, miles) = s$ and $\mathcal{L}(MHz, ppoints) = s$. The final flow signatures of methods in LoyaltyCard are depicted in Figure 3.12.

(a) Source code

```
void discount(MHz h) {
  int level = h.getLevel();
  if(level == GOLD) {
    this.discount = 20;
  }
}
```

(b) Step by step analysis

Instruction i	Q_i			Γ_i
	ρ_i	s_i	$\ddot{S}_i$	
`0: aload 1`	ρ_0	ϵ	$\emptyset$	$\emptyset$
`1: invoke getLevel`	$\backsim$	$\{(p_1^{p,s},\mathbf{r})\} :: \epsilon$	$\backsim$	$\backsim$
`2: istore 2`	$\backsim$	$\{(r_1^p,\mathbf{v})\} :: \epsilon$	$\ddot{S}_2$	$\backsim$
`3: iload 2`	ρ_3	ϵ	$\backsim$	$\backsim$
`4: ifeq 6`	$\backsim$	$\{(r_1^p,\mathbf{v})\} :: \epsilon$	$\backsim$	$\backsim$
`5: goto 9`	$\backsim$	ϵ	$\backsim$	$\{r_1^p\}$
`6: aload 0`	$\backsim$	ϵ	$\backsim$	$\{r_1^p\}$
`7: bipush 20`	$\backsim$	$\{(p_0^{p,s},\mathbf{r}),(r_1^p,\mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`8: putfield discount`	$\backsim$	$\{(Const^p,\mathbf{v}),(r_1^p,\mathbf{i})\} :: \{(p_0^{p,s},\mathbf{r}),(r_1^p,\mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`9: return`	$\backsim$	ϵ	$\ddot{S}_9$	$\emptyset$
exit	$\emptyset$	ϵ	$\ddot{S}_{exit}$	$\backsim$

$$
\begin{aligned}
\rho_0 &= \{(0 \mapsto (p_0^{p,s},\mathbf{r})),(1 \mapsto (p_1^{p,s},\mathbf{r})),(2 \mapsto \emptyset)\}\\
\rho_3 &= \{(0 \mapsto (p_0^{p,s},\mathbf{r})),(1 \mapsto (p_1^{p,s},\mathbf{r})),(2 \mapsto (r_1^p,\mathbf{v}))\}\\
S_{getLevel} &= \{R^p \xrightarrow{\mathbf{i}} p_0^{p,s}, R^p \xrightarrow{\mathbf{i}} Static^p\}\\
\ddot{S}_1 &= \{r_1^p \xrightarrow{\mathbf{i}} p_1^{p,s}, r_1^p \xrightarrow{\mathbf{i}} Static^p\}\\
\ddot{S}_9 &= \ddot{S}_2 \sqcup \{p_0^{p,s} \xrightarrow{\mathbf{v}} Const^p, p_0^{p,s} \xrightarrow{\mathbf{v}} r_1^p, p_0^{p,s} \xrightarrow{\mathbf{i}} p_1^{p,s}, p_0^{p,s} \xrightarrow{\mathbf{i}} Static^p\}\\
\ddot{S}_{exit} &= \ddot{S}_2 \sqcup \ddot{S}_2 = \{r_1^p \xrightarrow{\mathbf{i}} p_1^{p,s}, r_1^p \xrightarrow{\mathbf{i}} Static^p, p_0^{p,s} \xrightarrow{\mathbf{v}} Const^p, p_0^{p,s} \xrightarrow{\mathbf{v}} r_1^p,\\
&\quad p_0^{p,s} \xrightarrow{\mathbf{i}} p_1^{p,s}, p_0^{p,s} \xrightarrow{\mathbf{i}} Static^p\}\\
S_{discount} &= \{p_0^{p,s} \xrightarrow{\mathbf{i}} p_1^{p,s}, p_0^{p,s} \xrightarrow{\mathbf{i}} Static^p\}
\end{aligned}
$$

Figure 3.13: A detailled analysis example

An illegal information flow is disclosed when analysing method `FlyMaroc.discount`: the flow $p_0^{p,s} \xrightarrow{\mathbf{i}} p_1^{p,s}$ reflects the implicit flow from the secret part of parameter p_0 (MHz) to public part of parameter p_0 (`this`). Hence, FlyMaroc can infer information about loyalty points of FlyFrance.

To illustrate the operational semantics, we give a sketch of the analysis based on the LoyaltyCard example. In Figure 3.13, we detail the evolution of the states Q during the abstract interpretation on the bytecode in method `FlyMaroc.discount`. The abstract domain $\ddot{\Sigma}_m^L$ is

$$\ddot{\Sigma}_m^L = \{p_0^{p,s}, p_1^{p,s}, r_1^{p,s}, R^p, Static^p, Ex^p, IO^p, Const^p, \mathtt{null}^p\}$$

where r_1 is the abstract value corresponding to the return of the `invoke` instruction at bytecode `1`. This abstract value is local to method and does not survive method's execution; hence, the final abstract domain, Σ_m^L is

$$\Sigma_m^L = \{p_0^{p,s}, p_1^{p,s}, R^p, Static^p, Ex^p, IO^p\}.$$

The symbol $\backsim$ denotes repetition, *i.e.*, a set equal to the one above in the column. The abstract analysis starts with the initial state Q_0, which contains an empty stack and an empty flow signature, while the local variables are mapped to the methods parameters. Instruction `0` loads the local variable 1 on the stack, while instruction `1` invokes method `MHz.getLevel`; the abstract interpretation of `invoke` instruction consists in mapping the real arguments (stored on the stack) to formal parameters of the method and to translate flows between abstract values in the signature of invoked method ($S_{getLevel}$) to flows between mapped abstract values to the current flow signature ($\ddot{S}_1$).

It is worth noting that the instructions `5`, `6`, `7` and `8` belong to the control dependency region of `4`, thus their execution depends on the value tested by the conditional instruction (the top of the stack in Q_4). Hence, we can compute $\Gamma_5 = \Gamma_6 = \Gamma_7 = \Gamma_8 = \{r_1^p\}$. For example, besides the content of local variable 0, $(p_0^{p,s}, \mathbf{r})$, instruction `6` also pushes the element $(r_1^p, \mathbf{i})$ on the stack.

Let us denote by v the set $\{(Const^p, \mathbf{v}), (r_1^p, \mathbf{i})\}$ and by u the set $\{(p_0^{p,s}, \mathbf{r}), (r_1^p, \mathbf{i})\}$; we can rewrite the state Q_8 as $Q_8 = (\rho_3, v :: u :: \epsilon, \ddot{S}_2)$. The `putfield` instruction assigns v to a field of u. Thus, flows from v to u are created; moreover, as the assignment takes place in the control dependency region of instructions in $cxt(8)$, flows from abstract values in Γ_8 to u are generated too.

Concretely, the `putfield` instruction generates the following flows, according to abstract semantics rule in Figure 3.6:

- $p_0^{p,s} \xrightarrow{\mathbf{v}} Const^p$ and $p_0^{p,s} \xrightarrow{\mathbf{v}} r_1^p$
- $p_0^{p,s} \xrightarrow{\mathbf{v}} r_1^p$ (as $r_1^p \in \Gamma_8$)

As, before the execution of `putfield`, the signature $\ddot{S}_2$ contains the flows $r_1^p \xrightarrow{\mathbf{i}} p_1^{p,s}$ and $r_1^p \xrightarrow{\mathbf{i}} Static^p$, the function *closure* generates the flows $p_0^{p,s} \xrightarrow{\mathbf{i}} p_1^{p,s}, p_0^{p,s} \xrightarrow{\mathbf{i}} Static^p$. Informally, as p_0 *points to* r_1, and r_1 *points to* p_1 and $Static$, by transitivity, p_1 also points to p_1 and $Static$.

The flow signature for `discount` is S_{exit}, while the final flow signature, $S_{discount}$, is given by $\ddot{S}_{exit}$ restricted to elements in Σ_m^L, thus to elements that survive: $S_{discount} = \{p_0^{p,s} \xrightarrow{\mathbf{i}} p_1^{p,s}, p_0^{p,s} \xrightarrow{\mathbf{i}} Static^p\}$.

3.4 Information flow verifier

In the previous section, we presented a compositional information flow analysis complying with the Java dynamic class loading paradigm, which computes for each method a flow signature, containing all possible flows of information in that method. But experimental results show that the analysis is already expensive for a normal system and impracticable for a device with limited resources, both in time and memory. These results justify an approach based on “Lightweight bytecode verification” (LBV) [RR98] technique, which splits the analysis in two steps: an offboard phase, which computes flow signatures and proof elements, and an onboard phase, which verifies the flow signatures obtained in the offboard phase.

In this section we concentrate on the type verification and proof elements needed to perform it. The type and proof computation can be performed by any device or tool, as the small open system can verify the code it receives without relying on the external device. LBV relies on the lattice structure of flows and on unification operations on this lattice. The lattice of flows depicted in Figure 3.4 allows us to extend this technique and to use it in our context. While LBV checks explicit Java types, our algorithm has to infer information flows. We have to deal with type inference and with flow signature management.

Thanks to proof elements, the verification is linear in time and size, and it needs constant memory. The verifier is implemented as a custom class loader so it can be used as a plug-in on any JVM.

We first detail the flow signature verification, by describing the proof elements and the embedded verification algorithm. The proof elements are only used during the verification process and discarded afterwards, while flow signatures must be kept onboard, in a repository, for future verifications. Hence, as the correct and optimized use of the repository is a crucial element, we focus our attention on flow signature management. Finally, we show how the verification is integrated in a custom class loader, and how a heterogeneous class loader hierarchy is taken into account.

3.4.1 Flow-signature verification

Proof elements

The flow signatures computed by the prover are shipped within the code and must be verified onboard. The verification process is performed when loading a Java class.

In order to ease verification, we ship with each class C some proof elements:

- the abstract state for each jump target instruction in each method; this is needed in order to avoid the iteration (dictated by the absence of high level control flow constructs) on the set of bytecodes of a method m and to have a verification linear in code size,
- the flow signatures of methods invoked in C (pending signatures); due to the presence of inter-dependent methods and the fact that the JVM loads one class at a time, C may need flow signatures of methods not yet loaded; pending signatures are stored onboard and verified when corresponding code arrives,
- abstract values tested by conditional instructions, needed to compute the implicit flows,
- the security levels of fields used in C.

The proof elements are defined as new attributes of the `class` file structure, so the annotated classes can be loaded by any JVM, even by those not enforcing information flow security properties. After using it to verify the flow signature, the proof is discarded and only the flow signature is kept onboard. Almost constant memory is needed for the verification.

For example, for method `discount`, depicted in Figure 3.13, page 47, the proof elements are: the abstract states Q_6 and Q_9 of the jump target bytecode `6` and `9`; the pending signature of method `MHz.getLevel`, $S_{getLevel}$; the top of the stack in Q_4, corresponding to values tested by conditional instruction `4`.

Implicit flow verification

To infer implicit flows, we use the immediate postdominators [Bal93] in the control flow graph of a program. To verify these flows, the notion of immediate postdominators must be used for the embedded algorithm. There are two options: *(1)* either to compute postdominators offboard and load some proof elements to verify them onboard, or *(2)* to compute postdominators onboard. Many algorithms for verifying postdominators exist [GT05], but they do not improve considerably the complexity of algorithms computing postdominators [LT79], which perform in linear time. The main advantage of verifying instead of computing is that less memory is required. Still, the difference is not significant even for a small system. Thus, due to the low complexity of algorithms for computing postdominators, we chose to compute dominators onboard. The function computing dominators belongs to our trusted computing base.

For a method m and an instruction $i \in P_m$, $cxt(i)$ is computed locally. We remind that $cxt(i)$ denotes the set of conditional instructions on which the execution of i depends; Γ_i contains the set of abstract values tested by instructions in $cxt(i)$, thus the top of the stack for Q_i. To ease the implicit flow verification and the computation of Γ_i, we load, as proof elements, the top of the stack for all conditional instructions, thus instructions that can be found in $cxt(i)$. The soundness of this proof element is similar to the soundness of frames for label bytecodes.

Embedded abstract semantics

Let $Labels \subseteq P_m$ be the set of jump target bytecodes, $Labels = \{i \in P_m \mid \exists j \in P_m \text{ s.t. } i \in succ(j) \wedge i \neq j+1\}$. A jump target bytecode is a bytecode that can be reached through a branching

Algorithm 3.3 Embedded verification of a method m

```
compute $cxt(i), \forall i \in P_m$
initialize $Q_v$
for $i = 0$ to $|P_m|$ do
    apply the embedded semantics of $P_m[i]$
    for all $i'$ in $succ(i)$ do
        if $i' \in Labels$ then
            assert $Q_v \sqsubseteq Proof(i')$
        end if
        if $i' \in Cond$ then
            let $Q_v = (\rho, n :: s, S_A)$
            assert $n \sqsubseteq Proof_{ipd}(i')$
        end if
    end for
    if $i \in Labels$ then
        $Q_v = Proof(i)$
    end if
end for
```

bytecode, such as `ifeq` a or `goto` a. In these examples, a represents the targeted bytecode, and so $a \in Labels$.

With each target bytecode $i \in Labels$, we ship the abstract state Q_i computed by the external analysis, as presented in the previous section. Let *Proof* be a function that associates to each target bytecode its state Q_i.

Let $Cond \subseteq P_m$ be the set of conditional bytecodes, $Cond = \{i \in P_m \mid P_m[i] = \texttt{ifeq}\ \mathrm{a}\}$. For $i \in Cond$, let $Q_i = (\rho, n :: s, \ddot{S}_i)$ be the frame computed by the external analysis. Then, the value n tested by i (*e.g.*, the top of the stack), is shipped with the code. Let $Proof_{ipd}$ be a function that associates n to each conditional bytecode.

The verification process consists in a sequential interpretation of bytecodes of each method of the loaded class, starting with the first instruction ($P_m[0]$), as presented by Algorithm 3.3. The semantics (a small subset is presented in Figure 3.14) is similar to the semantics for external computation, the main difference consisting in the fact that there is only one frame, Q_v, needed to be stored in memory and not the entire function Q; each rule of an instruction i modifies the state Q_v, and not the states associated to successors of i.

When a target bytecode a is found, the current state of the JVM, Q_v must be compatible with the proof corresponding to the target bytecode, given by $Proof(a)$: if the compatibility is not tested, the class is rejected; otherwise the verification is carried on using as current state Q_v either by the result of the precedent instruction, if it is not a target bytecode, or by the proof loaded with the bytecode otherwise. The compatibility relation corresponds to *smaller than* relation: a state Q' is compatible with Q'' if $Q' \sqsubseteq Q''$. The compatibility of two flow signatures is defined similarly.

By using this mechanism, a small system can certify a method using the proof computed by a third party without relying on it. The mechanism is based on two main ideas:

1. the proof contains all possible conditions under which a bytecode can be executed (by construction);

2. the code agrees with the proof as, for the target bytecodes, we verify the compatibility of the state Q_v under which they can be executed, with the proof.

$$\frac{P_m[i] = \texttt{pop} \qquad Q_v = (\rho, v :: s, \ddot{S})}{Q_v = (\rho, s, \ddot{S})} \qquad \frac{P_m[i] = \texttt{ifeq}\ a \qquad Q_i = (\rho, v :: s, \ddot{S})}{Q_v = (\rho, s, \ddot{S})}$$

$$\frac{P_m[i] = \texttt{getfield}\ f_{C'} \qquad Q_v = (\rho, v :: s, \ddot{S})}{Q_v = (\rho, (v_{|f_{C'}}, \mathcal{T}(f_{C'})) \cup v_{|\mathbf{v}} \cup v_{|\mathtt{i}} \cup \mathit{TV}_{\Gamma_i} :: s, \ddot{S})}$$

$$\frac{P_m[i] = \texttt{putfield}\ f_{C'} \qquad Q_v = (\rho, v :: u :: s, \ddot{S}) \qquad \phi' = \phi \sqcap \mathcal{T}_{\mathcal{F}}(f_{C'})}{Q_v = (\rho, s, \mathit{closure}(\ddot{S} \sqcup \{u_{|f_{C'}} \xrightarrow{\phi'} e \mid (e, \phi) \in v\} \sqcup \{u_{|f_{C'}} \xrightarrow{\mathtt{i}} V^u_{\mathtt{i}} \cup \Gamma_i\}))}$$

$$\frac{P_m[i] = \texttt{invoke}\ m'_{C'} \qquad Q_v = (\rho, v_{n_m} :: .. :: v_1 :: u :: s, \ddot{S}) \qquad S_{m'} = \mathit{lookup}(C'.m')}{Q_v = (\rho, \{(r^p_i, \mathbf{r}), (r^s_i, \mathbf{r})\} :: s, \mathit{closure}(\ddot{S} \sqcup \mathit{apply}(\ddot{S}, S_{m'}, u, v_1, .., v_{n_m}))}$$

with $\phi, \phi' \in \mathcal{F}$ $\quad V^u_\phi = \{e \mid (e, \phi) \in u \wedge e \neq \texttt{null}^p\}$ $\quad u_{|\phi} = \{(e, \phi) \mid (e, \phi) \in u\}$ $\quad \mathit{TV}_{\Gamma_i} = \{(e, \mathtt{i}) \mid e \in \Gamma_i\}$
$u_{|f_{C'}} = \{\mathit{aprox}(e, f_{C'}) \mid e \in V^u_{\mathbf{r}}\}$ $\quad \mathcal{T}_{\mathcal{F}}(f_{C'})$ *gives the type of field* $f_{C'}$ *(*$\mathbf{v}$ *or* $\mathbf{r}$*) with respect to* $\mathcal{F}$.

Figure 3.14: A subset of embedded information flow verifications rules.

Discussion

The embedded verification has the advantage that each instruction is interpreted only once and so it is linear in time with the code size. Moreover, the proof is used only during the verification and not stored in the system. Only the final flow signature of each method is kept onboard. Another advantage is that each class is verified only once, even the code shared by many software units, as the signatures are kept onboard in a repository. If the type inference of flow signature fails, the class is rejected. If the type inference succeeds, we must ensure that the flow signatures used during validation fit within the system.

Due to limited resources of small open systems, the size of proof elements must be as small as possible. As the lattice of flows contains 256 possible elements, we chose a binary and compact solution on 1 byte to encode the flows. This solution allows simple manipulation operations. For example, adding a new flow to a flow signature requires only a bitwise logical operation. Formally, $\theta \cup \theta' = \theta''$ iff $\mathit{encode}(\theta) \& \mathit{encode}(\theta') = \mathit{encode}(\theta'')$ where encode gives a binary representation of a flow in Θ (the lattice of possible flows between two abstract values, defined in Section 3.2.3) which allows simplified bitwise operations.

Moreover, the flow signatures within the states of the JVM for each target instruction are encoded incrementally: the first signature is encoded, while the subsequent signature is defined by the difference from the previous signature (the flows added or removed since the last label). Experimental results showed that signatures have few changes from one label to another.

Dead code is ignored by the external analysis and thus not annotated. In order to deal with this situation, when a label bytecode without any proof annotation is found, we can assume it is the beginning of a block which is never executed. In this case, all the bytecodes following the label are ignored, until we meet an annotated label. If the label without a proof is not the start of a dead block, then the class is rejected when the compatibility of predecessors instructions with the proof of the label is tested. A simpler solution is simply to eliminate dead code, performing code optimization at the same time.

The analysis guarantees non-interference for loaded classes. All the possible flows from secret to public data are detected and stored in flow signatures. But, due to our simplifying assumptions, we might detect false flows. Nevertheless, our experiments (described and discussed in Section 4.1) have shown that this is not an issue in practice.

3.4.2 Flow signature management

The flow signature of a method depends on flow signatures of invoked methods. Thus flow signatures must be kept onboard, in a repository, denoted by $\mathcal{R}$, for future use. The repository must deal with two types of flow signatures:

- certified signatures, stored in a *certified repository*, $\mathcal{R}^c$,
- pending signatures, stored in a *pending repository*, $\mathcal{R}^p$.

Certified signatures refer to *(1)* flow signatures already verified, *(2)* hand-written signatures for native methods, which belong to our trusting base and *(3)* signatures not available, which are set to the top element according to the lattice of flow signatures.

Algorithm 3.4 init(C)

```
for all methods $C'.m'$ invoked in $C$ do
    Let $S_{m'}$ be the announced flow signature
    if $C'.m'$ already loaded then
        discard $S_{m'}$
    else
        if $\exists \mathcal{R}^p(C'.m')$ then
            Let $S'_{m'} = \mathcal{R}^p(C'.m')$
            $\mathcal{R}^p = \mathcal{R}^p[C'.m. \mapsto S_{m'} \sqcap S'_{m'}]$
        else
            $\mathcal{R}^p = \mathcal{R}^p[C'.m. \mapsto S_{m'}]$
        end if
    end if
end for
```

Pending signatures correspond to methods not yet available when analyzing a method. In this case, a signature can be announced, stored in the pending repository $\mathcal{R}^p$ and verified when code is finally deployed in the virtual machine.

Algorithm 3.4 presents how pending signatures announced by a class C are handled, while loading C. If the pending signature corresponds to a method already loaded, then it is ignored and discarded. Otherwise, the pending signature is stored in the pending repository, $\mathcal{R}^p$. There is possible to have different pending signatures for the same method. The naive solution would be to keep all pending signatures in $\mathcal{R}^p$. On the other hand, due to context of small systems and limited resources, the repository must be as small as possible. Thus, a way to reduce repository is to store only a pending signature for a given method; if different signatures are announced, then we store in the repository the infimum according to the lattice of flow signatures. The logic behind this idea is straightforward: given two pending signatures, S'_m and S''_m, and the certified signature S_m, S_m is accepted if it is compatible with both S'_m and S''_m ($S_m \sqsubseteq S'_m$ and $S_m \sqsubseteq S''_m$), hence if flows in S_m are in both S'_m and S''_m ($S_m \sqsubseteq S'_m \sqcap S''_m$).

The signature lookup algorithm (Algorithm 3.5) for a method $C.m$ first searches in the certified repository. If no signature is found, then it interrogates the pending repository. If the search fails again, then the flow signature for $C.m$ is set to the maximum and it is stored in $\mathcal{R}^p$.

The validation of pending signatures is performed when code arrives, as depicted by the Algorithm 3.6. The algorithm is performed after the verification of a class C, and consists of verifying, for each method m in C, that the certified signature is compatible with pending signature (certified signature must not contain more flows than pending signature). If the compatibility is not verified,

Algorithm 3.5 lookup($C.m$)

```
if $\exists\mathcal{R}^c(C.m)$ then
    return $\mathcal{R}^c(C.m)$
end if
if $\exists\mathcal{R}^p(C.m)$ then
    return $\mathcal{R}^p(C.m)$
end if
Let $S_m = \top$
$\mathcal{R}^p = \mathcal{R}^p[C.m \mapsto S_m]$
return $S_m$
```

Algorithm 3.6 end(C)

```
for all $m \in Method(C)$ do
    Let $S_m$ be the certified flow signature
    if $\exists\mathcal{R}^p(C.m)$ then
        Let $S'_m = \mathcal{R}^p(C.m)$
        assert $S_m \sqsubseteq S'_m$
        remove $S'_m$ from $\mathcal{R}^p$
    end if
    $\mathcal{R}^c = \mathcal{R}^c[C.m \mapsto S_m]$
end for
```

then the class C is rejected and not loaded. Otherwise, the pending signatures are erased from $\mathcal{R}^p$ and verified signatures added to $\mathcal{R}^c$.

The correctness of a flow signature also depends on the security levels of used fields. To have access to security levels of fields of an unavailable class, we proceed similarly: we ship the security levels of all fields and we keep them onboard on the pending repository (let us denoted it by $\mathcal{R}^p_f$). When the classes containing the field definition are loaded, we check the compatibility of fields and, if they are compatible, we erase them from the pending repository. Two fields are compatible if they have the same security level.

3.4.3 The verifier as a user-defined class loader

The loading process in a JVM is performed by the class loaders. Programs implement subclasses of ClassLoader in order to extend the manner in which the JVM dynamically loads classes. Class loaders may typically be used to check security properties. The standard JVM deals with multiple class loaders, hierarchically organized, and supports user-defined class loaders. The KVM virtual machine [Sun] does not support user-defined class loaders and has a single built-in class loader that cannot be overridden or replaced by the user.

We build a verifier that can be run on any JVM. It can be built in the single class loader of KVM or installed as a user-defined class loader for a standard JVM. The embedded JVM [Aon, Jav, Sun] are evolving towards the standard Java language, and therefore towards a multiple class loader hierarchy. The recently presented JAVACARD 3.0 does the same. We now describe how the verifier can be used as a plug-in within a standard JVM to validate annotated bytecode.

The verifier was implemented as a subclass of the *ClassLoader* class provided by the Java API, named *SafeClassLoader*. Certifying the underlying information flow of a software unit requires the instantiation of a *SafeClassLoader* with which the software unit should be loaded.

Extending the delegation model to flow signatures lookup

In the JVM delegation model, class loaders are arranged hierarchically in a tree, with the bootstrap class loader as the root of the tree. Each user-defined class loader has a "parent" class loader. When a load request is made by a user-defined class loader, that class loader usually first delegates the parent class loader, and only attempts to load the class itself if the delegate fails to do so. A loaded class in a JVM is identified by its fully qualified name and its defining class loader. This is sometimes referred to as the runtime identity of the class. Consequently, each class loader in the JVM can be said to define its own name space. In the same manner, each $SafeClassLoader$ defines its own repository containing the signatures of loaded methods. For a class loader Scl, let $Scl.\mathcal{R}^c$ denote its repository of certified flow signatures.

Algorithm 3.7 extended lookup($Scl.C.m$)

```
while $Scl \neq$ null do
    if $\exists Scl.\mathcal{R}^c(C.m)$ then
        return $Scl.\mathcal{R}^c(C.m)$
    end if
    $Scl = parent(Scl)$
end while
if $\exists \mathcal{R}^p(C.m)$ then
    return $\mathcal{R}^p(C.m)$
end if
Let $S_m = \top$
$\mathcal{R}^p = \mathcal{R}^p[C.m. \mapsto S_m]$
return $S_m$
```

In order to support multiple class loaders, we extend the lookup algorithm to take into consideration the delegation model (as depicted by Algorithm 3.7). When a class loader Scl needs the flow signature of a method $C.m$, it first searches in its own certified repository ($Scl.\mathcal{R}^c$), and if the search fails, it delegates the search to its parent, which repeats the procedure. If the parent also fails to find the signature, the lookup continues in the pending repository. Pending signatures are verified when code arrives. When Scl loads $C.m$, it must verify pending signatures for $C.m$ that were used in Scl, as well as in the classloaders that may delegate Scl (class loaders that have Scl as parent or parent of parent, etc.). Thus, Scl must have access to pending signatures in its child classloaders. In the same time, the JVM does not offer a mechanism which allows classloaders to access or have any knowledge about their child class loaders. To overcome this limitation, we implement the pending repository as a unique repository used by all class loaders. In order to allow classloader and hence class removal, the unique repository uses weakreferences. When a reference is broken by the garbage collector (*i.e.*, when the classloader and associated classes are removed from the JVM), the entry is removed from the pending repository. Otherwise the loaded class cannot be unloaded by the runtime environment.

We showed how the $SafeClassLoader$ extends the delegate model to the look up of a signature of a method. The same search process is extended to the look up of the security level of a field.

Example

To better understand the lookup and delegation models, let us consider the class loader hierarchy in Figure 3.15 containing three $SafeClassLoader$s: Scl_2, Scl_3 and their parent Scl_1, and the following loading scenario:

Scl_2 loads class C

pending signature S^2_{foo} for Scl_2.A.foo

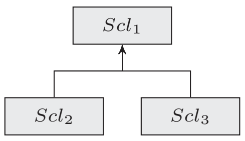


Figure 3.15: *SafeClassLoader* hierarchy

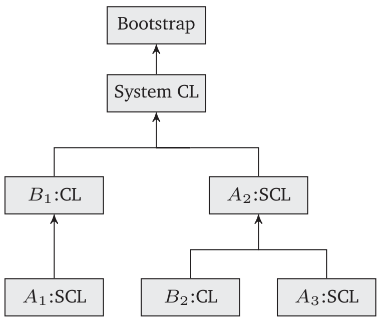


Figure 3.16: Class Loader hierarchy example

Scl_1 loads class D
    pending signature S^1_{foo} for Scl_1.A.foo
Scl_1 loads class A
    certified signatures S_{foo} for Scl_1.A.foo

Class loader Scl_2 requests to load class C. At first, it delegates its parent class loader, Scl_1, to load C. If the delegation fails, Scl_2 attempts to load the class by itself. While loading and verifying C, Scl_2 needs the flow signature of $A.foo$: it first searches in its own certified repository, and if the search fails, it delegates the search to its parent, which repeats the procedure. If the parent also fails, pending signature S^2_{foo} is used while validating C and stored in the pending repository. Class loader Scl_1 loads a class D announcing a different pending signature for $A.foo$. The signature S^1_{foo} is stored in the pending repository and is associated with Scl_1.

Finally, class loader Scl_1 attempts to load class A. Let S_{foo} be the certified signature of foo. Class A will be loaded by Scl_1 if and only if S_{foo} is compatible with the pending signatures for foo in the current class loader (Scl_1) and with pending signatures in class loaders that can delegate Scl_1. In our case, S_{foo} must be compatible with S^1_{foo} and S^2_{foo}. Otherwise, class A is rejected.

Mixed hierarchy of class loaders

We presented so far the case where all the class loaders in the hierarchy have the type *SafeClassLoader*. Actually, the hierarchy contains different types of class loaders. As shown in Fig. 3.16, the bootstrap class loader loads the classes from the JVM, as well as extensions to the JDK. The system class loader loads all the classes provided by the classpath. In the end, we have several additional class loaders, where SCL stands for *SafeClassLoader* and CL for any other type of class loader.

As a consequence, we must take into consideration the validation of classes loaded by any class loader. Let's consider that A_1 loads a class C that invokes a method of another class D already loaded by the parent B_1. As B_1 is not a *SafeClassLoader*, the classes it has loaded have not been validated at loading time. To ensure security for C, *SafeClassLoader* A_1 will try to retrieve, using the *getResourceAsStream* method, the .class files of the classes loaded by its parent and to verify the announced signatures. If the streams cannot be found, or if they do not contain information flow attributes, or if the signatures are not compatible with the announced ones, A_1 rejects class C. Otherwise, the signatures of classes belonging to a non-*SafeClassLoader* are stored in a special dictionary, named "system dictionary". The look up for a signature in a class loader is performed in its dictionary, if the class loader is a *SafeClassLoader*, and otherwise in the system dictionary.

In order to support any JVM, we do not interfere while the Bootstrap and System class loaders load the JVM and classpath classes, and thus we consider their signatures as part of our trust computing base.

3.5 Support for open class hierarchy

In the previous sections we showed how flow signatures are computed and verified. In this section, we add support for polymorphism and open world.

3.5.1 Overriding methods

Global flow signature

In an open world, where software units are loaded dynamically, the entire call graph is not available. The compositional analysis overcomes this limitation, as a context-insensitive (it does not depend on the call site) flow signature is computed for each method. The flow signature of a method depends on flow signatures of invoked methods. As the exact type of the called object (let us denote it by C) cannot be statically determined and, moreover, it can be overridden by future programs, inheritance and overriding can be a way to bypass the security controls. Loading a method overriding an existing method can be a threat for code certified using the old method.

To avoid such security leaks, we define *contracts* for methods overriding already loaded methods, called global flow signatures, which give the expected behaviour of new methods. Global flow signatures are a support to extensibility and openness of small systems [AGGSR07], as they describe the required behaviour of pieces of code not yet loaded in terms of information flow, more exactly the maximum flows that can potentially be generated by the execution of a method.

In order to be accepted in the system, newly loaded methods must respect the contracts imposed by the classes in its hierarchy.

Hence, we compute, for each method m in a class C, two flow signatures: *(i)* the *exact* flow signature, S_m, and *(ii)* a *global* flow signature, S^g_m. The exact flow signature is the signature computed by the intra-procedural analysis and it represents the flow of information generated if the method m in class C is executed. The global flow signature is computed, at a certain time, from the class hierarchy derived from C and stands true if any reimplementation of m in subclasses of C is executed.

The global signatures are computed by the prover, during the external phase, and they are used to validate methods in the class hierarchy in the embedded verification process.

During the compositional analysis, when an `invoke` instruction is encountered, if the exact type of the called object is known, the exact flow signature is used, otherwise, the global flow signature is used.

Exact type

In a static analysis, the run-time type of an object is not necessarily the declared type. The exact type of an object [BSF04] represents the run-time type of the object, on which `invoke` calls will be made.

For example, in the code below

```
static void foo(A a) {
    a.m();
    B b = new B();
    C c = new B();
    b.m1();
}
```

the exact type of `a` (and of parameters, in general) cannot be statically determined. But the exact type of `b` and `c` can be statically computed. In other words, the exact type of an object is known if we can statically decide which allocation site (and so which class) has been used to create the instance.

Knowing the exact type statically makes the analysis more precise, as the exact flow signature can be applied, without even taking into consideration the inheritance issues. If the exact type cannot be statically computed, then the global flow signature is used, which is an approximation of the entire set of methods that can be invoked.

Therefore, besides computing flow signatures, we also make a type analysis, for determining the exact types of objects. The function $exactType(u)$, where u is a set of abstract values, returns true is all abstract values have an exact type, of false otherwise.

The semantics rule of the `invoke` instruction which accounts for exact types is:

$$\frac{P_m[i] = \texttt{invoke}\ \ m'_{C'} \qquad Q_i = (\rho, v_{n_m} :: .. :: v_1 :: u :: s, \ddot{S}) \qquad S_{m'} = \mathcal{R}(u, m')}{Q_{i+1} = Q_{i+1} \sqcup (\rho, \{(r_i^{p,s}, \mathbf{r})\} :: s, closure(\ddot{S} \sqcup apply(\ddot{S}, S_{m'}, u, v_1, .., v_{n_m}))}$$
$$\text{with } \mathcal{R}(u, m') = exactType(u) ? \mathcal{R}^{exact}(m') : \mathcal{R}^{global}(m')$$

where $\mathcal{R}(u, m')$ returns the exact flow signature (from the repository of exact flow signatures $\mathcal{R}^{exact}$) or global flow signature (from $\mathcal{R}^{global}$) depending on the exact type of the object on stack u. For instructions `invokespecial`, which invokes an instance method, and `invokestatic`, which invokes a static method, the exact type is always known, hence they are a simplification of our `invoke` instruction. The `invokeinterface` method is similar to virtual invocation.

Computing global flow signatures

A completely open system, where all global flow signatures are set to top, would yield imprecise results, thus, when possible, we consider groups of classes and compute their global flow signatures; later, the global flow signature represent open doors for loading new classes.

The global flow signature of a method can:

- either be computed from the signatures of methods in the class hierarchy, when a set of classes is available,
- or defined manually in order to relax or define new contracts and allow more flexibility; in this case, the danger is to define too pessimistic flow signatures. Nevertheless, they will be accepted on the system only if they fit correctly in their class hierarchy.

In the first case, the global flow signature of $C.m$ is the union of the exact signature of $C.m$ and of all global flow signatures of the method m implemented in the classes derived from the static type of the object (C), formally

Global flow signature definition Given a class C, a method m in C and a set of classes Cl, the global flow signature of m_C, $S^g_{m_C}$ is defined as

$$S^g_{m_C} = \bigsqcup_{\forall C' \in Cl, C \leq C'} S_{m_{C'}}$$

The global flow signature is the least upper bound element according to the lattice of flow signatures for method m_C. As a consequence of the construction of the global flow signature, we have the following property:

Compatibility with global flow signature Given two classes C, C' such that C' extends C ($C \leq C'$), a method m in C and C', then:

$$S_{m_{C'}} \sqsubseteq S^g_{m_C}.$$

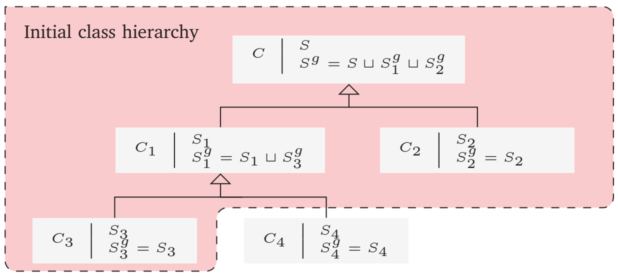

Initial class hierarchy
C | S, $S^g = S \sqcup S_1^g \sqcup S_2^g$
C_1 | S_1, $S_1^g = S_1 \sqcup S_3^g$
C_2 | S_2, $S_2^g = S_2$
C_3 | S_3, $S_3^g = S_3$
C_4 | S_4, $S_4^g = S_4$

Figure 3.17: Global signatures for class hierarchy

In the external phase, when a set of classes is being analyzed, global signatures can be relaxed: if a less restrictive flow signature than the global signature is found, the global signature can be adapted (relaxed) to include it. In the embedded phase, when loading new methods, their signatures must be compatible with the global signatures, as we will explain in the next subsection. Again, we rely on the idea that it is easier to verify the flow signatures compatibility (the $\sqsubseteq$ relation) than to compute the least upper bound flow signature (the $\sqcup$ relation).

This approach eases the use of our framework. The analysis allows the programmer to define a coherent set of classes, to use any of the features of the Java language like interfaces, abstract classes, without having to worry about signatures compatibility. Overriding does not necessarily require to respect the signature of the superclass. The global signature is used to help the programmer; if the global signature obtained is not satisfying, a manually generated and more pessimistic signature can be defined and verified. In this way, unknown classes can be dynamically loaded.

Example

Let us consider the class hierarchy in Figure 3.17; the class C contains a method m and all classes in the hierarchy override this method. For each class C_x, the exact signature of method m is S_x, and the global signature is S_x^g. The global flow signatures are computed from the initial class hierarchy, containing C, C_1, C_2 and C_3.

As class C_3 is at the bottom of the hierarchy and does not have any subclasses, the global signatures of method m is equal to the exact flow signature: $S_3^g = S_3$. Instead, the global signature of method $C_1.m$ and $C.m$ must stand true for any possible virtual invocation. Hence, the global flow signature of $C_1.m$ is the union of S_1 and the global flow signatures in classes extending C_1:

$$S_1^g = S_1 \sqcup S_3^g.$$

The same affirmation is true for C:

$$S^g = S \sqcup S_1^g \sqcup S_2^g.$$

Global flow signature embedded verification

Global flow signatures are computed offboard, and shipped with the code onboard. When a new class is loaded, the flow signatures of its methods must be compatible with the global flow signatures of the class hierarchy to which they belong. A method and its signature are compatible with the hierarchy if and only if it is compatible with (smaller than) the global signature of the classes above in the hierarchy.

For example, let us consider that the class C_4, extending C_1, is added to the initial class hierarchy in Figure 3.17. When loading class C_4 in Figure 3.17, the following properties must hold: $S_4 \sqsubseteq S_1^g$.

From the same example, we can observe that there is an inclusion relation between global signatures: $S_3^g \sqsubseteq S_1^g \sqsubseteq S^g$. Hence, it is sufficient to verify the compatibility of S_4 with the class from which it directly inherits (the class immediately above in the hierarchy): $S_4 \sqsubseteq S_1^g$. By transitivity and due to inclusion relation, further verifications are not required. We can now define the acceptance rule for a method $m_{C'}$ while loading class C': Formally,

Flow signature acceptance constraint Given a class C', its direct superclass C and a method m, the method $m_{C'}$ is loaded if and only if

$$S_{m_{C'}} \sqsubseteq S^g_{m_C}.$$

This constraint ensures that the global signature compatibility property is always true. In the JVM loading process, the superclass is always loaded before the child class, hence we have the insurance that while loading C', the superclass C and its flow signatures are already loaded.

3.5.2 Inheriting fields

Besides inheriting methods, a class also inherits fields from all its superclasses, whether direct or indirect. Extending a class and changing security policy of inherited fields represents a possible way of leaking confidential data, as computed signatures of inherited methods change, and the superclass must be reanalyzed. This is not convenient for our compositional approach and for an open system. To prevent such security leaks, we impose a restriction regarding inherited fields:

Inherited fields restriction The security level of inherited fields cannot be changed by a child class.

Nevertheless, the child class can declare new fields even with security level s.

3.5.3 Interfaces and abstract classes

Because object-oriented development efforts involve the interactions of objects, it is essential to develop and enforce strong contracts between those objects. Interfaces and abstract classes represent a way of establishing the interactions among the necessary objects, without forcing the early definition of the supporting objects.

While an interface is a contract that classes must respect, an abstract class defines the core identity of its descendants. Both abstract classes and interface define contracts through abstract methods, which do not have any implementation and which are not necessary overridden when deployed.

The global flow signature allows the perfect integration of abstract methods in our system. Considering an abstract class or an interface C and an abstract method m, then there is only a global flow signature associated to $C.m$. Exact flow signatures will be computed in classes implementing m as described in Section 3.5.1. Invoking an interface method $Cm.$, using `invokeinterface` instruction, consists of using the

3.5.4 Loyalty card post issuance example

In Section 3.1 we present an open system, LoyaltyCard, which consists of different loyalty software units which implement loyalty services.

To perform credit/debit operations with loyalty points, the card must communicate with untrusted hosts; to secure the communication, sent data is encrypted. JAVACARD offers different encryption algorithms, as depicted in the class hierarchy in Figure 3.18. It is an extensible hierarchy, as other encryption methods can be loaded later.

The encryption method must be of type `Cipher`, which offers the `doFinal` method as an interface for encryption and decryption. All cryptographic algorithms must extend the abstract

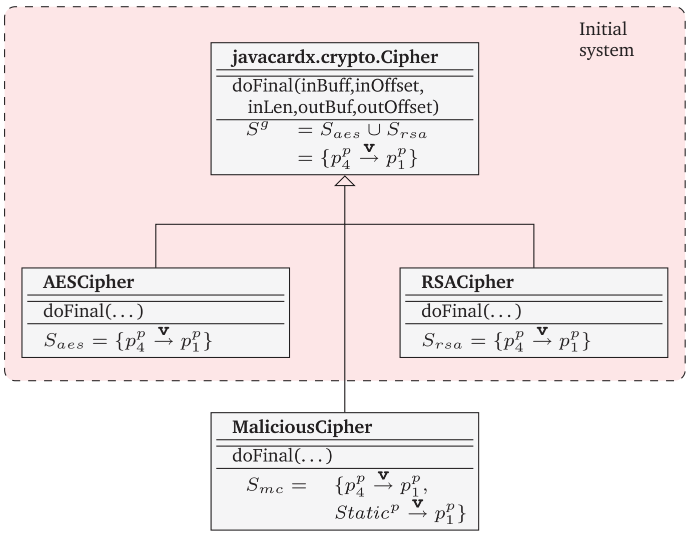

Initial system
javacardx.crypto.Cipher
doFinal(inBuff,inOffset, inLen,outBuf,outOffset)
$S^g = S_{aes} \cup S_{rsa} = \{p_4^p \xrightarrow{\mathbf{v}} p_1^p\}$
AESCipher
doFinal(...)
$S_{aes} = \{p_4^p \xrightarrow{\mathbf{v}} p_1^p\}$
RSACipher
doFinal(...)
$S_{rsa} = \{p_4^p \xrightarrow{\mathbf{v}} p_1^p\}$
MaliciousCipher
doFinal(...)
$S_{mc} = \{p_4^p \xrightarrow{\mathbf{v}} p_1^p, Static^p \xrightarrow{\mathbf{v}} p_1^p\}$

Figure 3.18: Javacardx.crypto.Cipher hierarchy

class `Cipher` and must overwrite the `doFinal` method. In our example, two algorithms are implemented: `AESCipher`, implementing the encryption based on single key shared by both parties, and `RSACipher` based on public/private key encryption.

The information flow analysis must hold for any encryption method in the `Cipher` hierarchy, so we compute a global signature for `doFinal` method in the `Cipher` abstract class, containing all possible flows of information in classes extending it. In the initial state, when only the `RSACipher` and `AESCipher` implementations are available, the global signature for method `doFinal` is the union of the implementation of this method in `AESCipher` and `RSACipher`. This signature indicates that there is a flow from the input buffer to the output buffer.

A possible danger may come from a `Cipher` instance shared by multiple software units. In this case it is possible that one software unit loads a class (*e.g.*, `MaliciousCipher`) that overwrites the method `doFinal` in a malicious way, for example, a method that assigns the input buffer to a static variable accessible to a third party.

```
class MaliciousCipher implements Cipher{
  public static byte[] s;
  short doFinal(byte[] inBuff, short inOffset,
                 short inLength, byte[] outBuff,
                 short outOffset){
     s = inBuff;
    ..}
..}
```

Our tool detects and prevents this kind of security threats, since when a class is added to a hierarchy, overwriting existing methods, these methods are accepted if and only if their signatures are compatible with the global signature of the class directly derived. Suppose that `MaliciousCipher` implements a symmetric encryption method. The signature S_{mc} for method `encrypt` in `MaliciousCipher` will contain, besides the flow between the output buffer (p_4) and the input buffer (p_1), a flow between the static value *Static* and the input buffer:

$$S_{mc} = \{p_4^p \xrightarrow{\mathbf{v}} p_1^p, Static \xrightarrow{\mathbf{v}} p_1^p\}$$

The class `MaliciousCipher` is rejected at loading time, as the signature S_{mc} is not compatible with the global signature S_g of the interface `Cipher`: in S_{mc} contains a flow not specified in `Cipher`. Even if the LoyaltyCard runs in an open environment, confidentiality is still enforced.

3.6 Collaboration policies for open multiapplication devices

One of the main challenge in current information flow research area is to define realistic security policies that faithfully describe the desired information flow within a system. Most of the current information flow models prevent all flows from secret data to public data by enforcing the non-interference property. Unfortunately, non-interference is often inadequate and is too strict to express desired security policies. There are a number of approaches that try to escape the limits of non-interference through downgrading (or declassification), but they failed to impose themself in practice as they are too restrictive or difficult to enforce. Moreover, usually these models mix source code and security policies in a coherent set, and do not address the problems raised by open environments; hence, redefining new policies requires reanalyzing the entire system. Integrating information flow policies for mobile code in open embedded systems requires, at least,

1. separation of code and security policies,
2. policy certification at loading time.

In the literature, information flow security policies are often limited to non-interference [GM82], as public outputs cannot depend on secret inputs. Non-interference policies do not allow any flows from secret to public values, but only flows from secret to secret. Nevertheless, non-interference does not make any distinction between the source of secrets. Policies defined with such a relation are too restrictive, and not the desired policies in most of the cases, and especially in open multiapplication systems [Gir99]. Our aim is to refine non-interference by defining more complex policies. In order to escape from non-interference strictness, we define a domain specific language, which describes the allowed flow of information between applications. The policies defined with our language are intransitive (*i.e.*, given three data sources a, b and c, if flows from a to b and from b to c are allowed, then this does not imply that flows from a to c are also allowed) and asymmetric (*i.e.*, if flows from a to b are allowed, then this does not imply that flows from b to a are allowed). Programs are certified by verifying that the flow signatures respect the desired security policies.

Our approach In the previous sections, we present an information flow analysis for open embedded systems. The security policy enforced by the analysis is the non-interference. In this section, we increase the practical side of our model by enriching it with security policies which relax non-interference.

Our aim is *(1)* to be able to express realistic security policies for multiapplication open systems and *(2)* to allow *a posteriori* definition of security policies.

In order to achieve our goal, we completely separate the definition and verification of security policies from the information flow type computation. The policies are defined in an external file, using a domain specific language. Verifying the security policy relies only on the knowledge of type certificate (the flow signatures) computed by our analyser (the prover). Security policies definition and verification extend our initial lightweight information flow certification model (Figure 3.19a) and they are integrated as independent components which can be activate or not by the user, as depicted in Figure 3.19b. Considering the dynamic loading of open systems, the security policies verification *must* be performed on card. Hence the embedded verifier must certify, besides flow signatures, the security policies. Nevertheless, in order to ease the development of safe software units, we provide a tool which certifies policies off card.

The certification of information flow policies must ensure the **incrementality** and **consistency** properties of the system, as defined in previous sections. Loading a software unit and its policy guarantees both the security of the new software unit and of the open system.

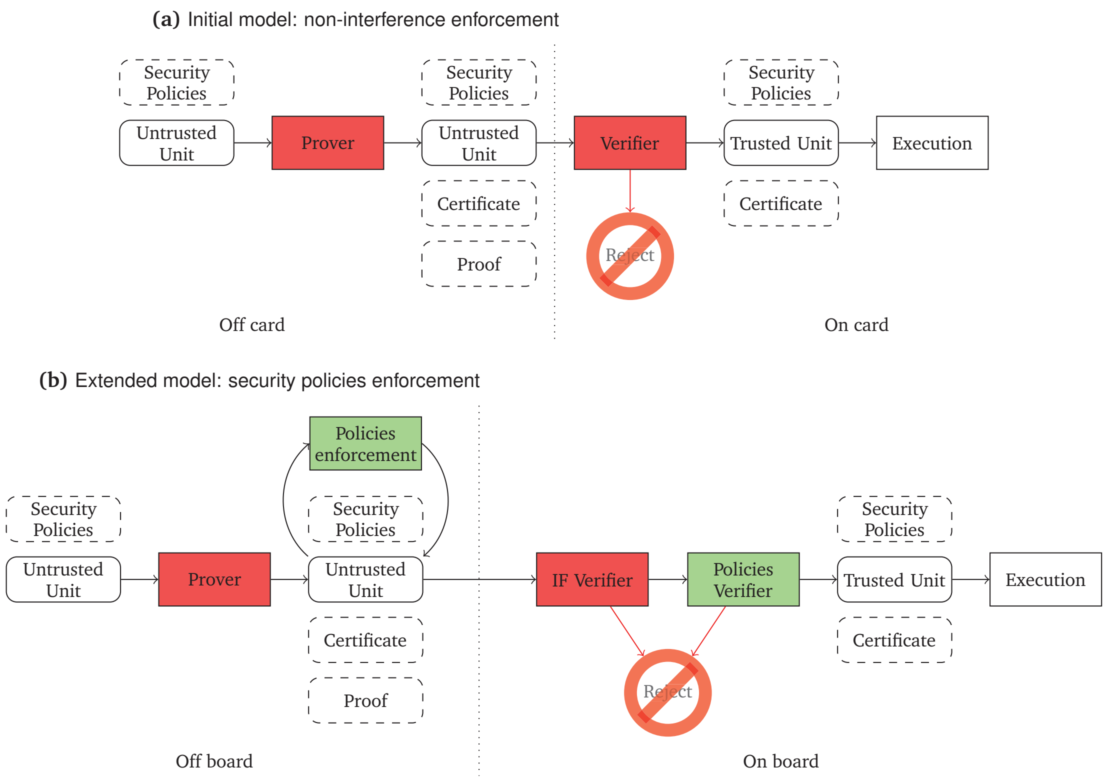


Figure 3.19: Lightweight information flow certification

Which security policy? Non-interference allows only flows of information from secret sources to secret destinations, without making any distinction between different sources and destinations. Usually, this distinction is achieved by using different security levels. In our model, we use two security levels, s for secret, confidential data and p public, observable data. As discussed in Section 3.2.1 at page 28, increasing the number of security levels will lead to a higher complexity, inadequate for small open systems. In order to distinguish different sources of information flow, we define security policies which specify the sources and destinations between which secret data may flow.

3.6.1 Example

Let us reconsider the LoyaltyCard example, described in previous chapter at page 9. The LoyaltyCard is an open smart card, containing four loyalty applications. Three of them (*i.e.*, FlyFrance, MHz and Illtone) form a group of partners an can share fidelity points. In the previous sections we showed how we infer information flows in JVM in these applications and how we enforce non-interference. But, inevitably, the results, *i.e.*, the flow signatures in Figure 3.12 at page 46, contain flows rejected by non-interference. In this section we show how we define collaboration policies for the LoyaltyCard and how we certify the flow signatures w.r.t. the defined policies.

$$
\begin{array}{lll}
S & ::= & (Class|Package)[,\, S] \\
F & ::= & Field[,\, F] \\
R_c & ::= & Class\ \textbf{secret}\ F; \\
R_s & ::= & S\ \textbf{shares with}\ S; \\
R_p & ::= & S\ \textbf{strict secret ;} \\
P & ::= & (R_c|R_s|R_n)[,\, P]
\end{array}
$$

Figure 3.20: A DSL for information flow policies

3.6.2 A Domain Specific Language for information flow policies

A domain-specific language (DSL) is a small, usually declarative, language that targets a particular kind of problem. The key characteristic of DSLs is their focused expressive power. DSLs are usually concise, offering only a restricted suite of notations and abstractions, thus adapted to express security policies.

DSL definition

In order to express security policies describing collaborations and information flows between software units, we define the domain-specific language in Figure 3.20. The security policies that can be expressed with the DSL are simple, but have enough power to model collaborations schemes in an open system. The DSL was designed for multiapplication open systems, but it can be extended to other systems, if necessary. The DSL consists in a set of rules describing trust relations between terminals.

Terminals Multiapplication systems allow data sharing and service sharing in order to optimize the use of resources (*e.g.*, API) and to allow collaborative schemes (*e.g.*, agreements or contracts between applications). In an open system, the entities exchanging or sharing data are the software units, thus the DSL contains rules defining *trust relations* between software units. Software units are addressed either by package or class names, using elements in the sets of terminals $Class$ and $Package$; $Field$ denotes a set of terminals containing field names.

DSL rules The DSL contains two types of rules: *(1)* rules identifying sources of secret data (R_c) and *(2)* rules defining allowed flow of information between terminals (R_s, R_p).

Rule R_c : expresses the secrets of a class, by listing the fields that should remain confidential, and thus that have the security level s. This rule relies on our interpretation of data sinks in Java open systems: confidential data resides in instance fields.

Rule R_s : is the main rule of the DSL and describes the allowed information flows. For example, the meaning of S_1 **shares with** S_2; is that all elements in S_1 can share their secrets with all elements in S_2.

Rule R_p : rule $R_p = S$ **strict secret ;** refines the security policies by specifying that an element A in S must not share its secrets with other objects of type A. By default, an element in S can share its secret with other elements having the same type (*e.g.*, class A shares its secrets with all instances of class A).

While the rule R_s defines type-based policies, rule R_p refers to instance-based policies, in the case when the instances have the same type. Instance-based policies are stronger than type-based

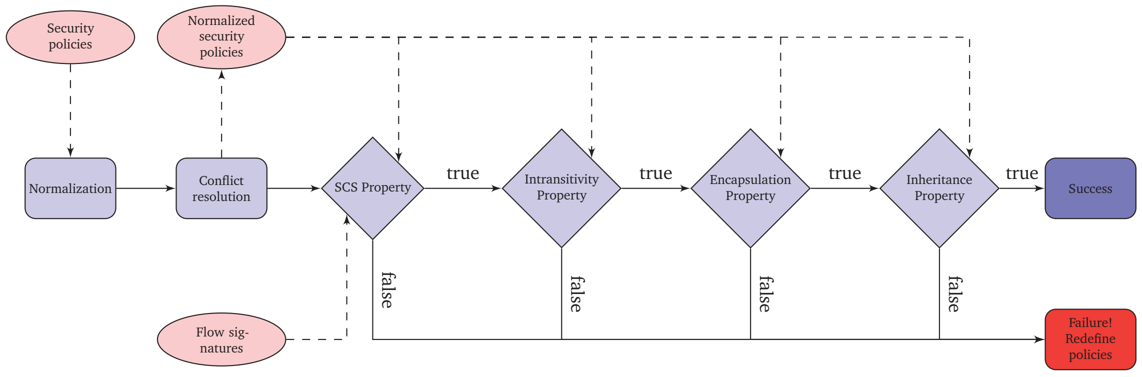


Figure 3.21: Certification process for security policy of class A

policies. Unfortunately, current models support type-based policies; the instance-based ones are difficult to define, as instance identification is difficult to achieve, especially in an open system.

The **shares with** relation can be associated to the *trust relation* defined in [Gir99] by Girard: one application transmits its secrets only to trusted applications. Moreover, the sharing relation has the same properties as the trust relation, as the sharing relation is neither symmetric nor transitive:

asymmetry : if A trusts B then not necessarily B trusts A; formally A **shares with** B does not imply B **shares with** A;

intransitivity : an application would not trust another application only because one of its trusted applications does.

Transitivity corresponds to data propagation. Detecting data leaks due to transitivity is one of the main concerns of information flow security. Allowing transitivity would make no distinction between information flow and access control.

3.6.3 Enforcing security policies

The certification process of an information flow policy consists of enforcing different security properties, mainly the asymmetry and intransitivity:

1. verifying *simple class sharing* (or SCS): an application gives its secrets only to trusted applications; this corresponds to the *asymmetry* property,

2. verifying *intransitivity* (or data propagation): an application trusted by A does not share confidential data with applications untrusted by A.

The *class sharing property* reduces to inspecting the flow signatures of methods and detect unpermitted data flows. Nevertheless, an object-oriented language offers other ways of bypassing security, such as *encapsulation* and *class inheritance*. In order to prevent all types of possible leaks, we identify a security property for each possible threat and give an algorithm which enforces the property. The certification process of the security policy of a class A is depicted in Figure 3.21. First, the rules of the DSL are normalized (packages are reduced to classes) and then possible conflicts are solved. Then, the security properties are verified on the normalized and conflicts-free security policies. The SCS property certifies the security policies w.r.t. flow signatures. Leaks arising from transitivity, encapsulation and inheritance are checked afterwards.

We now present the security threats that are treated at each step of the certification process, and we define the security property that must be verified in order to enforce the policies, under the assumptions that the security policies of all software units are available. In Section 3.6.7 we show how security policies are enforced at loading time, when software units are downloaded one by one.

Normalization

The rule P_i **shares with** P_j, where P_i and P_j are two packages, can be read as "any class in package P_i trusts any class in package P_j". Hence, the DSL in Figure 3.20 can be reduced to rules having the form A **shares with** B, where A and B are class names in $Class$. We can compute a function $share : Class \rightarrow \wp(Class)$ which associates to each class the classes it trusts. By default, a class trusts ifself, thus A **shares with** A**;** and $A \in share(A)$. Verifying the security policy of a class $A \in Class$ reduces to verifying that secrets of A flow only to elements in $share(A)$. Hence, we verify security policies at the granularity level of classes:

- ***Normalization property*** : all DSL rules are reduced to rules having the form A **shares with** B, where A and B are class names in $Class$. Security policies are enforced at the granularity level of classes.

Conflict resolution

While rule R_c and R_s are permissive, the rule R_p is restrictive and thus can generate conflicts. Let us consider the following policy for a class $x.A$, where x is the package to which A belongs:

$$x.A \textbf{ strict secret ; } x.A \textbf{ shares with } x.* \textbf{ ; } .$$

In the first rule, $x.A$ does not trust itself (*i.e.*, it does not trust instances of the same type), while in the second rule $x.A$ trusts all classes in package x, and thus it trusts itself. To solve such conflicts, we consider :

- ***Conflict resolution property*** : the rules R_p (**strict secret**) prevail over rules R_s (**shares with**).

Thus, we first construct the function $share$, which identifies all trusted classes, and only after we take into consideration the fact that a class A is **strict secret** or not; if A is **strict secret** , then we remove A from $share(A)$.

Simple class sharing (SCS) and flow signatures

We now show how policies defined using the DSL are enforced by the information flow analysis described previously. Confidential data resides in object fields. Let $fields_s : Class \rightarrow \wp(Field)$ be a function that associates to each class the fields having the security level s, thus fields in the rule $R_c = Class$ **secret** $Field$; the function $fields : Class \rightarrow \wp(Field)$ gives all fields of a class.

Secrets can be made accessible either by direct access to fields or through method invocations and operations performed by the method. In order to prevent direct access, secret fields of a class A in $fields_s(A)$ must be declared using the Java access modifier $private$. This restricts the access to secret fields only in the class where they have been declared and thus to which they belong.

Based on this, certifying a class A with respect to an information flow policy consists of verifying every method in A and methods that use the class A. Let $Method$ be the set of method names and $methods : Class \rightarrow \wp(Method)$ a function that gives the list of methods for each class.

Let us remind that our information flow model computes, for each method $m \in Method$, a flow signature S_m containing all the possible flow of information between abstract values in Σ_m (parameters, IO, Ex, $Static$, R). A flow is denoted by $a^{t_1} \xrightarrow{t} b^{t_2}$ with $t_1, t_2 \in L = \{s, p\}$ denoting the security level, and $t \in \mathcal{F} = \{\mathtt{r}, \mathtt{v}, \mathtt{i}\}$ denoting the type of interference (reference, value or

Algorithm 3.8 $scs(A)$: Certifying the SCS properties

```
1: for all $f$ in $fields_s(A)$ do
2:     if $f$ is not $private$ then
3:         return false
4:     end if
5: end for
6: for all $m$ in $methods(A)$ do
7:     if $\exists a^p \xrightarrow{f} b^s \in S_m$ then
8:         return false
9:     end if
10:    for all $a^s \xrightarrow{f} b^s \in S_m$ do
11:        $t_1 = \mathcal{T}(a)$, $t_2 = \mathcal{T}(b)$
12:        if $t_1 \notin share(t_2)$ then
13:            return false
14:        end if
15:    end for
16: end for
17: return true
```

implicit flow). Flows can be from public/secret parts of an abstract value to public/secret parts of another abstract value. Security is concerned with protecting flows from the secret parts to public/secret parts. As a general rule, flows from secret to public are forbidden, while flows from public to public are always allowed. The DSL defines that a class A shares its secrets with classes in $share(A)$; hence, only flows from secret parts of parameters of type A to parameters with type in $share(A)$ and to return (R) are allowed.

We can conclude with the two security properties for simple class sharing:

◇ ***SCS property 1 (for fields)*** : secret fields of a class A must be declared using the Java access modifier $private$.

◇ ***SCS property 2 (for methods)*** : flow signatures must contain only flows from secret parts of parameters of type A to the secret part of parameters with type in $share(A)$ and to return (R).

Let $\mathcal{T}$ be a function which associates to an abstract value its definition type, in $Class$. The algorithm that verifies the SCS properties in a class A is depicted in Algorithm 3.8. To permit the flows to return, we consider that $R \in share(A)$.

Intransitivity

Once a class A shares confidential data with a trusted class B, A loses control over its propagation. The secret of A becomes the secret of B. The policy of A holds if the policy of B is more restrictive: B does not share its secrets with applications untrusted by A. Formally, verifying safe data propagation can be summed up to verifying the intransitivity property:

◇ ***Intransitivity property*** : for all classes $B \in share(A)$, $share(B) \subseteq share(A)$ holds.

The algorithm enforcing the intransitivity property is depicted in Figure 3.9.

Algorithm 3.9 $intrans(A)$: Certifying the intransitivity properties

```
1: for all $B \in share(A)$ do
2:     if $share(B) \notin share(A)$ then
3:         return false
4:     end if
5: end for
6: return true
```

Encapsulation

The split of objects, due to our security level sensitive analysis, and the definition of secret/public part may open a door to bypassing security checks through encapsulation. For example the code `a.p.r=a.s`, where `s` and `r` have security level s and $\mathcal{T}(p) \notin share(\mathcal{T}(a))$, generates a flow $a^s \xrightarrow{\mathbf{v}} a^s$, allowed by our analysis (as, by default, $\mathcal{T}(a) \in share(\mathcal{T}(a))$), but illegal as the secret of a flows to an untrusted type ($\mathcal{T}(p)$). In order to avoid such leaks, we define the following encapsulation property:

◇ ***Encapsulation property*** : all secret fields and sub-fields of a class A must be trusted by A, where sub-fields refer to fields of fields and etc.

The verification of this property consists in unfolding the fields of each class and verifying that, for each secret field f, we have $\mathcal{T}(f) \in share(A)$. Let $encaps(A, B)$ be a function which verifies, recursively, that all secret fields of class B are trusted by A ($encaps : Class \times Class \rightarrow \{true, false\}$). The algorithm is depicted in Figure 3.10.

Algorithm 3.10 $encaps(A, B)$: Certifying the ecapsulation property (fields of B are trusted by A)

```
1: for all $f$ in $fields_s(B)$ do
2:     if $\mathcal{T}(f) \notin share(A)$ then
3:         return false
4:     end if
5: end for
6: for all $f$ in $fields(B) \setminus fields_s(B)$ do
7:     if $scr^*_C(\mathcal{T}(f))$ then
8:         return $encaps(A, \mathcal{T}(f))$
9:     end if
10: end for
11: return true
```

If we take into consideration that only few classes contain secret fields, we can label the classes containing only public fields and stop the unfolding when we meet such classes. Let $scr_C : Class \rightarrow \{true, false\}$ be a function which tests if a class contains some secret fields or not; $scr_C(A)$ refers only to secrets defined in A. We define by scr^*_C the closure of scr_C, which refers not only to secret fields defined in A but also to secret fields defined in fields of A, etc. Thus, to verify that a class A respects the encapsulation property, a call to $encaps(A, A)$ is sufficient.

3.6.4 Support for open class hierarchy

One of the most powerful attributes of object-oriented programming, and thus Java, is code reuse and factorisation, by the means of inheritance. But, apart from providing this powerful functionality, inheritance also provides means for leaking information. To prevent such leaks, we define some relations between policies of subclasses and superclasses.

Inheriting fields

The first restriction regards inherited fields:

◇ ***Inheritance property 1 (inherited fields)*** : security levels of inherited fields cannot be changed by the child class.

If the child class changes the security levels of inherited fields, then flow signatures of inherited methods change, and the superclass must be reanalyzed. This is not convenient for our compositional approach, and for open systems. Nevertheless, a child class can declare new fields even with security level s.

Overridden methods

While extending a class, for example `B extends A`, the security policy of B must not only enforce security for B (**incrementality**), but also for A and classes already verified using A (**consistency**). If the policy of B is greater than the policy of A, formally $share(B) \supseteq share(A)$, then the confidentiality of A is not respected anymore, as B can trust and share its secrets (and thus those of A) with classes which A does not trust. If the policy of B is smaller than the policy of A, formally $share(B) \subseteq share(A)$, in order to certify B we must reanalyse A, as A, and thus a part of B, have been certified using a greater policy. From these examples, we can conclude with:

◇ ***Inheritance property 2 (overridden methods)*** : the security policy of a class B must be the same as the security policy of its superclass A; formally, $share(B) = share(A)$.

Algorithm 3.11 $inheritance(B, A)$: Certifying inheritance (the policy of class B extends A)

```
Require: $A \leq B$
1: if $fields_s(B) \setminus fields_s(A) \cap fields(A) \neq \emptyset$ then
2:     return false
3: end if
4: if $scr^*_C(A) \wedge share(B) \neq share(A)$ then
5:     return false
6: end if
7: return true
```

Extension for *public* classes: API, interfaces, abstract classes

The constraint above is too strict for API classes, which are *public classes* (we use this term to denote classes which do not contain secret fields, hence classes for which scr^*_C returns $false$). By default, if A is such a class, then $share(A) = \emptyset$. If B extends A, then the inheritance property 2 requires that $share(B) = \emptyset$, which is too restrictive. For example, $share(Object) = \emptyset$ and all classes extend $Object$; with the rule above, we are not allowed to define new security policies and all classes should have an empty policy. In order to deal with such classes as $Object$, and hence also API classes, we relax the policy above in the following way:

◇ ***Inheritance property 3 (relaxation)*** : the policy of a subclass must be the same as the policy of the inherited class only if the inherited class contains secret fields.

Thus, the policy of a subclass can be any policy, if the inherited class is a public class. Problems may arise if we cast a public class to a class which contains secrets. To deal with such situations, we extend the flow signature with the types in which public classes are cast inside the method, and we take into consideration all these types while verifying the SCS property of the method.

For example, let us consider that we have `B extends A`. Security issues arise when class A does not contain any secrets ($\mathit{fields}_s(A) = \emptyset$) and B declares secret fields ($\mathit{fields}_s(B) \neq \emptyset$). In this case, the leak occurs only when a cast from A to B is made inside a method m. To solve this problem, while analysing m, we extend the flow signature of m with the types in which classes of type A are cast inside m.

Let $cast_m : \mathcal{T} \to \wp(\mathcal{T})$ be a function which associates to each type the types to which it may be cast while executing the method m. In order to compute $cast_m$, we extend the prover. The extension is straightforward and it is strongly related to the computation of exact types. If $cast_m$ returns only the direct cast, then the function $cast^*_m$ retuns the closure of $cast_m$. For example, if A is cast to B and B is cast to C, then $cast_m(A) = \{B\}$ while $cast^*_m = \{B, C\}$.

Using the $cast_m$ function, we can now define the **extended flow signature**, $\widehat{S_m}$:

$$\widehat{S_m} = (S_m, S_m^{cast})$$

where

- S_m is the flow signature,
- S_m^{cast} contains the types in which public classes are casted inside m. Formally,

$$S_m^{cast} = \{(\mathcal{T}(p_i), cast^*_m(\mathcal{T}(p_i))) \mid p_i \in P_{|Obj} \wedge scr^*_C(\mathcal{T}(p_i)) == false\}.$$

S_m^{cast} associates to the type of each parameter p_i of type reference ($p_i \in P_{|Obj}$), which does not contain secret fields ($scr^*_C(\mathcal{T}(p_i)) == false$), the list of classes to which it can be cast inside m ($cast^*_m(\mathcal{T}(p_i))$).

For example, if the parameter p_1 of type B might be cast in C or in D in method m, then $S_m^{cast} = \{(B, \{C, D\})\}$. In other words, $S_m^{cast}(B) = \{C, D\}$; if B is of primitive type or it is not a public class (contains secret fields) or it is never casted inside the method, then $S_m^{cast}(B) = \emptyset$. This list must be kept only for types which do not contain secret fields, thus for which the function scr^*_C does not hold.

The algorithm in Figure 3.11 certifies the inheritance property for fields and for classes which contain secret fields. In order to certify the third inheritance property, refering to public classes, which do not contain secret fields, we must extend the verification of *simple class sharing propery* (SCS), in Figure 3.8, to take into consideration the extended flow signatures. We achieve this by defining a special $\mathcal{T}_{\widehat{S_m}}(A)$ which gives for a class A, and for a method m with extended flow signature $\widehat{S_m}$, the type in which A can be casted inside m. Formally,

$$\mathcal{T}_{\widehat{S_m}}(A) = \mathcal{T}(A) \cup S_m^{cast}(A).$$

To complete the verification, we only have to replace $\mathcal{T}$ by $\mathcal{T}_{\widehat{S_m}}$ at line 11 of Algorithm 3.8, which certifies the SCS property:

11: $t_1 = \mathcal{T}_{\widehat{S_m}}(a)$, $t_2 = \mathcal{T}_{\widehat{S_m}}(b)$

Hence, when the security policies is certified w.r.t. the flow signature, will account not only for the type parameter, but also for types into which parameters can be cast.

3.6.5 Example

Figure 3.22 presents the information flow policy for the LoyaltyCard example. The first three rules define the confidential data, while the last two rules define the allowed information flow. The policy respects transitivity, as the policies of applications trusted by FlyFrance (MHz, Illtone) are smaller

$FlyFrance$ **secret** $miles$;
MHz **secret** $ppoints$;
$Illtone$ **secret** $ppoints$;
$FlyFrance$ **shares with** MHz, $Loyalty$, $Illtone$;
MHz **shares with** $Illtone$;

Figure 3.22: Security policy for LoyaltyCard

than the policy of FlyFrance. The verification fails while trying to validate the method `discount` defined in `FlyMaroc` (Figure 3.12 at page 46, as it contains a flow $p_0^{p,s} \xrightarrow{\texttt{i}} p_1^{s,p}$, where p_1 denotes the first parameter of the method, *i.e.*, the MHz application.

3.6.6 Extensions

Localized declassification

The DSL and the security policies can be extended to express more detailed rules about the release of information. The current DSL expresses policies that apply to entire program, and does not specify *where* the information release is permitted. We can define rules that delimit the methods where the information flow may occur, for example

$$R_m ::= S \textbf{ shares with } (\textbf{IO} \mid S) \textbf{ in } Method;$$

where $Method$ represents a method name or a list of methods and **IO** is a keyword (terminal) standing for the abstract value IO. The declassification adds power of expression as it also allows to send data on input/output channels.

Declassification relaxes the security policies in certain method. To support polymorphism and dynamic class loading, all the overridden classes must agree on the declassification contract:

- ***Declassification property*** : the declassification rule R_m defined in a method $C.m$ must be defined by every class in the class hierarchy derived from C.

Information flow policies as contracts

The DSL in Figure 3.20 allows the declarations of information flow policies for applications sharing confidential data. Not only this language has a declarative value, but it also has a contractual value. For example, with the rule $FlyFrance$ **shares with** MHz, FlyFrance imposes a contract to MHz: FlyFrance agrees to share its secrets with MHz only if MHz does not share its secrets with applications not trusted by FlyFrance. Thus, the policies defined using the DSL are contracts that applications must respect. An application accepts the contract of a trusted application only if it is smaller than its own contract. In order to deal with openness and overriding, the DSL imposes that the contracts of classes extending classes containing confidential data do not change, with respect to the contract of overridden classes.

3.6.7 Embedded verification of security policies

We have defined so far the DSL for information flow policies and we have shown how these policies can be enforced under the assumption that all components of the system and all security policies are available.

As the compiled JVM bytecode is downloaded through an unsecured channel, the information flow certification must be done oncard, preferably at load time, in order to avoid run-time overhead. In this section, we present how information flow policies defined in previous section can be enforced by any JVM.

The security policies are shipped on board within the code, as annotations to .class files and verified by a custom class loader. Classes are loaded one by one, hence we have to deal with linear verification and with security policies not yet loaded.

Both flow signatures and policies must be enforced by the embedded verifier. While the verification of flow signatures was described in Section 3.4, here we deal only with the verification of information flow policies.

Hence, in order to make the analysis practical and integrable with any existing JVM system, we

- load policies to be certified as attributes of .class files; systems not enforcing information flow can ignore these attributes,
- verify security policies with a custom class loader, that can be installed on any system.

Annotating .class files with information flow security policies

The policy of a class A is the list of classes with which it can share confidential data (denoted by $share(A)$). The .class attribute for the policy of A contains thus a list of class names. The class names are represented by their index in the ConstantPool of the class A. Considering that in a small open system the number of installed applications is not significant, thus the sharing policies are quite simple, the newly added attribute contains usually only few entries. The small size of the attribute is acceptable for a small system.

As classes are loaded one by one, it is possible to load A before loading all the classes used by A. While validating a class A, we also validate the policies of classes used in A. Thus, to be able to validate A, we also load the policies of classes used in A. If B is a class used by A, when loading A either *(i)* we take into consideration the policy of B, if B has already been loaded or *(ii)* use the policy of B that A announces and we keep it on board, in a temporary repository, in order to validate (and remove) it when B is loaded.

Verifying security policies using a custom class loader

The loading process in a JVM is performed by the class loaders. In order to integrate the information flow analysis on any JVM, the verification is performed by a custom class loader ($SafeClassLoader$), which can be built in the single class loader of KVM or installed as a user-defined class loader for a standard JVM. The $SafeClassLoader$ must verify both flow signatures and security policies.

Classes are loaded one by one. Once the flow signatures of the class have been verified, the $SafeClassLoader$ validates the information flow policy, using the flow signatures. The difficulty may arise from the fact that the loaded class A wants to share its secret with a class B not yet loaded. As the class is not present in the system, we do not have its security policy and we cannot verify the intransitivity, formally

$$share(B) \subseteq share(A).$$

In order to verify this condition when B is loaded, we keep a repository with rules having the form $share(B) \subseteq share(A)$. If B is used by another class C, the rule $share(B) \subseteq share(C)$ must be added to repository. In this case, the final rule kept in the repository is $share(B) \subseteq share(A) \cap share(C)$, as the policy of B should be more restrictive than both policies of A and C.

Thus, when we load B, we also verify that $share(B) \subseteq X$ with X denoting the intersection of security policies of classes that trust B. Moreover, we verify that the loaded class has the same policy as its super class: $share(B) = share(B')$ with B extends B'.

The management of the repository of security policies is similar to the management of the repository of flow signatures ; hence, we will not detail the algorithms again. In a few words, the repository of security policies sums up to:

- the security policies of each class,
- the security policies of classes not yet loaded, but used while validating already loaded class. These security policies are temporary stored; they correctness is verified when the classes are loaded, and after they are deleted, on success,
- some security contracts needed to enforce the *encapsulation* property, as it will be describe in a brief moment.

Hence, the lookup algorithm for flow signatures (in both cases when we have a single class loader or an hierarchy of class loaders) is easily extended to the lookup for security policies.

Verifying encapsulation

While loading a class and its security policies, all security properties depicted in Figure 3.21 must be enforced. In all the cases, except the encapsulation property, the verification algorithm is identical. Special attention should be payed to the encapsulation property, as its certification uses a recursive algorithm.

The problem that may arise when verifying encapsulation is similar to the one described above: we load A, which has a field of type B, but B has not been yet loaded. The encapsulation property of A depends on the encapsulation property of B. In order to verify, while loading B, that all secret fields of B are trusted by A, we keep the following rule to the repository:

$$do\_encaps(A, B).$$

When loading B, if a rule $do\_encaps(A, B)$ is found in the repository, than the function $encaps(A, B)$ (see Figure 3.10) is executed. If the test succeeds, the rule is removed from repository and the loading process continues, by performing other checks.

The result of $encaps(A, B)$ depends on $scr^*_C(B)$ (the function which tests if B or fields B contain secret fields). The value returned by $scr^*_C(B)$ depends also on fields of B. Hence, the final value of $scr^*_C(B)$ can be computed only when all fields, fields of fields, etc. have been loaded. To ensure the correctness of scr^*_C computation, we extend the repository with rules of type

$$scr^*_C(B) = scr^*_C(C_1) \vee scr^*_C(C_2) \vee \ldots \vee scr^*_C(C_n),$$

where $C_1 \ldots C_n$ represent the type of fields of B not yet loaded. This rule is deleted from repository when a class C_i is loaded with $scr^*_C(C_i) = true$ or when all classes $C_1 \ldots C_n$ are loaded. Moreover, to avoid the computation of scr^*_C each time when it is needed, the known values of scr^*_C are stored on the card, in a special repository.

Hence, the algorithm verifying encapsulation at loading time is similar to the external one ($encaps$) presented in Figure 3.10, except that it must also verify that the class $\mathcal{T}(f)$ has been loaded; if not, we add $encaps(B, A)$ to the repository.

3.7 Conclusion

This chapter describes a new model for detecting illegal information flows in small, Java enabled, open systems. The model addresses one of the most important limitations of previous work in this area: it runs in an open world and supports dynamic class loading. We annotate methods with statically computed signatures that are shipped with the code and verified at loading time. Openness allows us to accept loading new classes, but patched or modified classes can be reloaded only if they are compatible with already loaded and certified signatures.

We have achieved three challenges:

- to show the usefulness of information flow security mechanisms by applying them to real problems,
- to support separate compilation and dynamic downloading of different software units,
- to allow the embedded environment (the code receiver) to certify the security of hosted standard Java applications by itself.

The model is easy to use and can be applied to already existing programs as the separation of concerns between the application functionality and security is ensured. Full Java features are supported.

Motivated by the LoyaltyCard example, we extend the information flow framework with security policies which define collaborations schema in multiapplication open systems. The desired security policies are specified using a simple, but expressive domain specific language and are enforced are load time.

On the one hand, this model bridges the gap between information flow models and current running systems. While the foundations of information flow models are solid, their practical side is still to be proved. Our approach limits the overhead for adding information flow security to existing JVM, as security labels and policies are separated from the code, and the illegal information flow is detected by a custom class loader, installed on any JVM.

On the other hand, our model bridges the gap between information flow security requirements and actual security policies, which do not take into consideration data propagation due to information flow.

4 Applying information flow analysis in practice

Contents

In the previous chapter, we define an information flow model which enforces non-interference for Java-enabled embedded systems. From the start, the model was designed to address real problems raised by small, open environments: limited resources, dynamic loading, etc. In this chapter we show how our model can be successfully applied in practice in the context of small open systems.

First, we briefly present the tools implementing our model and we describe the results of experiments run on some significant benchmarks. On the one hand, experimental results (such as execution time, proof and certificate size) show that the two steps approach is justified, allowing dynamic class loading. On the other hand, they prove that the verifier can be successfully embedded, with an acceptable overhead at loading time and an acceptable size for the certificate that must be kept onboard. We also compare our models with other information flow implementations for Java and we discuss why our tool better fits the environment of embedded systems.

Secondly, we make a step further ahead to make the model easy to use by integrating it in a security analysis framework, which combines our static analyzer with a quantitative design security analysis technique (developped at University of Victoria, Canada). The security analysis framework ensures verification during the entire lifecycle of software developement, which is highly desired in order to eliminate security leaks due to bad design from early stages. To illustrate our approach, we present the case study of a medical card, the CareCard, holding the medical file of a patient.

Finally, we apply the information flow analysis on a particular case of embedded Java applications for mobile phones, *i.e.*, MIDlets. We show that our analysis can be easily adapted and applied to different cases. To better illustrate the approach, we reconsider the case study of the medical card reimplemented as a MIDlet suite.

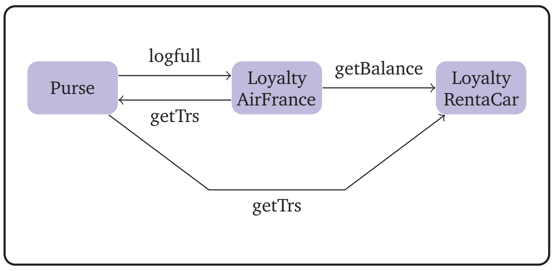


Figure 4.1: Information flow in PACAP

4.1 Implementation and experimental results

4.1.1 Prototype tools: STAN and VSTAN

Historically, JAVACARD was the first object oriented language dedicated to embedded systems and dealing with the execution of Java code on processor cards (like smart cards). Nowadays, the embedded virtual machines [Sun, Aon, Jav] are evolving towards the standard Java language. To address future evolutions, our model and the implementation support standard JVM language.

To test our model, we have implemented two tools:

1. **STAN** - the prover, which acts as a stand-alone tool for the JVM language; STAN implements the information flow model described in the previous chapter; it is a Java abstract analyser that allows static alias analysis and information flow certification. The STAN tool statically checks already compiled source code, computes the type inference and annotates the .class files with proof elements and flow signatures verifiable at loading time,

2. **VSTAN** - the verifier, which is implemented as a custom class-loader; as already discussed in Section 3.4 at page 48, in order to check information flow at loading time, the user must use the classloader provided by VSTAN, *i.e.*, *SafeClassLoader*. VSTAN verifies JVM bytecode annotated by STAN.

The two prototypes, documentation and tutorials can be found at `http://www.lifl.fr/~ghindici/STAN`.

4.1.2 Use case study

Besides the LoyaltyCard example, we run experiments on a well known application from the literature, a revised version [BFLM04] of the electronic purse case study of the PACAP project [BCG$^+$02]. PACAP project aims at checking information flows between objects on a JAVACARD platform using static analysis and a given configuration.

The electronic purse case study consists of three applets, a purse applet and two loyalty applets: a frequent flyer application (Loyalty AirFrance) and a car rental loyalty program (Loyalty RentaCar). The applets can share data through common interfaces. The Electronic purse performs some administration functions, and credit and debit operations which are kept in a transaction log.

Moreover, the electronic purse offers a service, *logfull*, which notifies the registered applets before overwriting the log. Registered applets have a chance to update their loyalty points before old transactions are deleted. Only Loyalty AirFrance has subscribed to *logfull* service, as the registration to this service requires the payment of a fee. At the same time, there is an agreement between AirFrance and RentaCar to exchange loyalty points: RentaCar points can be substituted for AirFrance points. So, when AirFrance updates its points, it also asks the balance of the RentaCar Loyalty.

			Prover (external)					Verifier (embedded)				
Benchmark	Classes	Methods	Class iterations	Bytecode iterations	Analysis time (s)	Average memory (Kb)	Maximum memory (Kb)	Execution time CL (s)	Execution time SCL (s)	Verification time SCL (s)	Average memory (Kb)	Maximum memory (Kb)
Dhrystone	5	21	3	1.47	5.4	4.26	35.94	0.12	0.47	0.40	0.78	3.80
fft	2	20	3	1.82	6.8	1.72	7.98	0.05	0.23	0.21	0.67	3.33
_201_ compress	12	43	3	2.21	7.7	3.52	20.84	0.32	0.59	0.36	0.85	4.31
_200_check	17	109	4	1.20	15.2	5.04	34.60	0.12	0.89	0.81	1.26	6.64
crypt	2	18	3	1.66	9.8	2.22	20.78	0.07	0.31	0.26	0.83	6.96
lufact	2	20	3	2.31	3.9	4.35	23.33	0.52	0.91	0.35	0.81	2.29
raytracer	12	72	5	1.85	8.7	2.54	25.23	0.08	0.58	0.54	0.84	3.33
Pacap	15	92	4	1.06	7.5	6.62	92.84	0.03	0.40	0.38	1.01	6.01

Figure 4.2: External analysis and embedded verification measurements

Supposing that AirFrance updates its points only when it is notified by the *logfull* service. In this case, when Loyalty RentaCar receives a request from AirFrance, it can infer that the log is full even if it has not subscribe for the service (Figure 4.1). It is not an implicit flow generated by the control structure of the program, but a flow generated by objects interactions. Such flows are detected by our analysis.

We ran two experiments on the electronic purse case study: first, we checked non-interference, and, secondly, we detected the leak of information generated by the *logfull* service, as other papers on PACAP did [BFLM04, BCG+02].

The electronic purse can perform credit/debit operations and administration functions. Before allowing these operations, successful authentication is necessary. The Java representation of electronic purse contains fields that ensure authentication (*pin*, *apin*, *puk*): the values of these fields must remain confidential. To check this property, we ran our algorithm with a policy specifying that these fields are secret and we analyzed the flow signatures of methods in the Electronic purse. The result showed that the secret fields cannot be accessible through any method nor directly: methods signatures do not contain any links to secrets and all secret fields have `private` access right.

The second experiment regards the *logfull* service offered by the Electronic Purse which can be accessed only by the AirFrance Loyalty, and cannot be accessed by the RentaCar Loyalty. Still, the unauthorized loyalty can access the service through the AirFrance loyalty (see Figure 4.1). We detected this leak of information using the following policy: the service is a secret of the purse, and the purse can share its secrets only to the allowed loyalty (AirFrance). The illegal information flow was found while we were trying to proof loyalty RentaCar, so, the embedded system will refuse to load RentaCar because of the used policy. We detected the same leak as the other papers [BFLM04, BCG+02] on PACAP, despite the fact that our conditions are more restrictive (open world and embedded systems with restrained resources, while PACAP considers a close world).

4.1.3 Experimental results

This section describes the results of experiments run on some significant benchmarks such as Dhrystone, a well known benchmark for embedded systems, The Fast Fourier Transform (FFT), a common signal processing application, crypt (a data encryption algorithm), and PACAP [BCG+02]. We ran the experiments using a Java Runtime Environment Standard Edition (build 1.5.0_09), on a Linux system running on a Intel(R) Pentium(R) M processor 2.13GHz with 1Gb memory.

First, we ran the external application (the prover) computing flow signatures and annotating the classes (Figure 4.2, Prover) in order to find out how the algorithm performs in practice. We

Benchmark	Initial .class size (Kb)	Annotated .class (Kb)	Flow Signatures (%)	Labels proof (%)	External methods (%)	External fields (%)
Dhrystone	8.2	14.4	8.00	52.22	5.15	0.20
fft	6.8	15.1	16.60	91.09	7.23	0
_201_ compress	20.1	28.3	3.95	23.66	4.36	0.34
_200_check	46.3	97.7	12.27	85.05	6.75	0.04
crypt	7.0	17.0	12.27	118.28	5.90	0.07
lufact	9.3	17.0	8.44	64.31	4.07	0.39
raytracer	24.0	42.8	20.44	36.12	12.06	0.57
Pacap	26.8	52.0	18.36	55.72	9.03	0.37
Total	148.5	284.3	12.54	65.80	6.81	0.24

Figure 4.3: Size of annotations

measured the number of iterations for the inter-method analysis (iterations on a set of classes), the iterations for the intra-method analysis (iteration on each method instruction set), and the time needed to perform the analysis. The results showed that the computation algorithm is quite expensive in terms of execution time: in average, we need 3 iterations on the set of classes, 1.7 iterations on the instruction set, and 0.22s for each method. For the JVM *spec* benchmarks, we performed the library analysis before carrying out the experiments.

Secondly, we loaded the annotated software units generated by the prover (Figure 4.2, Verifier). In order to find out how the JVM loading process is hampered by our verification, we measured the execution time in two cases: with (SCL) and without (CL) information flow verification. We observed that the verification implies an average execution time 3 times larger than the standard one. But the information flow verification is performed only once, at loading time, so any subsequent execution of the software units is not hindered. Moreover, the average verification time for a method (0.013s) is more than 15 times smaller than the average analysis time (0.22s). As expected, the verifier performs much faster than the prover.

Lastly, we measured the size of the proof and the flow signatures loaded with the code (Figure 4.3). The proof, the pending flow signatures, and the pending security levels for fields represent 66% of the total size of initial .class files. These data are used only during the verification process, at loading time, and it is not stored on the device so its size does not have a significant impact on the small open system. The flow signatures, which are stored in the dictionary and kept in the system, make up 12% of the initial .class size, an acceptable overhead. Moreover, the flow signatures can be used for other program analysis optimizations, such as escape analysis [GHSR07] and efficient memory placement.

4.1.4 The impact of implicit flow

The first results we had for our analysis did not account for implicit flow. Adding support for implicit flow and comparing the results lead to some interesting discussions and remarks regarding the role of implicit flow in information flow security.

We first compare (Figure 4.4) the performances of the prover when it takes into account implicit flow (**IF**, white columns) and when it does not (**WIF**, gray columns). While the number of iterations for inter-method and intra-method analysis does not change considerably, the analysis time is quite different. Sometimes, the **IF** analysis is even two times slower than the **WIF** analysis. As the number of bytecode iterations does not change, the slowdown is due to the fact that the number of flows increases. Hence, calls to the function which computes the closure over the flow signatures are much more frequent. We recall that the closure is computed after *each* flow generation.

			Prover (external)									
Benchmark	Classes	Methods	Class iterations		Bytecode iterations		Analysis time (s)		Average memory (Kb)		Maximum memory (Kb)	
			IF	WIF	IF	WIF	IF	WIF	IF	WIF	IF	WIF
Dhrystone	5	21	3	3	1.47	1.35	5.4	3.1	4.26	2.66	35.94	35.80
fft	2	20	3	3	1.82	1.55	6.8	3.2	1.72	1.50	7.98	7.86
_201_ compress	12	43	3	3	2.21	1.84	7.7	4.4	3.52	2.02	20.84	20.68
_200_check	17	109	4	4	1.20	1.18	15.2	8.7	5.04	3.87	34.60	34.45
crypt	2	18	3	3	1.66	1.32	9.8	4.1	2.22	2.91	20.78	20.58
lufact	2	20	3	3	2.31	1.75	3.9	3.4	4.35	3.90	23.33	23.22
raytracer	12	72	5	5	1.85	1.53	8.7	4.6	2.54	1.30	25.23	25.10
Pacap	15	92	4	4	1.06	1.05	7.5	4.7	6.62	3.00	92.84	92.68

Figure 4.4: External analysis measurements with (**IF**) and without (**WIF**) implicit flow

			Verifier (embedded)									
Benchmark	Classes	Methods	Execution time CL (s)		Execution time SCL (s)		Verification time SCL (s)		Average memory (Kb)		Maximum memory (Kb)	
			IF	WIF	IF	WIF	IF	WIF	IF	WIF	IF	WIF
Dhrystone	5	21	0.12	0.11	0.47	0.37	0.40	0.31	0.78	0.36	3.80	2.05
fft	2	20	0.05	0.06	0.23	0.17	0.21	0.16	0.67	0.39	3.33	1.80
_201_ compress	12	43	0.32	0.32	0.59	0.52	0.36	0.25	0.85	0.37	4.31	2.31
_200_check	17	109	0.12	0.12	0.89	0.88	0.81	0.79	1.26	0.58	6.64	3.51
crypt	2	18	0.07	0.04	0.31	0.26	0.26	0.22	0.83	0.56	6.96	3.68
lufact	2	20	0.52	0.53	0.91	0.87	0.35	0.30	0.81	0.45	2.29	1.25
raytracer	12	72	0.08	0.08	0.58	0.44	0.54	0.42	0.84	0.39	3.33	1.80
Pacap	15	92	0.03	0.03	0.40	0.28	0.38	0.27	1.01	0.38	6.01	3.19

Figure 4.5: Embedded verification measurements with (**IF**) and without (**WIF**) implicit flow

While the external analysis worsens due to the implicit flow, the embedded verification (Figure 4.5) is almost unaffected. This is due to the fact that the analysis is linear in time and code.

But the most significant consequence of the implicit flow is the increased size of annotations, which almost double, as depicted in Figure 4.6. Size of flow signatures (4.08% in **WIF** and 12.54% in **IF**) has tripled. As we have used the same encoding in both cases (one byte to encode a flow), the difference shows that almost two thirds of flows are implicit flows! This proves the complexity of information flow analysis comparing to other program analysis techniques (*e.g.*, points-to analysis which has to deal only with reference flows). Size of label proofs (36.06% in **WIF** and 65.80% in **IF**) is two times bigger if implicit flow is taken into account, but the proof elements are loaded one

Benchmark	Initial .class size(Kb)	Annotated .class (Kb)		Flow Signatures (%)		Labels proof (%)		External methods (%)		External fields (%)	
		IF	WIF	IF	WIF	IF	WIF	IF	WIF	IF	WIF
Dhrystone	8.2	14.4	11.9	8.00	3.17	52.22	23.07	5.15	2.42	0.20	0.20
fft	6.8	15.1	11.3	16.60	4.53	91.09	48.47	7.23	5.17	0	0
_201_ compress	20.1	28.3	28.3	3.95	3.79	23.66	23.36	4.36	4.21	0.34	0.33
_200_check	46.3	97.7	80.3	12.27	4.62	85.05	57.25	6.75	3.82	0.04	0.05
crypt	7.0	17.0	12.3	12.27	6.04	118.28	58.34	5.90	4.66	0.07	0.19
lufact	9.3	17.0	14.3	8.44	3.37	64.31	43.00	4.07	2.06	0.39	0.40
raytracer	24.0	42.8	33.4	20.44	1.41	36.12	16.62	12.06	5.51	0.57	0.63
Pacap	26.8	52.0	36.9	18.36	5.71	55.72	18.37	9.03	3.46	0.37	0.38
Total	148.5	284.3	228.7	12.54	4.08	65.80	36.06	6.81	4.03	0.24	0.27

Figure 4.6: Size of annotations with (**IF**) and without (**WIF**) implicit flow

at a time and discarded when the loading process finishes.

So, as we expected, the implicit flow plays a very important role in an information flow analysis. Due to implicit flow, the number of flows in a method has almost doubled, which, in our opinion, is a normal result as methods contain many conditional bytecodes.

4.1.5 Comparison with other tools (*Jif* and *SecJ*)

In this section, we compare our tools, STAN and VSTAN, to the other tools enforcing information flow security for Java programs. We concentrate on *Jif* [Mye99b], the most complete and mature tool for information flow, and *SecJ* [Sun08], the closest prototype tool to ours, due to its modularity.

Source code vs. bytecode The first significant difference is that both *Jif* and *SecJ* work as a source-to-source compiler. Both tools define their own languages, which are extensions of the Java language, enriched with security levels annotations. The results of the compiler is a standard Java source file.

Hence, the first step in a program verification is to rewrite the program from regular Java to the new language syntax. *Jif* is very mature, as it supports almost all Java features and its language, *JFlow*, subsumes the Java language syntax: most of the programs can be compiled without any rewriting.

SecJ is still a prototype and its syntax is fairly simple, so considerable effort is needed to rewrite programs. But many syntax restrictions are mainly because *SecJ* is still a prototype. Interfaces and exceptions are not supported.

In contrast, STAN does not define its own language and it works directly on JVM bytecode. All JVM programs can be analysed by our tool. To use *SecJ* and *Jif*, programs must be rewrite, hence programming knowledge is required; moreover programmers must learn and deal with a new syntax. In contrast, using STAN does not require any programming knowledge and hence it can be done by any user. The user must only specify the security policy, in a separate file; to do this, he/her should be able to identify the field which stores sensitive data.

Dynamic code download The reason while STAN checks JVM bytecode is that, in our target environment (*i.e.*, open systems), the only place where security can be guaranteed is the JVM itself. Performing the certification earlier, on source code, as *JFlow* and *SecJ* do, would be harmful as *(1)* the deployment context is unknown at compilation time and *(2)* the code is downloaded through an unsecured channel.

Security labels annotations As said before, *Jif* and *SecJ* have their own syntax and Java code must be rewritten: code must be annotated with security levels. STAN completely separates code and security policies, which are specified in a file apart. Security policies consist of labeling sensitive fields; if a field is not labeled, then its security level is considered as public.

In *SecJ*, security levels are part of the syntax; all types are augmented with security levels. As in STAN, only few annotations are necessary, if the security policy is not complicated. The rest of security levels are inferred using the inference engine. Figure 4.7 shows an excerpt from the LoyaltyCard applications rewritten in *SecJ*. Note that the type of field `FlyFrance.miles` is augmented with security level H (by default, the security lattice in *SecJ* contains two elements, H and L, which correspond to our p and s security levels). Also note that we had to slightly modify the code: we had to bypass the need for interfaces and hence to eliminate interface `Loyalty`, as abstract classes and interfaces are not supported in *SecJ*. As a consequence, `MHz` does not extend `Loyalty` interface anymore, while class `FlyFrance` has a member field for each partner instead of an array of `Loyalty`. Moreover, we had to move the `return` statement at the end of each method and we replaced static fields with constants.

```
class FlyFrance {
  private [int,H] miles;
  private MHz mhz;
  private Illtone ilt;

  public void updates(){
    mhz.update(miles);
    ilt.update(miles);
  }
}
```

```
class MHz{
  private int points;
  private int ppoints;

  public void update(int p){
    this.ppoints += p;
   }

  public int getLevel() {
    int p =  1;
    if(points+ppoints>1000)
      p = 2;
    return p;
  }
}
```

```
class FlyMaroc{
  int discount;

  void discount(MHz h) {
    int level=h.getLevel();
    if(level == 2) {
      this.discount = 20;
    }
  }
}
```

Figure 4.7: Excerpt from the Java implementation of LoyaltyCard in *SecJ*

In *Jif*, adding annotations is more complex and it requires much development time. Every class member (field, method, local variables, etc.) must be labeled. On the one hand, this leads to more flexibility and more sophisticated security policies, which cannot be expressed in STAN or *SecJ*. On the other hand, STAN and *SecJ* are static checking tools, so they can never adapt to run-time variables. On contrary, *Jif* integrates dynamic labels: when static checking is too restrictive, which allows performing some checks at run-time.

Modularity and signature inference A crucial property in the context of dynamic code downloading is modularity: pieces of code can be certified separately, as all classes are not loaded in the same time and the call graph is not available. *Jif* does not offer support for modularity and relies on the call graph, hence it cannot be used in an dynamically evolving environment. On the other hand, modularity is an important feature in *SecJ*. As in STAN, inference techniques are used to compute a signature for each method. The *SecJ* signature contains constraints which describe the contexts where the methods can be instantiated securely.

Figure 4.8 shows the security signatures produced by *SecJ* for methods in class `MHz`. If possible, *SecJ* infers security levels such that the method is secure: for example, the analysis has inferred that the field `ppoints` must have the security level H. Otherwise, the analysis uses levels variables (which represent a constant from the security lattice) and the signature is made generic.

The security signature `v37,() --<;v36>--> [int,H];` is inferred for method `getLevel`. This means that the return of the method must have the level H, while `v36` and `v37` are level variables for the heap effect of the method and the self level of the method (*i.e.*, the security level of the receiver cannot be lower than the self level).

For the FlyMaroc class, *SecJ* also infers that the field `discount` must have the security level H (as depicted in Figure 4.9). To detect the illicit flow, the field `discount` should be annotated with level L.

Jif performs only local type reconstruction, in the Java style. In particular, in *Jif* programs, methods arguments must be annotated with their whole type, including the security annotations. *Jif* programs do not need a separate signature file, because the labels in the program already represent the full security policy.

Running in an embedded context To adapt to sparse resources of open systems, we do not track fields individually. Our analysis is field independent, which leads to loss of precision. On contrary, *SecJ* individually tracks every field of every (abstract) object. Naturally, their signatures are more precise than ours but larger in size. Even if *SecJ* authors do not provide any measurements, we

```
class MHz<v2>{
    [int,v2] points;
    [int,H] ppoints;
  "MHz()":
    MHz: v39,() --<{};v38>--> void;{}
  "getLevel()":
    getLevel: v37,() --<{};v36>--> [int,H];{}
  "update(int)":
    update: v33,([int,H]) --<{};v32>--> void;{}
}
```

Figure 4.8: Signatures of `MHz` class produced by *SecJ*

```
class FlyMaroc<v2>{
    [int,H] discount;
  "FlyMaroc()":
    FlyMaroc: v46,() --<{};v45>--> void;{}
  "discount(MHz)":
    discount: v44,([MHz<v2>,v40]) --<{};v43>--> void;{}
}
```

Figure 4.9: Signatures of `FlyMaroc` class produced by *SecJ*

believe that the size of *SecJ* signatures is considerably bigger than our size, and hence difficult, if not impossible, to embed.

Method override To deal with overriding, *SecJ* defines a subclassing invariance requirement: overriding cannot bring new constraints other than those in the overridden method. For the moment, this requirement prevents the support for interfaces and abstract classes. In STAN, to deal with openness and inheritance, we define for each method two flow signatures: an exact flow signature, which corresponds to the exact type of the method, and a global flow signature, which corresponds to all the overriding methods in the class hierarchy. Overriding cannot bring new flows other than those in the global signature of the overridden methods. Hence, the global signature approach in STAN is more general and more flexible than the subclassing invariance in *SecJ*.

Exceptions STAN supports checked exceptions but it does not account for run-time and uncaught exceptions. *SecJ* does not support exceptions at all, while *Jif* supports all types of exceptions but the constraints are over-restrictive: all run-time exceptions must be protected by `try-catch` blocks, hence *Jif* programs contain many `try-catch` blocks with empty code. We can apply the same mechanisms for run-time exceptions in STAN (empty `try-catch` blocks); then we can say that our tool completely supports exceptions, even if their use is very restricted.

Embedded verifier In the previous paragraphs, we have compared STAN to *Jif* and *SecJ*, two powerfull tools for information flow enforcement. Due to its security policies, modularity and inference features, *SecJ* is the tool that comes closest to STAN. This is why, we concentrate our efforts on comparing STAN to *SecJ*. But the comparison refers always to STAN, the prover, and never to our verifier, VSTAN. This is simply because the other models do not address the issues of integrating the verification in a real virtual machine, as we have done in this thesis.

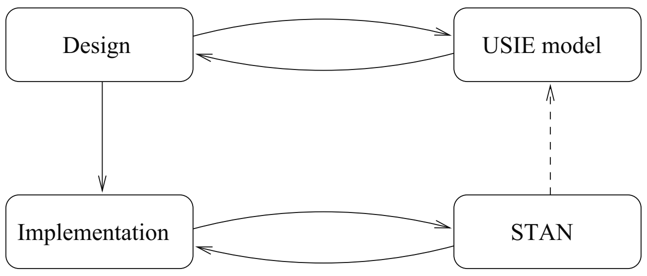


Figure 4.10: The development process

4.2 Integrated security analysis framework

Software systems that run in open environments are facing more and more attacks or intrusions. This situation has brought security concerns into the software development process. As stated in the National Institute of Standards Technology (NIST) handbook on computer security, "Security, like other aspects of a computer system, is best managed if planned throughout the computer system life cycle [NIS]". Generally, software services are expected not only to satisfy functional requirements but also to be resistant to malicious attacks. Secure design of a software service has to take into account the various situations under which the service may be misused. But secure design does not guarantee secure software. Malicious or unintentional refinement of a design into some implementation can lead to insecure software.

Addressing security issues from early design stages of the application improves software quality, but does not ensure security properties. Errors injected in the implementation can be detected only by analyzing the code ready to be executed. A safe and secure system requires security guarantees at all stages of the software lifecycle [CFH$^+$98] (*i.e.*, design, implementation, deployment, operation, and maintenance). We propose in this paper an integrated security validation and verification framework that spans the entire software lifecycle.

As illustrated by Figure 4.10, the framework combines the USIE (User-System Interaction Effect) paradigm, which is used to capture and check security events at design time, and the STAN (STatic Alias aNalyser) environment, which is used to check the program code. A USIE model captures a trace of the communication in a user-system interaction as defined by a UML interaction diagram highlighting the security events involved [LT04]. Various metrics can be derived from such model and used to analyze system security properties.

The code derived from the checked design is analyzed by STAN, our tool which implements the embedded information flow model presented in the previous chapter and briefly described in Section 4.1. STAN allows static alias analysis and checking of information flow, at deployment time. STAN certifies, at loading time, Java compiled code (bytecode). Based on STAN feedback, the developer can correct implementation errors, or, if there is a design error or the error is too complex, the designer may use the corresponding feedback in revising the design.

In this section, we illustrate how the two existing models, USIE and STAN, can be combined into a security validation and verification framework by presenting the case study of a patient medical record keeping system implemented as a smart card, and by focusing only on confidentiality properties.

We first present the medical case study and confidentiality properties to be validated; next, we define some metrics for measuring confidentiality; finally, we describe the design validation process using the USIE model, and the implementation process using STAN.

4.2.1 Medical Identification and Information Card: Case study

There is a growing interest in storing personal medical information in smart cards, because they are highly convenient from a practical perspective (*e.g.*, mobility). As the use of smart cards is growing, they are being associated with security since they provide a partial solution to the need for personal identification and non-repudiation. The self-containment and tamper-resistance of smart cards make them resistant to attacks, as they do not need to depend upon potentially vulnerable external resources. Because of this, smart cards are often used in applications, which require strong security protection and authentication.

The care card Our case study consists of a medical identification and information card, referred to as a **care card**, which maintains securely electronic medical records of a specific patient, who is the holder of the card. Emergency situations where vital health information of a patient cannot be easily obtained are often experienced. A medical identification and information card would prove to be invaluable in providing treatment in such situations.

Figure 4.11 illustrates the class diagram describing the business objects involved in the application. A patient's record ($MedicalRecord$ class in Figure 4.11) consists of two sections: prescription (the $Prescription$ class) and treatment (the $Treatment$ class). The treatment section represents the core of the medical record, in the sense that it carries the medical history of the patient. Only authorized doctors can read or modify a medical record. An authorized doctor is a registered doctor that a patient has chosen either as his family doctor or as a specialist to whom he has been referred to by his family doctor. (Authorized) nurses may only read prescriptions, but should not be allowed to access to the rest of the patient's record. The administrator is the only person who can create, delete, read and modify a patient record.

The care card stores the patient's medical record, so the information travels with the patient. Data can be read/write (only by authorized users) from/to the card at various terminals (operated by doctors, nurses, etc.).

The card contains two types of applications:

- a card manager (class $CardManager$), which handles access control rights and the medical record,
- one or many installed applications ($CardApplet$). The initial system contains two applications: a doctor application ($Doctor$) and a nurse application ($Nurse$).

The card manager also handles the list of the installed applications. Users can interact with the card via commands to applets. To satisfy commands needing information from the medical record, the applets will forward the requests to the card manager, which, based on the rights of the identified users, will provide or not the data. Each card has an administrator (*e.g.*, the family doctor) who has the right to add/remove users and rights.

Resource sharing security issues Resource sharing represents a real threat for smart cards security as the smart card technology evolves towards a "single card-multiple applications" approach [DGGJ03]. As the number of applications increase, card issuers are targeting at the same time significantly reduced time to market and lower costs, increasing the opportunity for more and more security problems. Moreover, the smart card runs applications issued by different providers, applications that may share resources or code. The insecurity may arise from the unauthorized interactions between installed applications through shared resources.

A large number of security issues inherent in concurrent software services can be viewed as negative side effects of (resource) sharing. The fact that several users may share resources poses enormous threats regarding security and privacy. It is essential to ensure that shared resources

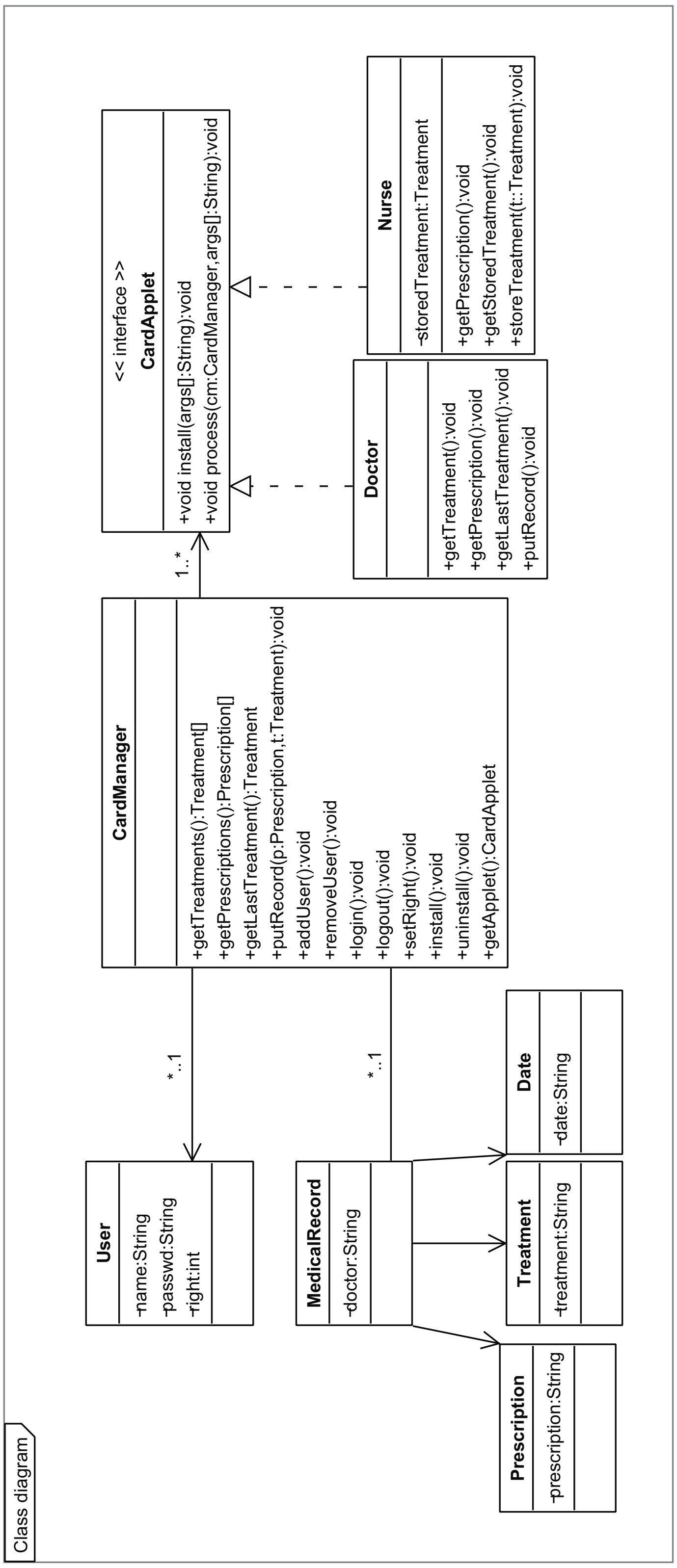


Figure 4.11: The class diagram

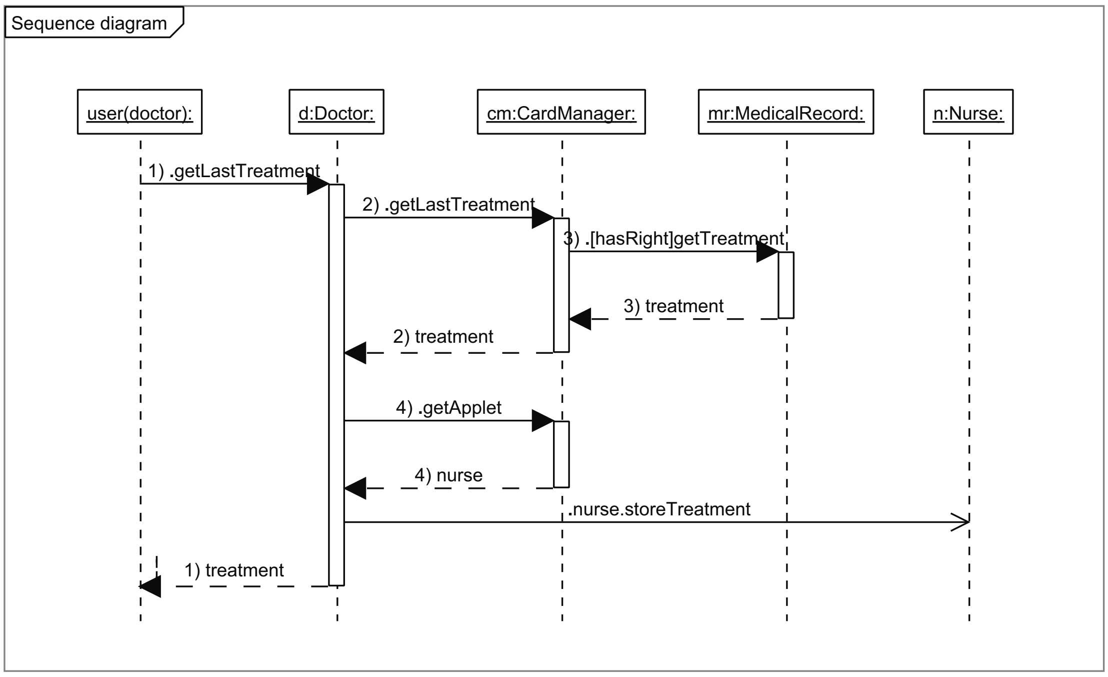


Figure 4.12: Sequence diagram 1

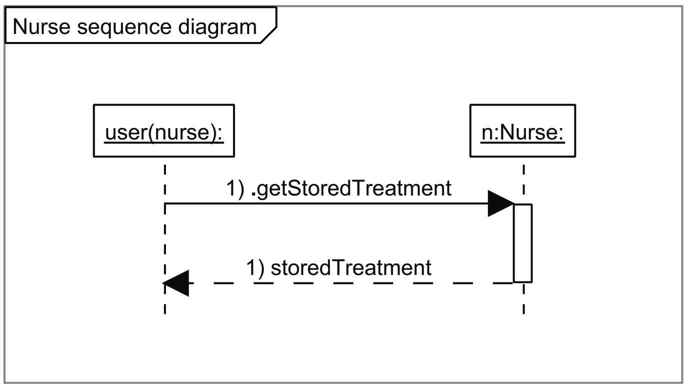


Figure 4.13: Nurse sequence diagram

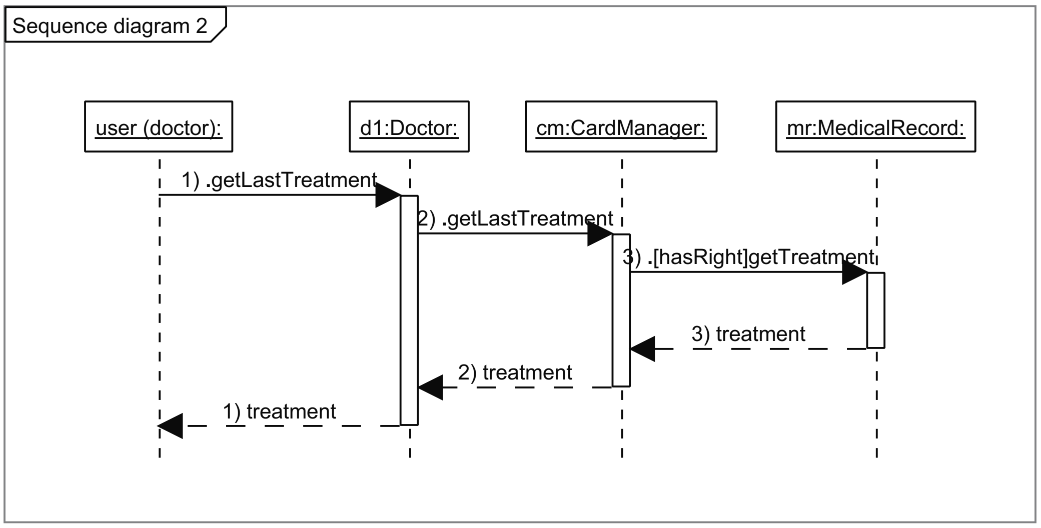


Figure 4.14: Sequence diagram 2

cannot be used as vehicles for leaking private information either directly or indirectly from a user domain to that of peers, or any other unauthorized third parties. Confidential data must be properly handled, and should not be made available to unauthorized applications.

Care card security issues Here, the medical record of the patient represents the confidential data. Not only the access to the medical record should be protected, but malicious usage, too: an authorized user should not make the data available to an unauthorized third-party. Hence, we can define the following information flow rule withit the care card:

Information flow policy within the care card: Authorized users are allowed to read/write data through the terminal connected to the card, but should not transfer the data to local storages accessible to any other users.

Confidentiality is seen as a measure of potential interference between different actions.

In our case study, insecurity arises from malicious implementations, as depicted by the sequence diagram in Figure 4.12. The diagram describes the interaction sequence involved in the method $getLastTreatment$ supported by the $Doctor$ application, method that returns the last treatment. The application requests the last treatment from the $CardManager$. Based on the rights of the requesting user, the $CardManager$ may or may not return the treatment. The result is sent to the user at the terminal. At the same time, however, a malicious manipulation of the data by invoking the method $storeTreatment$ of the class $Nurse$, which stores in a public field the treatment passed as parameter, will create an undesired information flow. An interaction of the $nurse$ with the system, as depicted in Figure 4.13, will allow nurses, which do not have the right to read the treatments, to access confidential data. An alternative design is described by the sequence diagram of Figure 4.14, which illustrates a more secure data access, as the treatment is no longer stored in a public field.

4.2.2 Mesuring confidentiality

Confidentiality is viewed as a function of the interference occurring among interactions between different users with the system. So possible measure of confidentiality may consist of evaluating this interference. This could be conducted by evaluating the existence of potential information leakage channels resulting from interference occurring among interactions between different users with the system.

In [LT04], authors define a measure of confidentiality of a user-system interaction with respect to another user-system interaction in terms of the number of significant information leakage channels existing potentially between them. If I_A represents an interaction of a user A with the system, and I_B an interaction of a user B with the system, we can define the confidentiality as

$Conf(I_A \rightarrow I_B)$, or the confidentiality of I_A with respect to I_B, is a measure of how much B can discover about I_A via I_B

This measure of confidentiality can be defined as:

$$Conf(I_A \rightarrow I_B) = \frac{1}{1 + NOSILC(I_A, I_B)} \tag{4.2.1}$$

Where "NOSILC(IA,IB)" represents the "Number Of Significant Information Leakage Channels" from I_A to I_B.

Using this formula, we validate a system security both at design time (based on USIE model) and at implementation time (based on the flow signatures of methods). In the next sections, we show how the "Number Of Significant Information Leakage Channels" can be measured at both stages of the software lifecycle (design and implementation).

4.2.3 Design validation

UML diagrams and security UML interaction diagrams can be used to describe collaborations of a software application. An interaction diagram corresponds to a sequence diagram or communication diagram, which are interchangeable because of their duality. However, security analysis is not the primary concern of standard UML interaction diagrams. Security related events could not be described satisfactorily using the basic or regular semantics of these diagrams. Moreover, the modeling features provided by the UML interaction diagrams make it rather difficult to conduct directly security analysis. For instance, the understanding of user interactions, the data and role entities involved in a collaboration play important roles in the analysis of security events. But regular UML sequence or collaboration diagrams provide only the communication information for an interaction while the responses of role entities are not expressed.

In order to highlight security events and facilitate security analysis, the USIE model is used to abstract collaborations. The primary design rationale of USIE paradigm is to provide convenient semantics for software security analysis at the architecture level. By augmenting a UML model with some predefined stereotypes expressing security events, corresponding USIE models can be derived automatically from UML interaction diagrams.

The USIE model A USIE model consists of a directed graph involving two kinds of nodes named *InteractionStart* and *RoleEntity*. An *InteractionStart* node represents the starting point of a collaboration, and is in principle named after the collaboration name (*i.e.*,, the name of the UML interaction diagram). A USIE graph involves a single *InteractionStart* node. In contrast, a USIE graph can have multiple *RoleEntity* nodes. A *RoleEntity* node corresponds to a role entity contained in a UML interaction diagram; it is named after corresponding role entity name and it can contain optional security related characteristics defined by dedicated stereotypes in the corresponding interaction diagram.

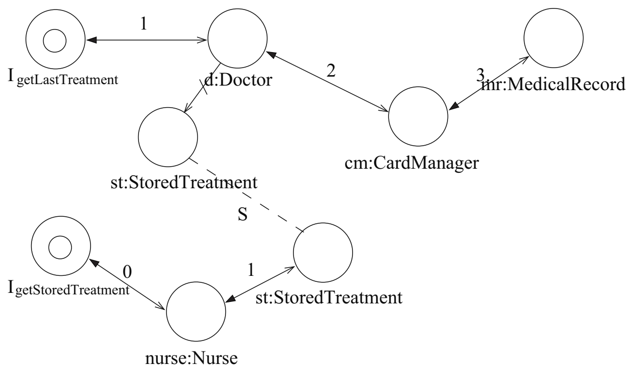


Figure 4.15: USIE diagram derived from Figure 4.12

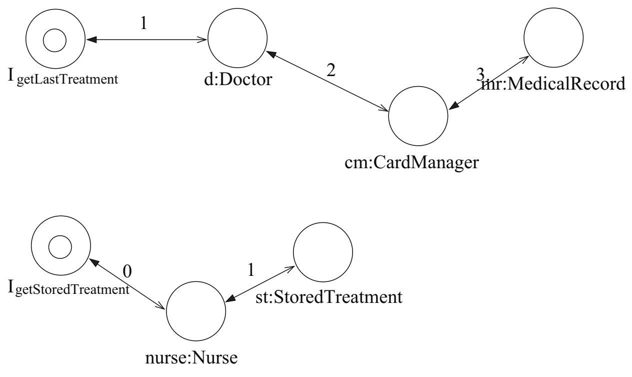


Figure 4.16: USIE diagram derived from Figure 4.14

In a USIE graph, an arrow linking a pair of nodes represents a single communication (sending an event or invoking a method) between two role entities. Further characteristics of such communication link are expressed by associating with the arrow the communication order and some optional attributes. There are two kinds of attributes named *ChangeState* and *ReturnInformation*. An edge has *ChangeState* attribute if the communication modifies the state of its target role entity; it has *ReturnInformation* attribute if the communication returns information to its source role entity. An edge can have zero or more attributes. The edge attributes, *ChangeState* and *ReturnInformation*, are expressed using the communication stereotypes of the interaction diagram semantics.

The USIE model for the Care Card We model the two designs of the $getLastTreatement$ service and the malicious service using USIE notations. Figure 4.15 shows the USIE models derived from sequence diagram 4.12; Figure 4.16 shows the USIE models derived from sequence diagram 4.14. In a USIE graph, *InteractionStart* nodes are modeled using double circles while *RoleEntity* nodes are represented using single circles. Simple arrows represent communications without any specified attribute; simple arrows crossed by a small bar depict communications with *ChangeState* attribute; double-edges arrows depict communications with *ReturnInformation* attribute. The interested reader is referred to [LT04] for more details about the graphical notations of USIE models.

Interference mesurements Based on the USIE models derived from the interaction diagrams, we can measure confidentiality according to our definition in Section 4.2.2.

An information leakage channel exists between two USIE models, whenever they share a common

Metrics	$Conf(I_{getLastTreament} \rightarrow I_{getStoredTreament})$
Sequence diagram 4.12	0.5
Sequence diagram 4.14	1

Table 4.1: Metrics values for confidentiality

RoleEntity node, which is targeted in one model by a *ChangeState* edge, and in the other model by a *ReturnInformation* edge. Information leakage channels can be categorized according to their importance. An information leakage channel between interactions is significant if the information leaked by this channel can be sent to the *InteractionStart* node, otherwise the channel is secondary. A dashed line linking the nodes represents an information leakage channel between two nodes. A label "S" is used to indicate significant leakage channels, as illustrated by Figure 4.15.

Based on formula in Equation 4.2.1, we can assess and compare the two designs of the $getLastTreatement$ service of the medical record system with respect to a software service that may be potentially misused. Table 4.1 summarizes the metrics values calculated for both designs. According to these results, the design based on the sequence diagram in Figure 4.14 is less vulnerable to confidentiality breach than the design based on the sequence diagram in Figure 4.12.

4.2.4 Software validation

Data propagation through software units interactions

In the previous section we showed how security leaks can be detected and avoided during design stage. Still, the code derived from the checked design is not certified, as security problems can arise from human errors or malicious intentions during the implementation phase. We now show how the Java mobile code implementing the functional requirements of the previously validated design is statically certified by STAN, our tool implementing the information flow model presented in the previous chapter. For more details on STAN, please refer to Section 4.1 at page 76 and to Chapter 3 for more details on the model that STAN implements.

The validation target in our case study is data confidentiality required in handling shared resources, as secret or confidential data should not be made available through public means. As stated in Section 4.2.2, confidentiality is viewed as a function of the interference occurring among interactions between different users with the system. Because applications collaborate in order to offer better services and because of limitations of small devices such as smart cards, applications issued from different providers must share resources and code (*e.g.*, API) and interact through it. The medical identification card security property that we verify is concerned with secure interaction between applications running on the card. Other security functions, like access control, are handled by the card manager and are part of the computing trusting base.

A way to compute confidentiality for insecure interactions that may lead to leaking confidential data through shared resources is to detect all the information flows from confidential storage to shared resources. We remind that our algorithm computes for each method a flow signature that carries the flows, potentially generated by the execution of the method, between elements that survive the method execution.

Measuring confidentiality

Confidentiality is strongly related to non-interference, which requires that confidential data should not interfer with public data. According to our definition of non-interference, a method is secure if its flow signature, *i.e.*, S_m, does not contain flows of type $a^p \rightarrow b^s$, except flows allowed by

```
class CardManager {
  MedicalRecord mr; //secret field
  CardApplet[] applets;
  ...
}
...
```

Figure 4.17: Source code

```
class Doctor extends CardApplet {
   ...
   public void getLastTreatment(CardManager manager, String []params){
      Treatment t = manager.getLastTreatments();
      Nurse n = (Nurse)manager.getApplet("nurse");
      n.storeTreatment(t);
      Printer.streamPrintln(t.toString());
   }
}

class Nurse extends CardApplet {
   ...
   public Treatmenet channel;
   public void storeTreatment(Treatment t) {
      channel = t;
   }
}
```

Figure 4.18: Code derived from sequence diagram in Figure 4.12

declassification. Let us denote by Dcl the set of allowed flows [1]. In other words an *insecure flow* ϑ is a flow of type $\vartheta = a^p \to b^s$ such that $\vartheta \notin Dcl$. Hence, we can define the number of significant leakage channels (NOSILC) as the number of insecure flows.

Defining the security policy

The STAN user must specify a policy, by labeling the confidential fields with security level s, for secret, confidential data before the execution of the algorithm. Because Java applications not concerned with propagation policies must run normally in our environment, the default policy declares all fields with the $public$ security level. Our care card application, which must protect the propagation of the secret expressed by the $MedicalRecord$, accessible through the $CardManager$, labels the $MedicalRecord$ field in $CardManager$ with security level s (Figure 4.17[2]).

The $getLastTreatment$ method must sent to the terminal (an I/O) a part of the medical file stored in the card manager. As the medical file is secret, this operation naturally generates a flow $IO^p \xrightarrow{\mathbf{r}} p_1^s$, where IO is our abstraction for any I/O storage channel, while p^1 denotes the first parameter of the method, the $CardManager$. We remind that we perform a field-insensitive but security-level sensitive analysis (as discussed in Section 3.2.1), hence all fields of an object having the same security level are treated as a single location; in this example, p_1^s denotes all secret fields of the $CardManager$ and hence, the medical record. Then, it is obvious that our declassification set contains this flow:

$$Dcl = \{IO^p \xrightarrow{\mathbf{r}} p_1^s\}.$$

[1] This set contains flows between the return of the method to secret parts, but also other flows defined by the user.
[2] STAN works on Java bytecode but, for simplification, we will present the algorithm based on the source code.

```
class Doctor {
   ...
   public void getLastTreatment(CardManager manager, String []params){
      Treatment t = manager.getLastTreatments();
      Printer.streamPrintln(t.toString());
   }
}
```

Figure 4.19: Code derived from sequence diagram in Figure 4.14

Interference measurements

Implementing the interaction in Figure 4.12 Analyzing the method $getLastTreatment$ of the class $Doctor$ in Figure 4.18, corresponding to the interaction in Figure 4.12, leads to two possible flows of information:

- a flow $IO^p \xrightarrow{\mathbf{r}} p_1^s$ from the secret part of the $CardManager$ to the I/O (as described above), flow generated by the method $streamPrintln$ which sends the parameter to an output channel,

- a flow $p_1^p \xrightarrow{\mathbf{r}} p_1^s$ from the secret part of the $CardManager$, p_1^s (containing the $MedicalRecord$ which contains the $Treatment$) to the public part (denoted by p_1^p) of the $CardManager$ (containing the $Nurse$ applet). This flow is generated by the invocation of the method $storeTreatment$ which stores the parameter p_1 in a public field of the $Nurse$: $S_{storeTreatment} = \{p_0^{p,s} \xrightarrow{\mathbf{r}} p_1^p\}$. Note that the flow signature of $storeTreatment$ is context insensitive; the exact security level of its parameter is taken into account when the method is invoked.

The flow signature of the method $getLastTreatment$, denoted by $S_{getLastTreatment}$, is:

$$S_{getLastTreatment} = \{p_1^p \xrightarrow{\mathbf{r}} p_1^s, IO^p \xrightarrow{\mathbf{r}} p_1^s\}.$$

The flow $p_1^p \xrightarrow{\mathbf{r}} p_1^s$ is *insecure* in respect to the $getStoredTreatment$ method declared in class $Nurse$, which is a field of p_1 (the $CardManager$). The $getStoredTreatment$ method has the flow signature: $S_{getStoredTreatment} = \{IO^p \xrightarrow{\mathbf{r}} p_0^{p,s}\}$. This is explained by the fact that the fields of the $Nurse$ serve as temporary shared resources to store confidential data, which can readily be made available later to unauthorized users, creating the potential for a significant leakage channel. In contrast, the flow $IO^p \xrightarrow{\mathbf{r}} p_1^s$ is secure as it belongs to the set *Dcl* of declassified flows.

Hence, there is one insecure flow and one significant information leakage channel:

$$NOSILC(I_{getLastTreament} \rightarrow I_{getStoredTreament}) = 1$$

and, according to the formula in Equation 4.2.1, the confidentiality is measured as:

$$Conf(I_{getLastTreament} \rightarrow I_{getStoredTreament}) = 0.5$$

Implementing the interaction in Figure 4.14 The code derived from the second design of the $getLastTreatment$ method (see Figure 4.14) is depicted by Figure 4.19. The corresponding signature is

$$S_{getLastTreatment} = \{IO^p \xrightarrow{\mathbf{r}} p_1^s\}.$$

This method is secure with respect to $getStoreTreatment$, since there is no insecure flow and no leakage channel:

$$NOSILC(I_{getLastTreament} \rightarrow I_{getStoredTreament}) = 0$$
$$Conf(I_{getLastTreament} \rightarrow I_{getStoredTreament}) = 1.$$

Based on formula (4.2.1) defining a measure of confidentiality, we can assess and compare the implementations corresponding to the two designs of the $getLastTreatment$ service of the medical record system, and obtain the same results as in Table 4.1.

The results show that the bytecode analysis is equivalent with the verification of the USIE models derived from the sequence diagrams. The advantage of a two-steps integrated verification and validation is that we can detect, at implementation level, interactions not expressed by sequence diagrams. These critical points of the software should be transposed in sequence diagrams and presented to the designer, in order to address the security weaknesses and restart the validation process.

Discussion Increasing attacks and intrusions require secure software systems. Security must be taken into consideration from early stages of software design to the deployment and production use of the application. In this section we propose an integrated security and validation framework for building secure applications. The framework combines the USIE (User-System Interaction Effect) paradigm for checking security at design time and the STAN(STatic Alias aNalyser) tool for checking the program code.

To illustrate our approach, we present the case study of a medical identification and information card, which maintains medical records of the holder of the card. In this case study, we focus on confidentiality properties and security breaches arising from users interactions with the system.

In the future, we plan to extend our approach to other security properties like integrity and availability and to support more complex policies. In the long term, it would be extremely interesting to use the integrated framework in a reverse engineering approach to point at design level security leaks detected in the code.

4.3 Information flow in MIDlets: The CareCard MIDlet suite

MIDlets are a particular case of Java applications, hence our tool cannot be applied directly. Nevertheless, as we will show in this section, only very few adjustements are required to adapt our tools to MIDlets, and these adjustements can be done by any user.

Therefore, in this section we describe how our analysis can be applied to MIDlets. We first start with a short introductions to MIDlets and to the Java ME security model. We then describe an implementation of the medical application, that we have aready seen before, using MIDlets, and finally we show how information flow security for the medical application is enforced using STAN.

4.3.1 Preliminaries

Java ME

Java 2 Micro Edition (Java ME) is a set of technologies and specifications developed for small devices like pagers, mobile phones, and PDAs. Java ME uses subsets of Java Standard Edition (Java SE) components, such as smaller virtual machines and leaner APIs.

Java ME does not define a new programming language. Rather, it adapts existing Java technology to handheld and embedded devices. Java ME maintains compatibility with Java SE wherever feasible. To address the stricter limitations of devices, Java ME sometimes replaces Java SE APIs and adds new interfaces. But the evolution of Java ME is as much about omitting unnecessary parts of the Java SE and Java Enterprise Edition (Java EE) platforms as it is about overriding and adding new ones.

Java ME, therefore, is divided into configurations, profiles, and optional packages. Configurations are specifications that detail a virtual machine and a base set of APIs that can be used with a certain class of device. A configuration, for example, might be designed for devices that have less than 512 KB of memory and an intermittent network connection.

The Connected Limited Device Configuration (CLDC) is a minimal Java ME configuration for devices with stringent restrictions on computational power, battery life, memory, and network bandwidth. These limits directly affect the kinds of Java technology-based applications they can support. CLDC does not require a lot of resources. It supports devices with 16-bit or 32-bit processors with at least 160 KB of persistent memory and at least 32 KB of volatile memory, for a total of 192 KB. Power consumption is low, and devices are typically battery-powered. At the heart of this configuration is a Java virtual machine with some Java SE capabilities removed. For example, CLDC does not support class finalization or thread groups.

Above the configuration are profiles and the application environment, which consist of the APIs and application environment such as the Mobile Information Device Profile (MIDP) and the Java Technologies for Wireless Industry.

Of the profiles designed for CLDC, MIDP is the most prevalent. As the first Java ME profile, MIDP [KV04] is the most mature and widely adopted, with millions of deployments all around the world, primarily on PDAs, cell phones, and other handheld communicators.

MIDlets

A MIDlet[3] is a Mobile Information Device Profile (MIDP) application. Like an applet, a MIDlet is a managed application. Instead of being managed by a web browser, however, it is managed by special-purpose application-management software (AMS) built into the device. External management of the application makes sense in this context because it may be interrupted at any point by outside events. It would be poor behavior, for example, for a running application to prevent the user from answering incoming phone calls while he/she is playing a game.

MIDlets security MIDlets run in a sandbox. The memory requirements of the Java SE security classes alone exceed the total memory budget available to Java ME CLDC/MIDP. Hence the Java ME CLDC specification dictates a much simpler ”sandbox” security model. A MIDP application (MIDlet) therefore runs in a closed environment and is only able to a access a predefined set of classes and libraries supported by the device.

A CLDC Virtual Machine has a built-in class loader. This class loader can however only load classes from the predefined set of system classes or the application (MIDlet) JAR file. To avoid this restriction being bypassed, unlike a Java SE Virtual Machine, the class loader cannot be replaced, overridden or reconfigured by the user. Similarly there is no support for loading native libraries.

In short it is not possible for the user/developer to define his own class loader and to extend the range of libraries (APIs) supported by a CLDC/MIDP phone beyond the pre-defined set that was built into the device. The application (MIDlet) may make use of third-party pure Java CLDC/MIDP libraries such as kSOAP or kXML by incorporating them into the application JAR file. Obviously such third-party libraries must be written entirely in Java (no native code) using the CLDC/MIDP API.

[3] `http://java.sun.com/javame/reference/apis/jsr118/`

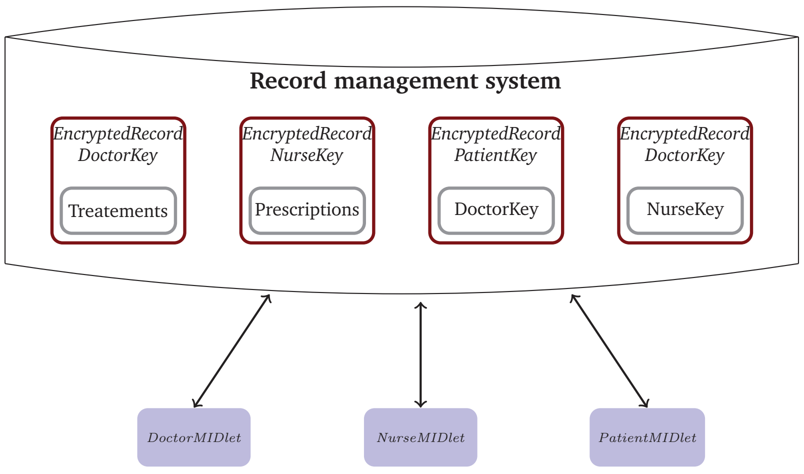


Figure 4.20: CareCard MIDlets Suite

MIDlets communication through persistent storage All these restrictions make impossible the communication thought shared objects between MIDlets. The only way MIDlets can exchange data is through the Record Management System (RMS), which offers persistent storage capabilities: MIDlets packaged in the same MIDlet suite can access to the same record.

The MIDP provides a mechanism for MIDlets to persistently store data and retrieve it later. This mechanism is a simple record-oriented database called the Record Management System (RMS). A MIDP record store (or database) consists of a collection of records that remain persistent after the MIDlet exits. When a MIDlet is invoked again, it can retrieve data from the persistent record store.

Record stores (binary files) are platform-dependent because they are created in platform-dependent locations. MIDlets within a single application (a MIDlet suite) can create multiple record stores (database files) with different names. The RMS APIs provide the following functionality: *(1)* allow MIDlets to manipulate (add and remove) records within a record store; *(2)* allow MIDlets in the same application to share records (access one anothers record store directly); *(3)* do not provide a mechanism for sharing records between MIDlets in different applications.

4.3.2 Case study: The CareCard MIDlet suite

Having described Java ME and MIDlets specifications and requirements, we now show how the medical CareCard, already described in Section 4.2.1 at page 84, is implemented on Java ME using MIDlets. We will describe how illegal information flow can occur in such systems (intuitively, through persistent records), and how it can be verified using our tool, STAN. The implementation of the MIDlet suite for CareCard and the tests using STAN have been done by an engineer[4] who had no knowledge about information flow problems or about STAN before this experience.

Let us briefly recall the medical card requirements. The medical card is based on the BMA policy, developed by Anderson in response to a growing demand for the protection of personal health information in several countries [And96]. Due to space limitations, we only focus on information flow items in the policy. The medical card is a card that can be used in hospitals to have access to information about a patient. According to the BMA security policy, the medical file of a patient has two parts: prescriptions and treatments. There are three types of users for this card : the doctor, the nurse and the patient. The doctor can read and write both treatments and prescriptions record,

[4]We thank Antoine Neveu for his development work and useful comments.

and he can also read patient private information; the nurse can only read patient information and prescriptions.

The goal of this example is to show that a nurse cannot have access to treatments of a patient, and to prove that a doctor or a patient cannot give to a nurse the access to an information she is not supposed to know.

Implementation

Each actor of the system (patient, doctor and nurse) has his own MIDlet. Each MIDlet implements functions associated with each actor according to the security policy (*i.e.*, the Nurse MIDlet gives access only to prescriptions and patient information).

A first security enforcement is access control via encryption keys. Each actor can login into the system using his own key (the DoctorKey, NurseKey and PatientKey).

As we have explained above, the only way to store and to share information with MIDlets is to use persistent storage, *i.e.*, record store. Hence, treatments and prescriptions are stored in such record stores, which can be accessed by all MIDlets in the medical MIDlet suite. But, according to the BMA security policy, treatments must be accessed only by doctors and the patient, and not by nurses, while prescriptions can be accessed by all actors. To implement this security requirements, we define encrypted record stores, implemented in class `EncryptRecord`, in which each information is encrypted with a key. Hence, the treatments record and prescriptions record are encrypted records and they use different keys, as depicted in Figure 4.20:

- the treatments record is encrypted with the doctor key,
- the prescriptions record is encrypted with the nurse key.

With this system, the nurse can access the prescriptions record. The doctor has access to the treatment record; moreover, to have access to the prescriptions record, it must know the nurse key: hence, the nurse key is stored in a record encrypted with the doctor key. The patient must have access to both prescriptions and treatments; hence, the doctor key is stored in a record encrypted with the patient key (Figure 4.20).

The encryption keys are initialized when the actors first use their specific application. The only constraint is the order in which the keys must be initialized: first, the patient key (needed to encrypt the doctor key), then the doctor key (needed to encrypt the nurse key) and finally the nurse key.

Illegal information flow example

Apart encrypted record stores, there are also unencrypted records, implemented by the class `PublicRecord`. A "public" record is shared between all MIDlets, hence data stored in public records are not protected by any encryption mechanism and they can be accessed by all actors.

As MIDlets cannot interact with other MIDlets, the only way to share information is through shared record stores. For example, an illegal use of information can occur if a malicious DoctorMIDlet reads the treatments and stores them in a public record. The illegal scenario is depicted in Figure 4.21 and consists of three steps:

1. the doctor reads the treatment from the encrypted record and decrypts it using his key,
2. the doctor writes the treatment to a public record,
3. the nurse reads the treatment from the public record.

This example shows that the nurse can access to treatments despite the fact that this is forbidden by the security policy. The only way for a nurse to read treatments is that someone who has the

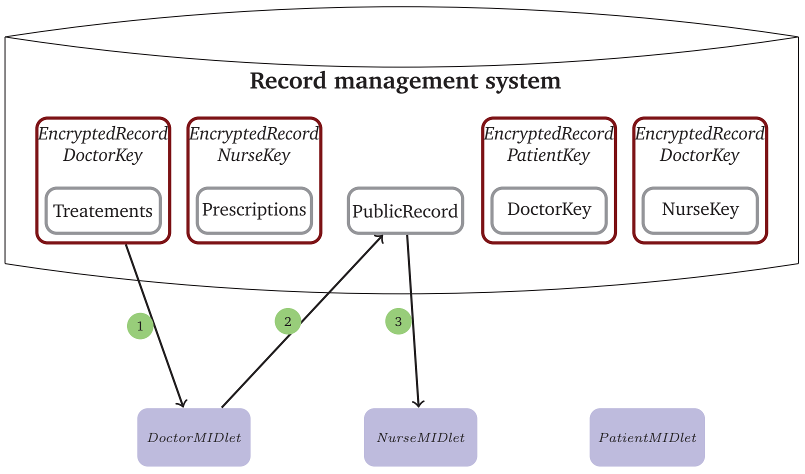


Figure 4.21: Illegal information flow in the CareCard MIDlets Suite

```
1   public class Doctor extends MIDlet implements CommandListener {
2    private static final Command cmdTreat=new Command("LT",Command.SCREEN,1);
3    private SecretRecord key;
4    [..]
5    public void commandAction(Command c, Displayable d) {
6      [..]
7      if (c == cmdTreat) {
8        String nurseKey = cm.getNurseKey(doctorKey);
9        Treatment[] t = cm.getTreatments(doctorKey, nurseKey);
10       Form listTreat = new Form("Treatments List");
11       try {
12         PublicRecord rs = new PublicRecord("test",true,true);
13         for (int i = 0; i < t.length; i++) {
14           String treat = t[i].toString();
15           listTreat.append("Treatment: "+treat+"\n");
16           byte[] b = t[i].toByteArray();
17           rs.addRecord(b, 0, b.length);
18         }
19       } catch (RecordStoreException e) {e.printStackTrace();}
20       [..]
21       display.setCurrent(listTreat);
22     }
23   }
24 }
```

Figure 4.22: Excerpt from `Doctor` MIDlet showing an illegal flow

right to read them to put them into a public record store, visible to everybody. Such illegal flow cannot be detected and prevented by the control access mechanisms; therefore, the information flow analysis is crucial. The goal of an information flow analysis for Java MIDlets is to detect if secret data (*i.e.*, read from an encrypted record) are stored in a public record.

The code in Figure 4.22 shows a part of the `commandAction` method in the class `Doctor` which reads and decrypts treatments (line `9`) and then stores them in a `PublicRecord`. The public record is created at line `12` while the treatments are stored in the public record at line `17`.

$$
\begin{array}{llcl}
EncryptedRecord: & S_{addRecord} & = & \{IO^s \xrightarrow{\mathbf{r}} p_1^{s,p}\} \\
EncryptedRecord: & S_{getRecord} & = & \{R^{s,p} \xrightarrow{\mathbf{r}} IO^s\} \\
PulicRecord: & S_{addRecord} & = & \{IO^p \xrightarrow{\mathbf{r}} p_1^{p,s}\} \\
PublicRecord: & S_{getRecord} & = & \{R^{p,s} \xrightarrow{\mathbf{r}} IO^p\}
\end{array}
$$

Figure 4.23: Manual flow signatures for MIDlets record store

Verifying information flow using STAN

We now show how the information flow is verified in the CareCard MIDlet suite using STAN. We first describe the security policy we define, then we describe how the tool runs and detects illegal flows. We put an accent on the slightly different information flow security model between standard Java applications and MIDlets, and we detail the adjustements that have been made in order to adapt our tool to the special requirements of MIDlets. We will see that these adaptations only consist of adding some flow signatures by hand.

Running STAN on an application consists of the following steps:

1. specifying security levels (*i.e.*, label fields that containt sensitive data with security level s),
2. adding declassification (*i.e.*, hand written flow signatures which declassify information flow in a method),
3. running STAN and computing flow signatures,
4. specifying the security policy which describes collaborations between applications using the Domain Specific Language,
5. verifying the security policy w.r.t. flow signatures.

Step 1: Secret fields In the first step, we must identify the "secret" fields, *i.e.*, fields which will store confidential data. This is required because in "normal" Java applications sensitive data is stored in class fields, but in the CareCard MIDlet suite, sensitive data is stored in an encrypted record store and not in class fields. Hence, in this case, there is no need to specify secret fields: all fields are public, by default.

Step 2: Declassification through manual flow signatures As discussed above, secret data is stored in encrypted records. Only authorized users can access the secret data. Writing and reading data to/from the encrypted record store is done using the `addRecord` and `getRecord` functions. As these functions perform encryption/decryption and in the same time write/read to a file stream (using native methods), we choose to add manually written flow signatures, as depicted in Figure 4.23.

Hence, the flow signature of `addRecord` contains a flow $IO^s \xrightarrow{\mathbf{r}} p_1^{s,p}$, meaning that the first parameter of the method is written to a "secret" record store. The flow signature of `putRecord` contains a flow $R^{s,p} \xrightarrow{\mathbf{r}} IO^s$ meaning that the method returns a value read from a "secret" record store. We remind you that the abstract value IO stands from input/output files.

In the same way, we manually add flow signatures for `addRecord` and `getRecord` methods in the `PublicRecord` class. The flow signatures are also presented in Figure 4.23. The difference between `PublicRecord` and `SecretRecord` is that `PublicRecord` reads from/writes to a "public" record store.

$$\boxed{\begin{array}{llll} \mathit{Doctor}: & S_{commandAction} & = & \{\mathit{Static}^p \xrightarrow{\mathtt{i}} \mathit{Static}^p, \mathit{Static}^p \xrightarrow{\mathtt{i}} IO^s, \mathit{Static}^p \xrightarrow{\mathtt{i}} p_0^{p,s} \\ & & & IO^{p,s} \xrightarrow{\mathtt{i}} \mathit{Static}^p, IO^p \xrightarrow{\mathtt{r}} IO^s, IO^{p,s} \xrightarrow{\mathtt{i}} p_0^{p,s} \\ & & & p_0^p \xrightarrow{\mathtt{i}} \mathit{Static}^p, p_0^p \xrightarrow{\mathtt{i}} IO^s, p_0^p \xrightarrow{\mathtt{r}} p_0^p\} \end{array}}$$

Figure 4.24: Computed flow signatures for MIDlets record store

Manually adding flow signatures for these classes implies that their implementation belongs to our Trusting Computing Base. In the same time, this allows to "label" encrypted record stores with the security level s.

Step 3: Computing flow signatures The next step in verifying information flow security is to compute the flow signatures. We only concentrate on methods which present a special interest for our case study in order to show the illegal information flow, hence on method `commandAction` in the class `Doctor`. The implementation of this method has already been presented in Figure 4.22.

The complete flow signature of `commandAction` is shown in Figure 4.24. You can notice that the flow signature contains many information flows. This is due to the fact that the method is complex, it deals with many variables (the IO, the display - which is a static variable, the encryption keys - which are fields of the `Doctor` class). We do not detail all flows here, we will only show later how this flow signature does not respect the desired security policy.

Step 4: Specifying the security policy As we explained before, the only way in which MIDlet can collaborate and exchange information is through record stores. Secret data are kept in an encrypted record which can be accessed only by authorized MIDlets. If an authorized MIDlet wants to give the sensitive data to an unauthorized user, it must write it in a public record, accessible to all MIDlets. Hence the illegal information flows are flows from a secret record (*i.e.*, IO^s) to a public record (*i.e.*, IO^p). All the other flows inside a MIDlet must be allowed, as the MIDlet runs in an isolated sandbox.

Our DSL for specifying collaboration policies allows specifications in the opposite way: we suppose that all flows are forbidden, and we define allowed flows explicitly. Hence, to be able to apply our DSL in the context of MIDlets, we had to slightly extend it: a policy must always start with defining a type, *i.e.*, *positive* or *negative*. A *negative* policy is the one presented in the previous chapter: everything is denied except explicit flows. A *positive* policy is the one needed in the case of MIDlets: everything is allowed except explicit flows. In the case of *positive* policies, we must specify forbidden flows, *i.e.*, **IO^s does not shares with IO^p**.

Defining new types of security policies does not require changing the analysis, as each security policy is translated in a class extending a predefined class `Report` which offers methods for accepting/rejecting flows.

Step 5: Verifying the security policy The security policy verification consists of inspecting all flows signatures of methods in the MIDlet suite in order to detect flows forbidden by the policy, *i.e.*, flows of type $IO^p \xrightarrow{\mathtt{r,v,i}} IO^s$. The policy certification fails while analysing the flow signature of `Doctor.commandAction` (depicted in Figure 4.24) as it contains a forbidden flow: $IO^p \xrightarrow{\mathtt{r}} IO^s$.

4.4 Conclusion

In this chapter, we tackle the practical side of our information flow model. Experiments conducted to evaluate the model show that it can be successfully applied to small systems and that our simplifying assumptions are reasonable in practice. Moreover, we have shown that previous models cannot be successfully applied on the target devices.

To prove the ease of use, we have integrated our model in a security verification framework which ensures software verification and validation from the early stage of the development cycle.

Then, we have presented a case study based on J2ME MIDlets which are different from normal Java applications: sensitive data is stored in record stores and not in class fields. We have seen that adapting our analysis on different targets does not require many efforts: we had to add only a few manual flow signatures (which belong to our TCB) and to specify a security policy. The only difficulty is the onboard verification, as the J2ME relies on a built-in class loader. The integration requires little effort, but the major challenge is to convince J2ME platform providers to integrate the information flow verifier in the OS of their devices.

5 A sound dependency analysis using abstract memory graphs

Contents

The information flow model and the extensions presented in the previous chapters have been designed aiming at a practical framework, which addresses real security problems. The goal was to have a practical model which can also be formally proven sound. To prove the soundness of the embedded analysis, we defined a more general analysis, which does not need to aproximate the abstract model anymore and which can be applied in a wider context. The new analysis keeps the main features of the previous one: flow-sensitivity and compositionality. The main difference, motivated by correctness considerations, consists of representing the flows by means of abstract memory graphs, which are an abstract representation of objects in the memory enriched with information regarding fields of primitive values and dependencies induced by the control flow

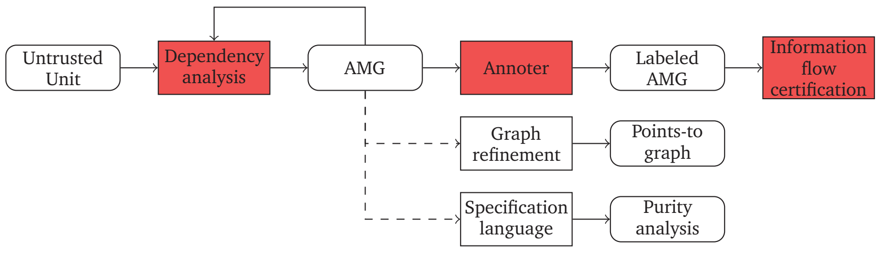


Figure 5.1: Dependency analysis schema for secure information flow

of the program. In order to adjust to the constraints of small open systems, the previous model approximates fields of objects and performs an object-sensitive, field-insensitive but security level-sensitive analysis. The AMG[1], the purpose of which is to be a general analysis framework, provides more precise results, as it is the result of a field-sensitive and object-sensitive analysis. The flow signature of a method is an approximated representation of the AMG of a method.

Our approach consists of computing, at different program points, an AMG, which tracks how input values of a method may influence its outputs. This computation subsumes a points-to analysis (reflecting how objects depend on each other) by extending it with dependencies arising from data of primitive types and from the control flow of the program. In contrast to many type-based information flow techniques, our approach does not require security levels to be known during the computation of the graph: security aspects of information flow are checked by labeling the AMG with security levels *a posteriori*.

In this chapter we present the construction of the AMG, while in the next chapter, our graph construction is proven sound by establishing a non-interference theorem. The theorem states that if an output value is unrelated to an input value in the AMG, then the output remains unchanged when the input is modified.

5.1 Preliminaries

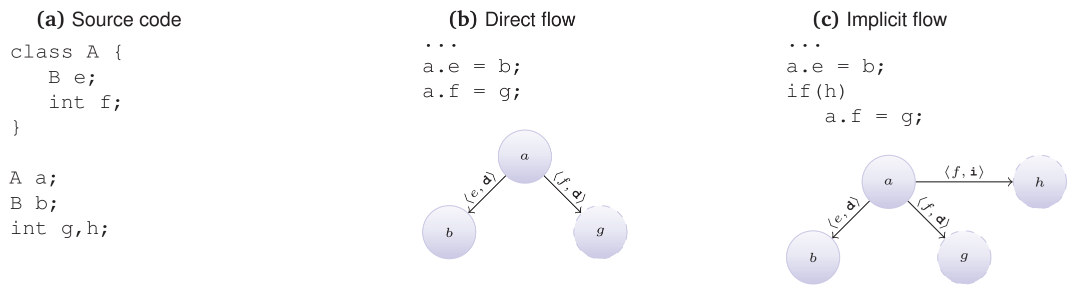
5.1.1 General presentation

In this chapter we present a general analysis for Java bytecode, computing an AMG including references and primitive types. The AMG is computed without any knowledge about information security. Security levels are applied *a posteriori*, by an annoter, as depicted in Figure 5.1. The result is a labeled AMG on which different security properties, such as non-interference, can be certified. The AMG can be successfully used for other program analysis applications such as points-to analysis, purity analysis, etc.

AMG description The AMG is a points-to graph extended with primitive values and dependencies raised by control flow. Nodes of the graph are objects and fields of objects, while edges represent names of fields and the type of flow (direct or implicit). On the one hand, the graph characterizes how fields of objects point to other objects. On the other hand, it describes the dependencies between primitive values through direct or indirect flow, in the sense of non-interference.

Figure 5.2 shows some examples[2] and the corresponding AMGs. The instruction `a.e = b` in Figure 5.2b creates an edge from a to b, labeled with $\langle e, \mathbf{d} \rangle$, meaning that the fields e of a points to

[1]Recall that, throughout the remainder of this thesis, AMG abbreviates “abstract memory graph”.
[2]Our analysis works on JVM bytecode, but for simplification, we will show some examples at source level.



(a) Source code

```
class A {
   B e;
   int f;
}

A a;
B b;
int g,h;
```

(b) Direct flow

```
...
a.e = b;
a.f = g;
```

(c) Implicit flow

```
...
a.e = b;
if(h)
   a.f = g;
```

Figure 5.2: AMG example

the object b. The second element of the label (**d**) indicates that the edge originates from a direct assignment. This edge is created by a normal points-to analysis, as it indicates to which object the field e of a might point to in the memory.

In addition to the points-to graphs, we also consider nodes and assignments of primitive type. For easier reading, nodes with dashed surroundings (*e.g.*, g) depict primitive values. The instruction `a.f = g` creates an edge from a to g, labeled with $\langle f, \mathbf{d} \rangle$.

Flows arising from the control structure of the program are identified by the second element in the label of an edge: **i**. The code `if(h) a.f = g` in Figure 5.2c creates two edges:

1. an edge from a to g, due to the assignment, as explained above,
2. an edge from a to h, labeled with $\langle f, \mathbf{i} \rangle$, meaning that: the field f of a depends, through implicit flow, on h.

Note that our AMG should not be confused with data dependency graphs such as in [CPS+00] which represent dependencies among registers and memory reads and writes and aim to be used for program slicing.

AMG construction To obtain a modular analysis, we split the construction of the AMG in two steps:

1. an *intra-procedural* analysis which constructs the AMG for programs without method invocation,
2. an *inter-procedural* analysis which adds support for method calls.

Labeling the AMG with security levels Our analysis is more general than "traditional" information flow analysis as it computes an AMG abstracting the dependencies between program data: a node a is related to b if the value of b may influence the value of a. These dependencies are not made explicit in "traditional" information flow analysis and replaced by coarser flows of security levels from an a priori fixed security lattice [Den76].

In our work, we compute these dependencies between values independently of any *a priori* information like security levels, and hence we can use the AMG for other program analysis applications, such as escape analysis, etc. The AMG (the nodes and edges) is labeled with different security policies a posteriori. Therefore, when applied to secure information flow, our approach allows to reuse the same analysis for various security lattices without re-analysing the code.

Let us consider a security lattice with two security levels, low and $high$. Applying security levels to the graph in Figure 5.2c results in the labeled graphs in Figure 5.3. The first graph, in Figure 5.3b,

(a) Source code

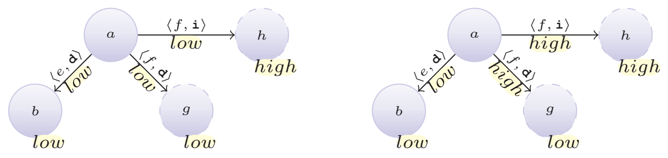
```
class A {
   B e;
   int f;
}

A a;
B b;
int g,h;
```

(b) Unsecure program

```
...
a.e = b;
if(h)
   a.f = g;
```

(c) Secure program

```
...
a.e = b;
if(h)
   a.f = g;
```



a
$\langle f, \mathbf{i} \rangle$
low
h
$high$
$\langle e, \mathbf{d} \rangle$
low
$\langle f, \mathbf{d} \rangle$
low
b
low
g
low
a
$\langle f, \mathbf{i} \rangle$
$high$
h
$high$
$\langle e, \mathbf{d} \rangle$
low
$\langle f, \mathbf{d} \rangle$
$high$
b
low
g
low

Figure 5.3: AMG labeled with security levels

reveals an illegal information flow as some high data, h, can be read from an object low, on a low path. The second graph and its security policies, as depicted in Figure 5.3c, corresponds to a secure program as there is no path from a low to a high.

Non-interference Our AMG is an abstraction of the memory and contains possible relations (due to assignments or implicit flow) between objects and input primitive values. The non-existence of a path in our graph between two nodes ensures that the two nodes are not related. Hence, we can informally state the non-interference for a AMG:

Non-interference property : If a node a is not related to a node b in the AMG (in the sense that there is no path from a to b), then a does not depend on b. In other words, changing the input value of b does not affect the output value of a (*i.e.*, the graph of a does not change).

Soundness The correctness of our construction with respect to non-interference is formally proved in the next chapter.

Chapter structure We first present the instruction set of our analysis and then we formally define the AMG of a method. We continue by presenting the intra-procedural analysis for a sequential program (without method calls). Next, support for method invocation through a compositional analysis is added. We conclude by applying the AMG to information flow. and by showing other possible applications of our analysis.

5.1.2 Hypothesis and the JVM instruction set

We reconsider the instruction set used for the embedded information flow model presented in Figure 3.1 page 25. The instruction set contains all representative Java features, such as objects, method invocation, static fields, arrays.

In this model, we only deal with mono-threaded executions which terminate and do not throw exceptions. Recall that we assume the bytecode programs to be well-typed and to successfully pass the class file verifier. We deal only with flows arising from direct assignments or from implicit flow. Our model does not take into account flows from covert channels, such as timing channels, termination channels, resource exhaustion etc.

5.1.3 Notations

As in the previous chapters, we consider a set of class names $Class$, a set of methods $Method$, and a set $Field$ of field names. For a method m of class C, n_m denotes the number of its arguments and

P_m its instruction list; $P_m[i]$ denotes the ith bytecode of the method m, while $f_{C'}$ designates the field f of the class C'.

We consider finite graphs whose edges are labelled by elements of the set $\mathcal{L}$: a graph G is given by (V, E) where V is its set of vertices (or nodes), and $E \subseteq V \times V \times \mathcal{L}$ is its set of labelled edges. In a graph G, the edge from vertex u to v, labeled with l is denoted by (u, v, l), and $adj_G(u, l)$ is the set of adjacent vertices of u in G, reached by an edge labeled with l. A node u is a leaf in a graph (V, E) if for any node v and label l, $(u, v, l) \notin E$.

A vertex v is reachable from u in a graph G if there is a path (a sequence of edges leading) from u to v. By $Reach_G(u)$ we denote the set of vertices reachable from u in G; $u \in Reach_G(u)$. We define $G\lfloor u \rfloor$ as the subgraph of $G = (V, E)$ given by $(Reach_G(u), \{(v, w, l) \mid (v, w, l) \in E$ and $v, w \in Reach_G(u)\})$. The union of two graphs $(V_1, E_1) \cup (V_2, E_2)$ is the graph $(V_1 \cup V_2, E_1 \cup E_2)$; the graph inclusion $(G_1 \subseteq G_2)$ is defined accordingly.

For a graph $G = (V, E)$, $G[(u, f) \mapsto v]$ agrees with G except that all the edges of the form (u, u', f) in E are replaced by a unique edge (u, v, l).

Finally, for D, a subset of the domain of the function f, $f_{|D}$ is the restriction of f on D.

5.2 Abstract memory graph definition

In the next paragraphs, we give an informal definition and an intuitive explanation of our abstract model for a AMG. An AMG is an abstraction of the JVM heap model, obtained during the execution of a program: nodes in the AMG abstract either objects or input primitive values (method parameters, object fields).

5.2.1 Nodes

Nodes can be classified in two main categories: *reference nodes*, abstracting from objects, and *primitive nodes*, abstracting from primitive input values. Let us denote the set of nodes of AMGs by $Node$.

Reference nodes may abstract method parameters, objects created inside a method (*i.e.*, inside nodes), static nodes. Primitive nodes abstract parameters of primitive types, constants, and initial values for primitive fields of parameter and inside objects. Moreover, there are special kinds of nodes: the special node $n_\perp^{\texttt{null}}$ which models the special reference value `null`, and the return node which holds as fields nodes returned by the method.

Some kind of reference nodes (inside nodes, parameter nodes, static nodes) are the root of a graph structure containing the abstractions of the object fields.

Inside nodes They model objects created by the analyzed method m. An inside node, denoted by n_{pc}^n with $pc \in P_m$, models all the objects created by the execution of the object allocation instruction pc (`new` or `newarray`). We use the *object allocation site model* as all objects created at the same program statement have the same abstraction. Each graph structure rooted at n_{pc}^n contains nodes for each primitive field and edges to the $n_\perp^{\texttt{null}}$ node for object fields.

Constant nodes Denoted by $n_{pc}^c, pc \in P_m$, constant nodes model constant values created at instruction pc. The constant value is encoded in the instruction itself; hence, it represents a source of information which must be taken into account. There is a unique node for every constant creation statement in the method.

Return node This node, denoted by $n^r_\top$, is used to indicate the return of the method being executed. Fields of the return node represent objects or primitive values returned using an α`return` instruction.

Context node The context node, denoted by $n^{cxt}_\perp$, is used to correctly model implicit flows in the compositional approach, where the calling context, hence the condition on which the execution of a method depends, is not available. During the intra-procedural analysis we consider that the execution of the analyzed method implicitly depends on $n^{cxt}_\perp$; during the inter-procedural analysis, we map the real nodes on which the method invocation depends to $n^{cxt}_\perp$.

Static nodes To model static fields, we create a static node n^s_C for each class C containing static fields. If the class C has several static fields, they are modeled as normal fields of n^s_C. Hence, this static node acts as a wrapper for the static fields of the class C.

Parameter nodes For compositional reasons, the analysis of a method cannot use (and does not have access to) objects created outside the scope of the method. Hence, to deal with parameters, we use *placeholders*, which model objects used by method m but created before the method was called.

For each parameter i of a method m, n^p_i denotes the i^{th} parameter of the method. Parameter nodes abstract both parameters of reference type and of primitive type. We assume that the the parameters are maximally unaliased by introducing one parameter node n^p_i for each concrete parameter of a method m. If two parameters are aliased, we discover it in the inter-procedural analysis and we merge the aliased parameters by mapping them to the same nodes. Aliased parameter represent an important case which can easily generate mistakes, hence we carrefully treat them during the composition of two AMGs in Section 5.4.

Outside nodes The outside nodes of a method m model objects created by one of the methods transitively called by the analyzed method. These nodes could be computed during the analysis, but, for simplicity, we consider that they are known from the beginning. We give details how these nodes are computed while explaining the inter-procedural analysis; for now we leave them out of discussion.

5.2.2 Edges

Edges model, on the one hand, heap references and on the other hand, the type of flow. Hence, an edge from a node u to a node v has two labels: $\langle f, t\rangle$, where $f \in \mathit{Field}$ is the name of the field through which u might point to v and $t \in \mathcal{F} = \{\mathtt{d}, \mathtt{i}\}$ is the type of flow: $\mathtt{i}$ for implicit flow and $\mathtt{d}$ for direct flows, with the order relation $\mathtt{i} \sqsubseteq \mathtt{d}$. Hence, edges are modeled as quadruples from the set $\mathit{Node} \setminus \{n^{\mathtt{null}}_\perp\} \times \mathit{Node} \times \mathit{Field} \times \mathcal{F}$. If Edges denotes the set of edges in AMGs, then

$$\mathit{Edges} = \wp(\mathit{Node} \setminus \{n^{\mathtt{null}}_\perp\} \times \mathit{Node} \times \mathit{Field} \times \mathcal{F})$$

and we denote an edge by $(u, v, \langle f, t\rangle)$.

Direct and implicit edges

Depending on the type of flow ($t \in \mathcal{F}$), we can distinguish two types of edges: *direct edges* and *implicit edges*.

Direct edges Direct edges, labeled with $\langle f, \mathtt{d}\rangle$, model heap references. For example, the edge $(a, b, \langle b, \mathtt{d}\rangle)$ in Figure 5.2b shows that a points to b through the field b. Both, a and b, are object abstractions; the edge $(a, g, \langle f, \mathtt{d}\rangle)$ indicates that the field f of a may contain the value abstracted by the primitive node g.

Implicit edges Implicit edges, labeled with $\langle f, \mathtt{i} \rangle$, model data flows arising from the control structure of the program. An edge $(u, v, \langle f, \mathtt{i} \rangle)$ in the AMG means that the field f of u depends (through implicit flow) on v. In the example in Figure 5.2c, the edge $(a, h, \langle f, \mathtt{i} \rangle)$ shows that the value of the field f of a depends on the condition tested by the `if` instruction: someone who knows the control structure of the program and the value of h can infer information about the field f of a.

As the conditional instructions usually test primitive values, implicit edges are between reference nodes and primitive nodes. Bytecode such as `ifnull` which makes reference comparison is discussed in the next paragraphs.

5.2.3 Modeling of array cells

In Java, arrays are a special kind of objects. Hence, we model an array as an object having two fields: *(1)* a primitive field, l, holding the size of the array and *(2)* a reference field, $[]$, holding the cells of the array.

5.2.4 Reference comparison

Some bytecodes (`if_acmp`, `ifnull`, `ifnonnull`) perform reference comparison. For example, `ifnull` a jumps to a if the reference comparison between the top of the stack and `null` succeeds. This may lead to implicit flows between references.

To unify the model, we add a special value field ref of primitive type to each object, which holds the address of the object. When comparing two objects or an object to `null`, the value tested is the ref field. The code `if(o == null) a.f = b` generates an implicit flow from the field f of a to the address ref of o ($(a, o.\mathit{ref}, \langle f, \mathtt{i} \rangle)$).

The special field ref allows us to keep an important graph property stating that implicit edges are always between a reference and a primitive node.

5.2.5 Abstract memory graph definition

The AMG computed by the analysis for a given program point conservatively models the memory state created by any execution path that reaches that point. At a certain program point, the AMG G is a representation of the concrete memory such that, when restricted to objects, G and the concrete memory (which we will later define as a memory graph) are related by an abstraction relation. Then this abstraction is extended with primitive values and implicit flow dependencies.

Formally, an AMG G is a pair

$$G = (V, E) \in \wp(\mathit{Node}) \times \mathit{Edges}$$

consisting of the set of nodes V and the set of edges E.

For an AMG $G = (V, E)$, $\mathcal{O}(V) = \mathcal{O}(G)$ denotes the set of reference nodes (abstracting JVM objects), while $\mathcal{V}(V) = \mathcal{V}(G)$ denotes the set of primitive nodes (corresponding to primitive values). The function $\mathit{Type} : \mathcal{O}(V) \rightarrow \mathit{Class}$ returns the type of a reference node.

AMG properties

Implicit flows are from objects modified inside a region to values on which the execution of the region depends. These facts, combined with the use of the special field ref, allow us to deduce the properties of an AMG G:

Property 5.1. *In an AMG $G = (V, E)$, any primitive node $u \in \mathcal{V}(G) \cup \{n_{\perp}^{null}\}$ is a leaf.*

Property 5.2. *In an AMG $G = (V, E)$, for any implicit edge of type $(u, u', \langle f, \mathtt{i} \rangle) \in E$, u is a reference node and u' is a primitive node ($u \in \mathcal{O}(G) \setminus \{n_{\perp}^{null}\}$ and $u' \in \mathcal{V}(V)$).*

These properties state that edges between two references are always direct edges: if $u_1, u_2 \in \mathcal{O}(G)$ and $(u_1, u_2, \langle f, t\rangle) \in E$ then $t = \mathbf{d}$. Primitive nodes are always leaves; reference nodes, except $n_\perp^{\texttt{null}}$ are never leaves, and implicit edges are always connecting a reference and a primitive node. Thus, the edges generated by the information flow and the primitive values are not imprecated in the graph restricted to objects. Our construction is a points-to graph prolonged with primitive nodes and edges to primitive nodes.

This property is crucial for the soundness of our analysis, as it allows us to split the correctness proof in two parts: we first show the correctness of the points-to graph, and secondly we prove the correctness of the prolongation (with primitive nodes) by stating a non-interference theorem.

5.2.6 Non-interference definition

Having modeled the AMG, we can now analyse the graph in order to verify the non-interference between two nodes. First of all, we define a non-interference predicate for an AMG G, which states that two nodes, u and v, do not interfere if there is no path from u to v (or v is not reachable from u, formally $v \notin \mathit{Reach}_G(u)$).

Definition 5.3 (Non-Interference predicate)**.** Given an AMG $G = (V, E)$, a reference node $u \in \mathcal{O}(G)$ and a primitive node $v \in \mathcal{V}(G)$, the non-interference predicate $ni_G(u, v)$ holds if $v \notin \mathit{Reach}_G(u)$.

5.3 Intra-procedural analysis

Next, we present the intra-method analysis for building the AMG of a method m, focusing on programs without method calls: the algorithm consists of computing an AMG for each program point of the method m. The method invocation and the compositional inter-method analysis are addressed in Section 5.4. In Chapter 6, we prove the correctness of our construction by defining non-interference for a concrete JVM and by proving the soundness of *(1)* the restriction of the AMG to reference nodes (the points-to graph) and *(2)* a non-interference theorem for primitive nodes.

The algorithm is a flow-sensitive abstract interpretation which relies on an abstract transformation rule for each bytecode. The result of the algorithm is an AMG for each method. The construction is context-insensitive: the analysis starts from a “general” initial state, which does not take into account the context under which the method might be called. This leads to an analysis which is more general than a context-sensitive one, but it lies on the simple idea that it is easier to consider unaliased nodes and merge them later (during the inter-procedural analysis) than to consider aliasing and split nodes afterwards.

5.3.1 Control dependency regions

In order to deal with implicit flows, we use the notion of postdominance, as already described in Chapter 3 at page 35. Recall that CF_m denotes the intra-method control flow graph of a method m. For techniqual reasons, we split the instructions on blocks, and we extend the postdominance definitions to CF_B, the control flow graph of blocks.

For a method m, a *block* B is a subset of the instruction list P_m together with a distinguished instruction, called the *entry-point* of B ($\mathsf{Entry}(B)$), such that the control flow graph CF_B of B is a subgraph of CF_m; all vertices in CF_B are reachable from $\mathsf{Entry}(B)$ and CF_B has a unique exit-point ($\mathsf{Exit}(B)$). We assume that P_m is itself a block by adding a unique exit-point to its control flow graph.

We extend the definition of postdominance to the control flow graph of blocks: in a control flow graph CF_B of a block B, a node n' post-dominates a node n if n' belongs to every path from n to $\mathsf{Exit}(B)$. We denote $PD(n)$ the set of post-dominators of n. The definition of immediate post-dominator ipd, control dependency region cdr and context cxt are also extended to blocks and CF_B. Recall that function $cxt(i)$ gives the set of conditional instructions under which i is executed; by Γ_i we denote the set of abstract values tested by conditional instructions in $cxt(i)$. Hence, all abstract values manipulated by instruction i implicitly depend on values in Γ_i. See page 35 for exact definitions.

5.3.2 Abstract domain and abstract execution state

The abstract domain of a method m is represented by the nodes presented in the AMG of the method. The nodes are abstractions of concrete JVM objects and initial primitive values manipulated by a virtual machine.

Building the AMG requires the approximation of local variables array and stack contents. In order to deal with implicit flow, we need to know the conditions under which the local variables and the stack are modified. Thus, elements in the stack and local variables have the form (u, t) where $u \in V$ and $t \in \mathcal{F}$.

Hence, an abstract state has the form $Q = (\rho, s, (V, E))$ where (V, E) is the AMG, ρ denotes the local variables array and it is a mapping from χ_m to $\wp(V \times \mathcal{F})$, while s is the operand stack with elements in $\wp(V \times \mathcal{F})$.

5.3.3 Initial abstract execution state

The analysis of a method starts from an initial abstract state, which initializes the local variables array, the operand stack and the AMG.

We define an initial abstract state of a method m as $Q_{\mathtt{m}}^{init} = (\{0 \mapsto (n_0^p, \mathbf{d}), \ldots, n \mapsto (n_n^p, \mathbf{d})\}, \epsilon, G^{init})$. The initial local variables array stores, in each local variable i, the ith parameter node. The initial state also contains an empty operand stack, denoted by ϵ, and the initial AMG, G^{init} such that $G^{init} = (V, E^i)$, where V is the set of nodes required by the analysis of m and E^i is the set of initial edges between nodes in V, as we will explain in the following paragraphs (*e.g.*, edges between parameters and their fields).

The nodes of the abstract graph, V, could be computed during the analysis, but for simplicity we consider that all the nodes are available from the start. This implies that for each reference parameter (n_i^p) and each inside node (n_i^n), the initial graph not only contains the abstract node designating the object, but also the graph rooted by this abstract node representing all the contents of this object (For parameters of type reference, all the object fields are considered as not null). For convenience, we add all the nodes to the initial graph even if some of them are not used in the method, they are eliminated later at the end of the analysis. Figure 5.4 depicts the initial graph of the method m, by showing different subgraphs rooted by the parameter nodes, the inside nodes, and static nodes needed in m. The complete initial graph of method m is the union of all subgraphs depicted in the figure.

The set V also contains all constant nodes, the return node, the context node (as depicted in Figure 5.4g), the null node, and all the static nodes. Moreover, some nodes manipulated by a method m are created in methods called by m. The initial graph must contain these nodes and subgraphs rooted by these nodes. By G_m^m we denote the initial graph for nodes created in methods invoked by m. The difficulty arises from the existence of inter-dependent methods. During the inter-procedural analysis we show how this graph is computed.

Unfolding parameter nodes Each parameter is abstracted by a node: n_i^p denotes the i^{th} parameter of the method. Hence, we add the minimal subgraph rooted by n_i^p, $G_i^p = (V_i^p, E_i^p)$ to G^{init}. This graph represents the parameter and all of its contents; all object fields are considered as not null. For example, the graph in Figure 5.4b shows the subgraph rooted by the parameter 1 of the

(a) Data structure declaration and source code

```
class A {                           void m(A a, List l){
   int f;                              ...
   B b;                                i:  new A
   static B st;                        ...
   static int s;                       j:  anewarray A
}                                      ...
                                       k:  getstatic A.st
class B {                              ...
   int g;                              k': getstatic A.st
}                                      ...
                                       l:  putstatic A.s
List {                                 ...
   int v;                              o:  bipush 2
   List next;                          ...
}                                   }
```

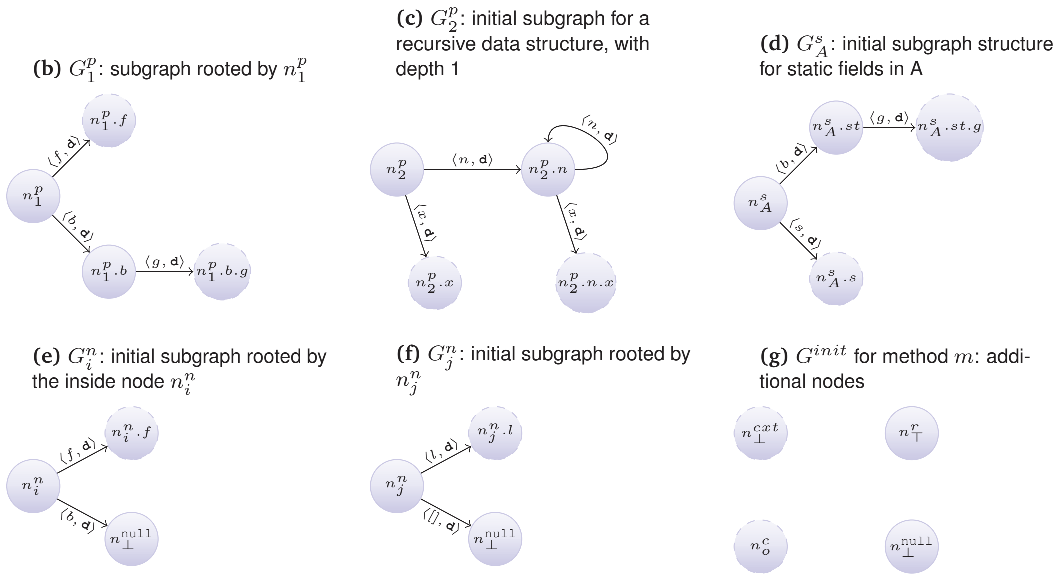

(b) G_1^p: subgraph rooted by n_1^p
n_1^p
$n_1^p.f$
$n_1^p.b$
$n_1^p.b.g$
$\langle f, \mathbf{d}\rangle$
$\langle b, \mathbf{d}\rangle$
$\langle g, \mathbf{d}\rangle$
(c) G_2^p: initial subgraph for a recursive data structure, with depth 1
n_2^p
$n_2^p.n$
$n_2^p.x$
$n_2^p.n.x$
$\langle n, \mathbf{d}\rangle$
$\langle x, \mathbf{d}\rangle$
(d) G_A^s: initial subgraph structure for static fields in A
n_A^s
$n_A^s.st$
$n_A^s.st.g$
$n_A^s.s$
$\langle b, \mathbf{d}\rangle$
$\langle g, \mathbf{d}\rangle$
$\langle s, \mathbf{d}\rangle$
(e) G_i^n: initial subgraph rooted by the inside node n_i^n
n_i^n
$n_i^n.f$
$n_\perp^{null}$
$\langle f, \mathbf{d}\rangle$
$\langle b, \mathbf{d}\rangle$
(f) G_j^n: initial subgraph rooted by n_j^n
n_j^n
$n_j^n.l$
$n_\perp^{null}$
$\langle l, \mathbf{d}\rangle$
$\langle [], \mathbf{d}\rangle$
(g) G^{init} for method m: additional nodes
$n_\perp^{cxt}$
$n_\top^r$
n_o^c
$n_\perp^{null}$

Figure 5.4: The initial AMG of a method m

method; all reference nodes are non null and are completely unfolded. We can define the minimal subgraphs rooted by parameters of a method m as:

$$G_m^p = \bigcup_{0 \leq i \leq n_m} G_i^p.$$

Recursive data structure A special case is represented by the recursive data structures (such as linked lists), as the unfolding process cannot be precise in a static analysis. To avoid infinite graphs which might be generated by such structures, we use a parameter h, which is called the *depth* of the unfolding, and represents how deep we unfold the recursive data structures. The graph G_i^p is the minimal graph that contains the parameter node n_i^p such that $\forall u \in \mathcal{O}(G_i^p)$, for each field f_1 of $\mathit{Type}(u)$:

- either there exists a path $u_1^1 u_2^1 \ldots u_k^1 u_1^2 u_2^2 \ldots u_k^2 \ldots u_1^h u_2^h \ldots u_k^h u$ labelled by $(\langle f_1, \mathbf{d}\rangle\langle f_2, \mathbf{d}\rangle\langle f_3, \mathbf{d}\rangle \ldots \langle f_k, \mathbf{d}\rangle)^h$ in G_i^p such that $\forall 1 \leq i \leq h, 1 \leq j \leq h$, $\mathit{Type}(u_l^i) = \mathit{Type}(u_l^j)$, then $(u, u_1^h, \langle f_1, \mathbf{d}\rangle) \in E_i^p$.

- or $u.f_1 \in V_i^p$ and $(u, u.f_1, \langle f, \mathbf{d}\rangle) \in E_i^p$.

Let us consider the class `List` of Figure 5.4 which implements a linked list. The subgraph rooted by the second parameter (of type List) of method m is depicted in Figure 5.4c. The depth of the unfolding considered in this example is equal to 1. Hence, the node $n_2^p.n$ abstracts the field n of n_2^p but also all subsequent fields n.

Unfolding inside nodes For each allocating instruction pc, n_{pc}^n denotes the abstraction of all objects created at instruction pc. The initial graph G^{init} contains the subgraph rooted at n_{pc}^n, $G_{pc}^n = (V_{pc}^n, E_{pc}^n)$. This graph contains nodes for each primitive field, and edges to the abstraction of `null` for object fields. We define the minimal subgraphs rooted by inside nodes of a method m as:

$$G_m^n = \bigcup_{P_m[pc]=\texttt{new}} G_{pc}^n \cup \bigcup_{P_m[pc]=\texttt{anewarray}} G_{pc}^n.$$

The initial graphs for the `new` instruction at bytecode i is depicted in Figure 5.4e, while the creation of an array with the instruction `anewarray` at bytecode j is depicted in Figure 5.4f.

Unfolding static nodes To model static fields, we define a node n_C^s for each class C holding its static fields. For each class we build its static graph, G_C^s containing the the root node n_C^s and a node for each static field of C. If the static field is of primitive type, then we have a primitive node; if the static field is a reference, the minimal graph rooted by this node is added (similar to reference parameters: all object fields are considered as non null).

By G^s we denote the static graph of all classes: $G^s = \{G_C^s \mid C \in \mathit{Class}\}$. As this graph can statically be computed, we consider that it is known from the start and the initial graph G^{init} of a method contains it: $G^s \subset G^{init}$.

Initial abstract memory graph: definition We can now formally define the initial AMG of a method m as the union of initial graphs of formal parameters, of inside nodes, and of static nodes; moreover the initial graph contains the constant nodes, and the rest of special nodes $(n_\perp^{\texttt{null}}, n_\top^r, n_\perp^{cxt})$:

$$G_m^{init} \quad = \quad G_m^p \cup G_m^n \cup G^s \cup G_m^m \cup (\{n_\perp^{\texttt{null}}, n_\top^r, n_\perp^{cxt}\} \cup \{n_{pc}^c \mid P_m[pc] = \texttt{bipush}\}, \emptyset)$$

5.3.4 Abstract semantics

So far we defined the design of the AMG. The AMG of a method is constructed from an initial state by applying abstract semantics rules corresponding to JVM instructions. We now give these semantics rules and explain how they model the behaviour of the JVM w.r.t. the abstract execution state and AMG.

For each bytecode b, we define an abstract rule $Q' = instr_b(Q, \Gamma)$ where Γ is the set of nodes u corresponding to values on which the execution of the bytecode depends, reflecting the impact of the implicit flow. In order to simulate the context under which a method might be called, we use a dummy node, $n_\perp^{cxt}$. Hence in the initial state, $\Gamma^{init} = \{n_\perp^{cxt}\}$. The abstract rules are presented in Figure 5.5.

To reflect the impact of control regions on the stack and the local variables array, every instruction modifying the last two (push for stack or store for local variables array) takes the values in the context Γ into consideration. For example, the instructions `new` and `bipush` push new abstract values on the stack as well as the context under which the operation takes place, represented by

$$.\ \frac{(\rho, v_1 :: v_2 :: s, G)}{(\rho, v_1 \cup v_2 \cup TV_\Gamma :: s, G)}\,\texttt{prim}\ op \qquad \frac{(\rho, v :: s, G)}{(\rho, s, G)}\,\texttt{pop}$$

$$\frac{(\rho, s, G)}{(\rho, \{(n_i^c, \mathtt{d})\} \cup TV_\Gamma :: s, G)}\,\texttt{bipush}\ iv \qquad \frac{(\rho, s, G)}{(\rho, \{(n_\perp^{\mathtt{null}}, \mathtt{d})\} \cup TV_\Gamma :: s, G)}\,\texttt{aconst\_null}$$

$$\frac{(\rho, s, G)}{(\rho, \{(n_i^n, \mathtt{d})\} \cup TV_\Gamma :: s, G)}\,\texttt{new}\ iC \qquad \frac{(\rho, s, G)}{(\rho, \{(n_i^n, \mathtt{d})\} \cup TV_\Gamma :: s, G)}\,\alpha\texttt{newarray}\ iC$$

$$\frac{(\rho, v :: s, G)}{(\rho, s, G)}\,\texttt{ifeq}\ a \qquad \frac{(\rho, v :: s, G)}{(\rho, s, G)}\,\texttt{ifnull}\ a \qquad \frac{(\rho, s, G)}{(\rho, s, G)}\,\texttt{goto}\ a$$

$$\frac{(\rho, s, G)}{(\rho, \rho(x) \cup TV_\Gamma :: s, G)}\,\alpha\texttt{load}\ x \qquad \frac{(\rho, u :: s, G)}{(\rho[x \mapsto u \cup TV_\Gamma], s, G)}\,\alpha\texttt{store}\ x$$

$$\frac{(\rho, u :: s, G)}{(\rho, \{(e,t) \mid e \in adj_G(e', \langle f_{C'}, t\rangle) \wedge e' \in V_{\mathtt{d}}^u\} \cup \{(e, \mathtt{i}) \mid e \in V_{\mathtt{i}}^u\} \cup TV_\Gamma :: s, G)}\,\texttt{getfield}\ f_{C'}$$

$$\frac{(\rho, v :: u :: s, (V, E))}{\left(\rho, s, \left(V, \begin{array}{ll} E & \cup\{(e, e', \langle f_{C'}, t\rangle) | (e, \mathtt{d}) \in u, e \neq n_\perp^{\mathtt{null}}, (e', t) \in v\} \\ & \cup\{(e, e', \langle f_{C'}, \mathtt{i}\rangle) | (e, \mathtt{d}) \in u, e \neq n_\perp^{\mathtt{null}}, e' \in \Gamma \cup V_{\mathtt{i}}^u\} \end{array}\right)\right)}\,\texttt{putfield}\ f_{C'}$$

$$\frac{(\rho, s, G)}{(\rho, \{(e,t) \mid e \in adj_G(n_{C'}^s, \langle f_{C'}, t\rangle)\} \cup TV_\Gamma :: s, G)}\,\texttt{getstatic}\ f_{C'}$$

$$\frac{(\rho, v :: s, (V, E))}{(\rho, s, (V, E \cup \{(n_{C'}^s, e', \langle f_{C'}, t\rangle) | (e', t) \in v\} \cup \{(n_{C'}^s, e', \langle f_{C'}, \mathtt{i}\rangle) | e' \in \Gamma\}))}\,\texttt{putstatic}\ f_{C'}$$

$$\frac{(\rho, v :: u :: s, G)}{(\rho, \{(e,t) \mid e \in adj_G(e', \langle [], t\rangle) \wedge e' \in V_{\mathtt{d}}^u\} \cup \{(e, \mathtt{i}) \mid e \in V_{\mathtt{i}}^u\} \cup TV_\Gamma \cup v :: s, G)}\,\alpha\texttt{aload}$$

$$\frac{(\rho, v_1 :: v_2 :: u :: s, (V, E))}{\left(\rho, s, \left(V, \begin{array}{ll} E & \cup\{(e, e', \langle [], t\rangle) | (e, \mathtt{d}) \in u, e \neq n_\perp^{\mathtt{null}}, (e', t) \in v_1\} \\ & \cup\{(e, e', \langle [], \mathtt{i}\rangle) | (e, \mathtt{d}) \in u, e \neq n_\perp^{\mathtt{null}}, e' \in \Gamma \cup V_{\mathtt{i}}^u\} \end{array}\right)\right)}\,\alpha\texttt{astore}$$

$$\frac{(\rho, u :: s, G)}{(\rho, \{(e,t) \mid e \in adj_G(e', \langle l, t\rangle) \wedge e' \in V_{\mathtt{d}}^u\} \cup \{(e, \mathtt{i}) \mid e \in V_{\mathtt{i}}^u\} \cup TV_\Gamma :: s, G)}\,\texttt{arraylength}$$

with $V_{\mathtt{d}}^u = \{e \mid (e, \mathtt{d}) \in u\}$ $\quad V_{\mathtt{i}}^u = \{e \mid (e, \mathtt{i}) \in u\}$ $\quad TV_\Gamma = \{(e, \mathtt{i}) \mid e \in \Gamma\}$

Figure 5.5: Subset of the abstract transformation rules

elements in $TV_\Gamma = \{(e, \mathtt{i}) \mid e \in \Gamma\}$. The α`store` bytecode not only stores the top of the stack in the local variables array, but also the elements in TV_Γ, as the writing is done under their control.

Most of the instructions are trivial and the details will not be given here. We explained how the implicit flow is taken into account by the abstract semantics and now we discuss the most important bytecodes, which handle object fields and generate dependencies (*i.e.*, `putfield` and `getfield`).

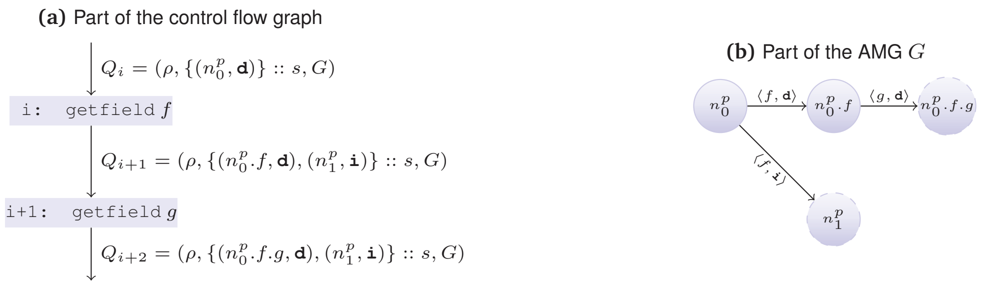

(a) Part of the control flow graph
$Q_i = (\rho, \{(n_0^p, \mathtt{d})\} :: s, G)$
i: getfield f
$Q_{i+1} = (\rho, \{(n_0^p.f, \mathtt{d}), (n_1^p, \mathtt{i})\} :: s, G)$
i+1: getfield g
$Q_{i+2} = (\rho, \{(n_0^p.f.g, \mathtt{d}), (n_1^p, \mathtt{i})\} :: s, G)$
(b) Part of the AMG G
n_0^p
$\langle f, \mathtt{d}\rangle$
$n_0^p.f$
$\langle g, \mathtt{d}\rangle$
$n_0^p.f.g$
$\langle f, \mathtt{i}\rangle$
n_1^p

Figure 5.6: Getfield example

If u is a reference node, the adjacent of u is pushed on the stack by the instruction getfield. If u is a primitive node arising from implicit flow, the instruction keeps it on the stack. As the instruction modifies the stack, values in the context (TV_Γ) are also pushed on the stack. Let us consider the example in Figure 5.6a and a part of the corresponding AMG in Figure 5.6b. The first getfield, starting from the state Q_i, will push the field f of n_0^p (hence, the node $(n_0^p.f, \mathtt{d})$) on the stack, but also $(n_1^p, \mathtt{i})$ as the field f implicitly depends on n_1^p (there is an edge $(n_0^p, n_1^p, \langle f, \mathtt{i}\rangle)$ in the AMG). The second getfield instruction, executed on the newly obtained state Q_{i+1}, will push $(n_0^p.f.g, \mathtt{d})$ (the field g of $n_0^p.f$) on the stack, and it will keep $(n_1^p, \mathtt{i})$ on the stack. The intuition behind is that if the field f of n_0^p implicitly depends on a node (n_1^p in our case), then all fields of fields f and so on also implicitly depend on this element.

The most significant bytecode is putfield, as it modifies the AMG. It adds three types of edges:

- edges between direct nodes in u (having the form $(e, \mathtt{d})$), and elements in v. The type of flow preserves the type of elements in v (if $(e', \mathtt{i}) \in v$, then the edge arises from implicit flow, too),
- implicit flow edges from direct nodes in u (of form $(e, \mathtt{d})$) to implicit nodes in u (of form $(e', \mathtt{i})$). The presence of pairs like $(e', \mathtt{i})$ in u means that the reference nodes in u depend on e'. Thus, we propagate the implicit dependencies of an object to every field being modified of that object,
- implicit flow edges from the nodes being modified ($(e, \mathtt{d}) \in u$) to nodes in Γ (nodes corresponding to values on which the execution of the instruction depends). The idea is that someone who knows the control structure of the program and can observe the field f of nodes being modified, is able to deduce the values in Γ.

For instance, the example in Figure 5.7 sets the field g of $n_0^p.f$ to n_2^p. Informally, $n_0^p.f.g = n_2^p$. Instruction getfield loads the field f of n_0^p on the stack, while iload loads n_2^p on the stack. The putfield instruction, executed on Q_{i+2}, adds two edges: *(1)* an edge $(n_0^p.f, n_2^p, \langle g, \mathtt{d}\rangle)$ which corresponds to the direct assignment and *(2)* an edge $(n_0^p.f, n_1^p, \langle g, \mathtt{i}\rangle)$. The second edge is a result of the fact that $(n_1^p, \mathtt{i})$ was loaded on the stack by the getfield bytecode. Hence, the node $n_0^p.f$ implicitly depends on n_1^p and, as a consequence, all fields of $n_0^p.f$ should also depend on n_1^p implicitly.

The rules dealing with static variables (getstatic, putstatic) and with arrays (αaload, αastore) are similar to getfield and putfield. The instructions getstatic and putstatic are a simplified version as the reference on which we get/put a field is a statically known class, abstracted by node $n^s_{C'}$. Instructions αaload and αastore deal with arrays and with the special field [], which holds all elements of an array.

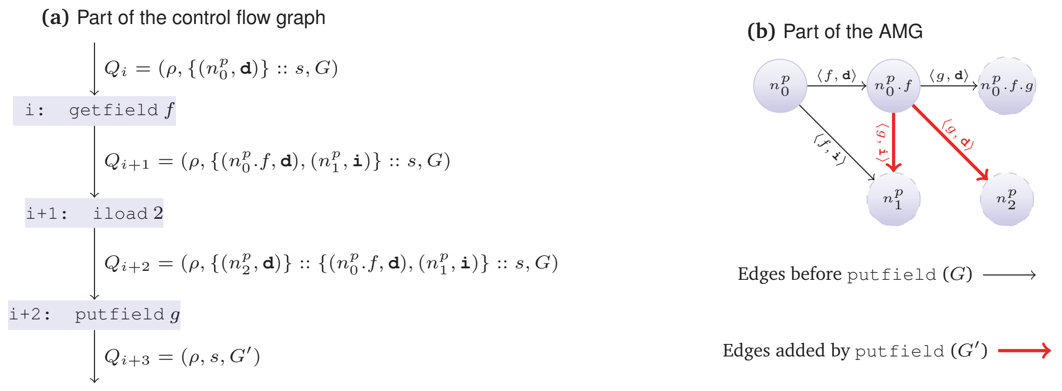

(a) Part of the control flow graph
$Q_i = (\rho, \{(n_0^p, \mathbf{d})\} :: s, G)$
i: getfield f
$Q_{i+1} = (\rho, \{(n_0^p.f, \mathbf{d}), (n_1^p, \mathbf{i})\} :: s, G)$
i+1: iload 2
$Q_{i+2} = (\rho, \{(n_2^p, \mathbf{d})\} :: \{(n_0^p.f, \mathbf{d}), (n_1^p, \mathbf{i})\} :: s, G)$
i+2: putfield g
$Q_{i+3} = (\rho, s, G')$
(b) Part of the AMG
n_0^p
$\langle f, \mathbf{d}\rangle$
$n_0^p.f$
$\langle g, \mathbf{d}\rangle$
$n_0^p.f.g$
$\langle f, \mathbf{i}\rangle$
$\langle g, \mathbf{i}\rangle$
$\langle g, \mathbf{d}\rangle$
n_1^p
n_2^p
Edges before putfield (G)
Edges added by putfield (G')

Figure 5.7: Putfield example

5.3.5 Algorithm

Our analysis is flow sensitive, computing an AMG at each program point; it is defined in the context of a monotone framework [NNH99] for data flow analysis. To comply with this framework, we define an order relation $\sqsubseteq$ and a join operator $\sqcup$ on the property space $\mathcal{S}$ (the set of pairs (Q, Γ) augmented with $\perp$ the neutral element of $\sqcup$) such that $(\mathcal{S}, \sqsubseteq, \sqcup)$ forms a semi-join lattice which satisfies the Ascending Chain Property [NNH99] (all increasing sequences in $\mathcal{S}$ eventually become constant).

Definition 5.4 (Ordering relation on $\mathcal{S}$)**.** Let $S_1 = ((\rho_1, s_1, G_1), \Gamma_1) \in \mathcal{S}$ and $S_2 = ((\rho_2, s_2, G_2), \Gamma_2) \in \mathcal{S}$ be two states. Then, S_1 is said to be *smaller* than S_2, *i.e.*, $S_1 \sqsubseteq S_2$ if and only if $s_1 \sqsubseteq s_2$, $\rho_1 \sqsubseteq \rho_2$, $G_1 \subseteq G_2$ and $\Gamma_1 \subseteq \Gamma_2$.

To order the local variables arrays ρ_1 and ρ_2, we use the classical point-wise ordering relation between functions. Given two stacks s_1 and s_2, s_1 is said to be smaller than s_2, *i.e.*, $s_1 \sqsubseteq s_2$, if both stacks are empty or if $s_1 = v_1 :: s_1'$ and $s_2 = v_2 :: s_2'$ with $v_1 \subseteq v_2$ and $s_1' \sqsubseteq s_2'$.

The join operator is defined as follows: $(Q_1, \Gamma_1) \sqcup (Q_2, \Gamma_2) = (Q_1 \sqcup Q_2, \Gamma_1 \cup \Gamma_2)$. The join operator $\sqcup$ on two states $(\rho_1, s_1, G_1) \sqcup (\rho_2, s_2, G_2) = (\rho_1 \sqcup \rho_2, s_1 \sqcup s_2, G_1 \cup G_2)$ is a component-wise operator.

Then, $(\mathcal{S}, \sqsubseteq, \sqcup)$ forms a semi-join lattice which satisfies the Ascending Chain Property [NNH99] (all increasing sequences in $\mathcal{S}$ eventually become constant) required by the monotone framework.

The execution of a method is abstracted to a set of equations based on the abstract rules $instr_b$: the analysis of a block of instructions $B \subseteq P_m$ of a method m, assumed to be represented by its control flow graph, is described by an equation system $\mathcal{E}_B$, starting from a given initial state $(Q^{init}, \Gamma^{init})$ (recall that we consider terminating executions without exceptions and initial states that allow such executions). For every node i in the control flow graph of B, (Q_i, Γ_i) represents the state and the context (required by the implicit flow) under which the instruction i is executed. The context represents the conditions tested (the top of the stack) by the instructions in $cxt(i)$. Thus, for all nodes i in CF_B:

$$\begin{aligned}(Q_i, \Gamma_i) = \quad & (S_Q \sqcup (\sqcup_{j \in pred(i)} instr_{B[j]}(Q_j, \Gamma_j)),\\ & S_\Gamma \cup \{val(u) \mid \langle u, t\rangle \in v \text{ with } Q_k = (\rho, v :: s, G), k \in cxt(i)\})\end{aligned} \tag{5.3.1}$$

where (S_Q, S_Γ) equals to $(Q^{init}, \Gamma^{init})$ if $i = \mathsf{Entry}(B)$ and to $(\perp, \varnothing)$ otherwise; $val(u) = u$ if $u \in \mathcal{V}(G)$, or $val(u) = u.ref$ if $u \in \mathcal{O}(G)$.

The abstract rules are monotone with respect to the ordering relation $\sqsubseteq$, thus we can solve the system (5.3.1) using standard methods for monotone dataflow analysis.

Lemma 5.5 (Monotonicity of the transformation rules)**.** *For every instruction b, the transformation rule $instr_b$ is monotone with respect to the ordering relation $\sqsubseteq$.*

Proof. The Proof of this Lemma is made by case analysis on each instruction b, based on the transformation rules in Figure 5.5, it is given in Appendix A.1. □

Our algorithm is a forward dataflow may-analysis [HBCC99], similar to [SR01, WR99], as the computed AMG for a given program point is the union of graphs created by all the execution paths reaching that point. For a block B and a pair (Q, Γ), we define $instr_B(Q, \Gamma)$ as the state $instr_{B[e]}(Q_e, \Gamma_e)$ where $e = \mathsf{Exit}(B)$ and (Q_e, Γ_e) is obtained as the least solution of the system of equations $\mathcal{E}_B$ starting from the initial state (Q, Γ).

Example 5.1. Based on the source code for tax calculation in Figure 5.8a, a detailed analysis example is presented in Figure 5.8. Method `tax` computes the tax based on the salary and some predefined rates. The table in Figure 5.8e presents, for each instruction i, its associated abstract state Q_i. Recall that Q_i represents the state *before* executing instruction i. Note that the symbol $\backsim$ denotes repetition, *i.e.*, an equal set to the one above in the column.

The analysis consists of applying each instruction i on the state Q_i, starting from an initial state $Q_0 = (\rho^{init}, \epsilon, G^{init})$ and $\Gamma_0 = \{n_\perp^{cxt}\}$, according to equation 5.3.1. The initial local variables array contains formal parameters:

$$\rho^{init} = \{0 \mapsto \{(n_0^p, \mathbf{d})\}, 1 \mapsto \{(n_1^p, \mathbf{d})\}, 2 \mapsto \emptyset\}.$$

The initial AMG G^{init}, depicted in Figure 5.8c, unfolds the fields of formal parameters and contains also the context node $n_\perp^{cxt}$ [3].

A state Q_i is computed as the union of states created by each execution path reaching i. The only instruction having two predecessors is `13`, hence Q_{13} is the union of states resulting from the execution of `9` and `12`. The equation system ensures that all execution paths are considered, as the algorithm iterates on the set of instructions until a fixed point has been reached. The table in Figure 5.8e shows the abstract states for the final iteration.

Regarding implicit flows, all instructions depend on the dummy node $n_\perp^{cxt}$. Moreover, instructions `6-12` belong to the control dependency region of `5`[4], $cdr(5) = \{6, 7, 8, 9, 10, 11, 12\}$. Hence, their execution depends on the values tested by the comparison bytecode `5` (that is to say, the top of stack in Q_5), hence

$$\Gamma_6 = \cdots = \Gamma_{11} = \Gamma_{12} = \Gamma' = \{n_1^p.avg, n_0^p.sal, n_\perp^{cxt}\}.$$

All operations performed by these instructions (push, array store, load) take Γ' into account.

The only instruction modifying the AMG is `putfield`, at line `18`. It assigns a value to field tax of n_0^p and generates the following edges: (i) edges labeled $\langle tax, \mathbf{d}\rangle$ from n_0^p to $n_0^p.sal$, $n_1^p.min$, $n_1^p.max$ and (ii) edges labeled $\langle tax, \mathbf{i}\rangle$ from n_0^p to $n_0^p.sal$, $n_0^p.avg$ and $n_\perp^{cxt}$, as the local variable 2, was modified in a control region depending on these nodes. Hence, the AMG of the method `tax` contains dependencies between fields `tax` and `sal`, `min`, `max` (due to direct flow) and between `tax` and `avg`, `sal` (due to implicit flow). The resulting graph, depicted in Figure 5.8d, is also the final graph of the method.

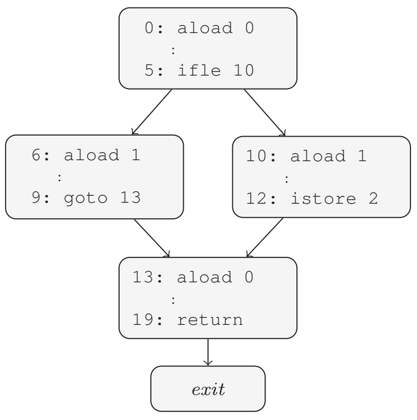
[3]The initial graph must also contain the special node $n_\perp^{\mathtt{null}}$, but, for simplicity reasons and since it is not revelant for our example, we ommit it.

[4]The instruction `ifle` a is similar to `ifeq` only that it jumps to address a if the top of the stack is less than 0.

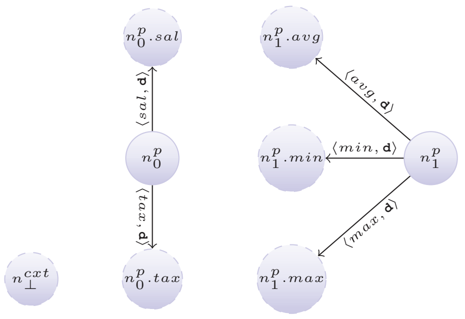
(a) Source code

```
class Rate {
  int avg, min, max;
}

class Income {
  int sal, tax;
  ..
  void tax(Rate p1){
   int tmp;
   if(sal < p1.avg)
     tmp = p1.min;
   else
     tmp = p1.max;
   tax = sal * tmp;
 }
}
```

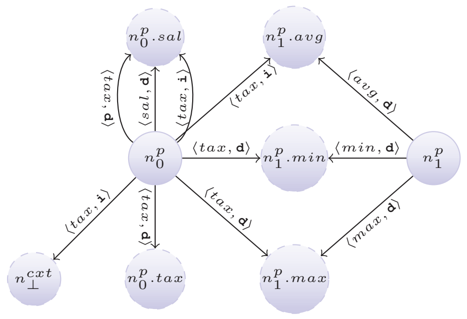
(b) Control flow graph, CF_{tax}



(c) Initial AMG G^{init}


$n_0^p.sal$
$\langle sal, \mathbf{d}\rangle$
n_0^p
$\langle tax, \mathbf{d}\rangle$
$n_0^p.tax$
$n_\perp^{cxt}$
$n_1^p.avg$
$\langle avg, \mathbf{d}\rangle$
$n_1^p.min$
$\langle min, \mathbf{d}\rangle$
n_1^p
$\langle max, \mathbf{d}\rangle$
$n_1^p.max$

(d) AMG G


$n_0^p.sal$
$\langle tax, \mathbf{d}\rangle$
$\langle sal, \mathbf{d}\rangle$
$\langle tax, \mathbf{i}\rangle$
$\langle tax, \mathbf{i}\rangle$
$n_1^p.avg$
$\langle avg, \mathbf{d}\rangle$
n_0^p
$\langle tax, \mathbf{d}\rangle$
$n_1^p.min$
$\langle min, \mathbf{d}\rangle$
n_1^p
$\langle tax, \mathbf{i}\rangle$
$\langle tax, \mathbf{d}\rangle$
$\langle tax, \mathbf{d}\rangle$
$\langle max, \mathbf{d}\rangle$
$n_\perp^{cxt}$
$n_0^p.tax$
$n_1^p.max$

(e) Step by step analysis

Instruction i	Q_i			Γ_i
	ρ_i	s_i	G_i	
`0 : aload 0`	ρ^{init}	ϵ	G^{init}	Γ_0
`1 : getfield sal`	$\backsim$	$\{(n_0^p, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`2 : aload 1`	$\backsim$	$\{(n_0^p.sal, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`3 : getfield avg`	$\backsim$	$\{(n_1^p, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \{(n_0^p.sal, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`4 : isub`	$\backsim$	$\{(n_1^p.avg, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \{(n_0^p.sal, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`5 : ifle 10`	$\backsim$	$\{(n_1^p.avg, \mathbf{d}), (n_0^p.sal, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`6 : aload 1`	$\backsim$	ϵ	$\backsim$	Γ'
`7 : getfield min`	$\backsim$	$\{(n_1^p, \mathbf{d})\} \cup TV_{\Gamma'} :: \epsilon$	$\backsim$	$\backsim$
`8 : istore 2`	$\backsim$	$\{(n_1^p.min, \mathbf{d})\} \cup TV_{\Gamma'} :: \epsilon$	$\backsim$	$\backsim$
`9 : goto 13`	ρ_9	ϵ	$\backsim$	$\backsim$
`10: aload 1`	ρ^{init}	ϵ	$\backsim$	Γ'
`11: getfield max`	$\backsim$	$\{(n_1^p, \mathbf{d})\} \cup TV_{\Gamma'} :: \epsilon$	$\backsim$	$\backsim$
`12: istore 2`	$\backsim$	$\{(n_1^p.max, \mathbf{d})\} \cup TV_{\Gamma'} :: \epsilon$	$\backsim$	$\backsim$
`13: aload 0`	$\rho_9 \sqcup \rho_{12}$	ϵ	$\backsim$	Γ_0
`14: aload 0`	$\backsim$	$\{(n_0^p, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`15: getfield sal`	$\backsim$	$\{(n_0^p, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \{(n_0^p, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`16: iload 2`	$\backsim$	$\{(n_0^p.sal, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \{(n_0^p, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`17: imul`	$\backsim$	$u :: \{(n_0^p.sal, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \{(n_0^p, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`18: putfield tax`	$\backsim$	$u \cup \{(n_0^p.sal, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \{(n_0^p, \mathbf{d}), (n_\perp^{cxt}, \mathbf{i})\} :: \epsilon$	$\backsim$	$\backsim$
`19: return`	$\backsim$	ϵ	G	$\backsim$

$$
\begin{aligned}
\Gamma' &= \{n_1^p.avg, n_0^p.sal, n_\perp^{cxt}\} \\
TV_\Gamma &= \{(e, \mathbf{i}) \mid e \in \Gamma\} \\
\rho^{init} &= \{0 \mapsto \{(n_0^p, \mathbf{d})\}, 1 \mapsto \{(n_1^p, \mathbf{d})\}, 2 \mapsto \emptyset\} \\
\rho_9 &= \{0 \mapsto \{(n_0^p, \mathbf{d})\}, 1 \mapsto \{(n_1^p, \mathbf{d})\}, 2 \mapsto \{(n_1^p.min, \mathbf{d})\} \cup TV_{\Gamma'}\} \\
\rho_{12} &= \{0 \mapsto \{(n_0^p, \mathbf{d})\}, 1 \mapsto \{(n_1^p, \mathbf{d})\}, 2 \mapsto \{(n_1^p.max, \mathbf{d})\} \cup TV_{\Gamma'}\} \\
u &= \{(n_1^p.max, \mathbf{d}), (n_1^p.min, \mathbf{d}), (n_1^p.avg, \mathbf{i}), (n_0^p.sal, \mathbf{i}), (n_\perp^{cxt}, \mathbf{i})\}
\end{aligned}
$$

Figure 5.8: Example 5.1

5.4 Inter-procedural analysis: a compositional approach

The previous section enforces non-interference for sequential programs. In order to be context-insensitive, the analysis is compositional and computes an AMG for each method. In this section, we add support for method invocation by describing the composition of two AMGs (of the caller and callee).

A context-insensitive AMG is computed for each method; this graph represents the execution of the method on formal parameters (*i.e.*, without considering any particular context). Then, when a method m is invoked, we compose the AMG of method m with the current state, instead of analysing the code of m.

In this section, we concentrate on a complete definition and description of the composition process. We discuss the soundness of method invocation in Chapter 6. To prove the soundness of this approach, we show that the result of composition gives an AMG at least equal to the graph obtained by an approach based on the inter-procedural control flow graph (ICFG). Hence, first we present and prove the correctness of the ICFG approach, and afterwards we prove the correctness of the compositional approach, by comparing it with the ICFG approach.

5.4.1 The initial AMG of a method

We define the AMG of a method as the graph resulting from the execution of the method on formal parameters, starting from the initial state as defined in Section 5.3.3. Recall that the initial state is $(Q_m^{init}, \Gamma_m^{init})$, with

$$\begin{array}{rcl} Q_m^{init} & = & (\{0 \mapsto \{(n_0^p, \mathbf{d})\} \ldots n \mapsto \{(n_n^p, \mathbf{d})\}\}, \varepsilon, G_m^{init}), \\ \Gamma_m^{init} & = & \{n_\perp^{cxt}\}. \end{array}$$

Thus, the stack (that corresponds to the concrete new frame) is empty at the beginning of the method, and the local variables array contains formal parameters.

Initial and new edges

We need to distinguish two kinds of edges: *initial edges*, which belong to the initial graph, and model dependencies between nodes which might exist before the execution of the method; and *new edges*, which model edges created by the analyzed scope (the current method and by one of the methods transitively called by it).

This distinction is needed by the compositional approach: in the inter-procedural analysis, initial edges are used to map nodes first, while new edges are translated to edges between the mapped nodes afterwards.

Hence, in the inter-procedural analysis we define the AMG G as $G = (V, E \cup E^c)$, where E represent the initial edges (*i.e.*, the edges from G^{init}) and E^c the edges added by the analysis. Using this notation, all edges generated by the abstract semantics rules belong to E^c. Note that $E \cap E^c \neq \emptyset$, as edges from E might be added to E^c (*e.g.*, by the dummy assignment `p0.f = p0.f`).

Addition to the definition of the initial graph

The graph G_m^{init} is the initial AMG. For simplicity, we assume that all the nodes of the graph are known from the start, but they could be computed during the analysis. Hence, the initial graph contains, among others, the subgraphs rooted by parameter nodes, by inside nodes (created in the method), etc., but also contains subgraphs rooted by nodes created in invoked methods, in methods invoked in invoked methods, etc. (we denote the union of all these subgraphs by G^m).

In Section 5.3.3, we presented the initial graph but we left the computation of G^m outside of the discussion. Here, we present how we deal with such subgraphs.

First, we extend our naming strategy to keep disjoint instruction number sets for methods and unique node names in the AMG. Thus, each instruction pc of a method m is now named m^{pc} (with m the fully qualified name of the method). Therefore, in node names of the AMG like n^c_{pc} and n^n_{pc}, pc now encodes the fully qualified name of the method and the line number. For example, a newly allocated object at instruction i in m is identified by $n^n_{m^i}$.

Thus, if m_1 calls the method m_2 at instruction i,

```
m1
  ...
  pc: invoke m2
  ...
```

the initial graph of m_1 contains the subgraphs of surviving nodes of m_2 (nodes created in m_2); we denote this subgraph by $G^m_{m_1,m_2,pc}$. The graph $G^m_{m_1,m_2,pc}$ contains nodes created in m_2 but we have to add a proper renaming in order to have a unique identifier for each node: intuitively, they will have the form $n^m_{pc}.u$, where u represents the *surviving* nodes in m_2. Hence, we can define the initial graph of nodes created by methods invoked by m_1 as

$$G^m_{m_1} = \bigcup_{P_{m_1}[pc]=\texttt{invoke}\ m_2} G^m_{m_1,m_2,pc}.$$

We now formally define $G^m_{m_1,m_2,pc}$; let $G^m_{m_1,m_2,pc} = (V, E \cup E^c)$. We first consider the case when m_1 and m_2 are not mutually recursive methods, and then we add support for such a case.

Non-recursive methods In the case when m_1 and m_2 are not inter-dependent methods (*i.e.*, m_2 or methods invoked in m_2 etc. do not invoke m_1), the graph definition of $G^m_{m_1,m_2,pc}$ is straightforward: it contains nodes having the form $n^m_{pc}.u$, where u represents the *surviving* nodes in m_2 (*i.e.*, nodes in subgraph of surviving nodes in m_2, $G^{Surv}_{m_2} = (V', E' \cup E^{c\prime})$) and translates edges between surviving nodes (*i.e.*, edges between newly allocated objects and their primitive fields):

$$\begin{array}{lcl} V & = & \{ren_{m_1,m_2}(u, pc) \mid u \in V'\} \\ E & = & \{(ren_{m_1,m_2}(u, pc), ren_{m_1,m_2}(v, pc), \langle f, t, \rangle) \mid (u, v, \langle f, t, \rangle) \in E'\} \\ E^c & = & \{(ren_{m_1,m_2}(u, pc), ren_{m_1,m_2}(v, pc), \langle f, t, \rangle) \mid (u, v, \langle f, t, \rangle) \in E^{c\prime}\}. \end{array}$$

The function $ren_{m_1,m_2} : V' \times P_{m_1} \to V$ renames surviving nodes in m_2 in the following way:

$$ren_{m_1,m_2}(u, pc) = \begin{cases} n_\perp^{\texttt{null}} & \text{if } u = n_\perp^{\texttt{null}} \\ n^m_{pc}.u & \text{otherwise} \end{cases}$$

This function ensures that there is only a unique node $n_\perp^{\texttt{null}}$.

We only need to define $G^{Surv}_{m_2} = (V', E', E^{c\prime})$, which denotes the subgraphs rooted in surviving nodes in m_2. Surviving nodes of m_2 are nodes whose lifetime may exceed the execution of the method, *i.e.*, $n_\perp^{\texttt{null}}$, constant nodes of form n^c_i, inside nodes (nodes representing objects created in m_2 and denoted by subgraph $G^n_{m_2}$), and surviving nodes of methods called by m_2. We can formally define $G^{Surv}_{m_2}$ as:

$$G^{Surv}_{m_2} = \{n_\perp^{\texttt{null}}\} \cup (\cup_{P_{m_2}[i]=\texttt{bipush}}\{n^c_i\}\emptyset) \cup G^n_{m_2} \cup G^m_{m_2}.$$

Note that the definition of $G^m_{m_1}$ depends on $G^{Surv}_{m_2}$ which depends on $G^m_{m_2}$. In the case when m_1 and m_2 are mutually recursive methods, the definition of $G^m_{m_2}$ will also depend on $G^m_{m_1}$, hence to a loop of recursive definitions. This is the reason why we treat mutually recursive methods separately.

Mutually recursive methods A challenge arises from the renaming of nodes created in recursive methods or in inter-dependent methods. To solve this problem and avoid infinite loops, we propose a solution based on automata building starting from the call graph of the recursive method. We aim at targeting open systems and, as a consequence, we do not rely on the call graph of a particular application. However, to be able to analyse inter-dependent methods, we need to have access to their code (or at least to their AMG). Thus, for a method m, we can consider the set $InterDep(m)$ which is the minimal set of inter-dependent methods containing m_1. Then, for each set of methods $M = InterDep(m_1)$ for some m_1, we build an automaton $\mathcal{A}^M = (S, \Sigma, \delta, m_1, F)$ such that:

$$\begin{array}{rcl}
F & = & \{n_i^c \mid \exists m' \in M, n_i^c \in V_{m'}^{init}\} \\
 & \cup & \{n \mid \exists m' \in M, n \in V \text{ with } G_i^n = (V, E \cup E^c) \in G_{m'}^{init}\} \\
S & = & M \cup F \\
\Sigma & = & \{n_i^m \mid \exists m_1, m_2 \in M, P_{m_1}[i] = \texttt{invoke}\ \ m_2\} \\
 & \cup & \{n_i^c \mid \exists m' \in M, n_i^c \in V_{m'}^{init}\} \\
 & \cup & \{n_i^n \mid \exists m' \in M, n_i^n \in V_{m'}^{init}\} \\
 & \cup & \{f \mid \exists m' \in M, n_i^n.f \in \mathcal{V}(V_{m'}^{init}\}) \\
\delta & = & \{(m_1, m_2, n_i^m) \mid m_1 \in M, m_2 \in M, P_{m_1}[i] = \texttt{invoke}\ m_2\} \\
 & \cup & \{(m', n_i^c, n_i^c) \mid m' \in M, n_i^c \in V_{m'}^{init}\} \\
 & \cup & \{(m', n_i^n, n_i^n) \mid m' \in M, n_i^n \in V_{m'}^{init}\} \\
 & \cup & \{(n_i^n, n_i^n.f, f) \mid m' \in M, n_i^n \in V_{m'}^{init}, n_i^n.f \in \mathcal{V}(V_{m'}^{init})\}
\end{array}$$

where $G_{m'}^{init} = (V_{m'}^{init}, E_{m'}^{init})$
The set of states of the automaton, S, is represented by methods in M and by surviving nodes in M, denoted by the set F (*i.e.*, constant nodes and inside nodes). The initial state is the method m_1. The transition function δ creates

- a transition from m_1 to m_2, labeled with n_i^m, if m_1 invokes m_2 at instruction i,
- a transition from m' to n_i^c/n_i^n, labeled with n_i^c/n_i^n, if method m' creates a constant/reference at instruction i,
- a transition from n_i^n to $n_i^n.f$, labeled with f, if method m' creates a reference at instruction i and f is a primitive field.

Hence, the alphabet Σ of the automaton is given by the labels described above.

Using this automaton, we can define $G_{m_1,m_2,pc}^m = (V, E \cup E^c)$, for the case when $m_2 \in InterDep(m_1)$:

$$\begin{array}{rcl}
V & = & \{l_0.l_1 \ldots l_k \mid \delta(m_1, l_0.l_1 \ldots l_k) = u \text{ with } l_0 = n_{pc}^m, u \in F \text{ and } \forall i \neq j \Rightarrow l_i \neq l_j\} \\
 & \cup & \{n_\perp^{\texttt{null}}\} \\
E & = & \{(u_0 \ldots u_k, u_0 \ldots u_k.f, \langle f, t\rangle) \mid u_0 \ldots u_k, u_0 \ldots u_k.f \in V, u_{k-1} = m', \exists G_i^n \in G_{m'}^{init} \\
 & & \text{and } (u_k, f, \langle f, t\rangle) \in V_i^n\} \\
 & \cup & \{(u_0 \ldots u_k, n_\perp^{\texttt{null}}, \langle f, t\rangle) \mid u_0 \ldots u_k \in V, u_{k-1} = m', \exists G_i^n \in G_{m'}^{init} \\
 & & \text{and } (u_k, n_\perp^{\texttt{null}}, \langle f, t\rangle) \in V_i^n\} \\
E^c & = & \emptyset
\end{array}$$

The automaton builds nodes based on labels; the loops are eliminated by only considering paths containing distinguished nodes. Recall that, in the inter-procedural analysis, pc refers to the fully qualified name of the method. This allows us to distinguish calling sites at the same program counter, but in different methods.

This procedure allows us to correctly rename inside nodes and their primitive fields: not only the nodes are renamed and added, but also edges between inside nodes and primitive fields, and inside nodes and $n_\perp^{\texttt{null}}$.

(a) Source code for m_1

```
m1
 ...
 i: invoke m2
 j: invoke m3
  ...
```

(b) Source code for m_2

```
m2
 ...
 i': new A
  ...
```

(c) Source code for m_3

```
m3
 ...
 k: invoke m4
 k': bipush
  ...
```

(d) Source code for m_4

```
m4
  ...
 l: invoke m1
 l': new A
  ...
```

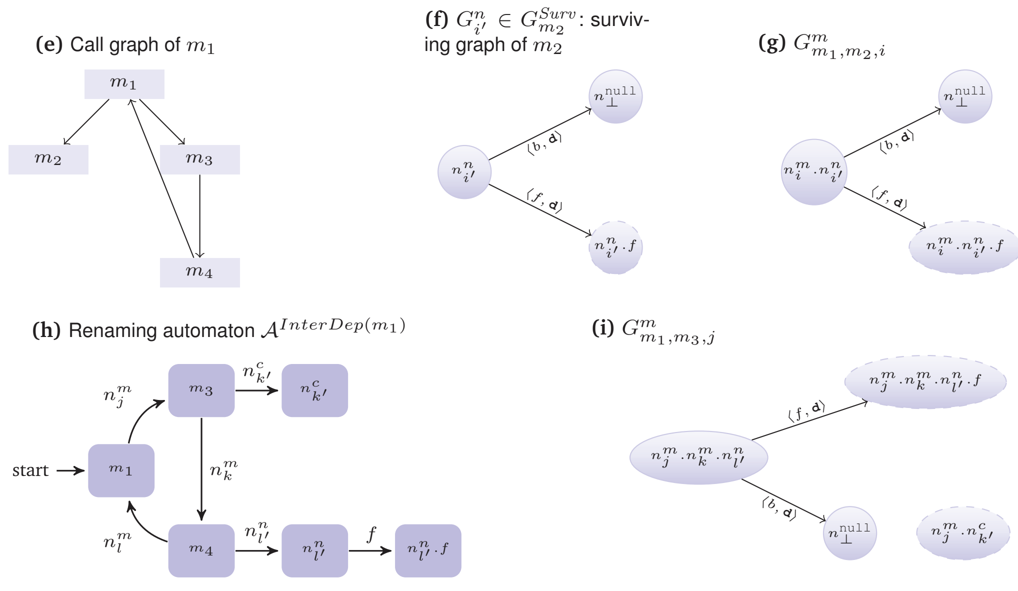


Figure 5.9: Renaming example

Renaming function

Based on the automaton, we can extend the function ren to deal with renaming of nodes from mutually recursive methods:

$$ren_{m_1,m_2}(u, pc) = \begin{cases} n^{\text{null}}_\perp & \text{if } u = n^{\text{null}}_\perp \\ n^m_{pc}.u & \text{if } m_2 \notin InterDep(m_1) \\ n^m_{pc}.v & \text{if } m_2 \in InterDep(m_1) \text{ and } \exists w \text{ s.t. } w.n^m_{pc}.v = u \end{cases}$$

Hence, for mutually recursive methods, ren detects loops and eliminates them by cutting them (*i.e.*, the node $n^m_{pc}.w.n^m_{pc}.v$ is beeing cutted to $n^m_{pc}.v$).

Example 5.2. Let us consider the example in Figure 5.9 and the call graph of Figure 5.9e for a method m_1. The graph $G^m_{m_1} \subset G^{init}_{m_1}$ of nodes created by methods invoked in m_1 is computed as:

$$G^m_{m_1} = G^m_{m_1,m_2,i} \cup G^m_{m_1,m_3,j}.$$

The set of methods which mutually depend on m_1 is $InterDep(m_1) = \{m_1, m_3, m_4\}$.

As m_2 is not an interdependent method, $G^m_{m_1,m_2,i}$ is computed by renaming nodes in $G^{Surv}_{m_2}$ (depicted in Figure 5.9f) and translating edges between renamed nodes. Every non null node of $G^{Surv}_{m_2}$ is prefixed with n^m_i; the result, $G^m_{m_1,m_2,i}$, is depicted in Figure 5.9g.

For the inter-dependent methods, we build the the automaton $\mathcal{A}^{InterDep(m_1)}$ in Figure 5.9h. The graph $G^m_{m_1,m_3,j}$, depicted in Figure 5.9i, is computed according to this automaton; labels on loop-free paths, starting from m_1 and ending in nodes in $F = \{n^c_{k'}, n^n_{l'}, n^n_{l'}.f\}$ result in the nodes to be added: $n^m_j.n^m_k.n^n_{l'}$, $n^m_j.n^m_k.n^n_{l'}.f$ and $n^m_j.n^c_{k'}$.

5.4.2 Abstract semantics

Having defined the complete initial state, we can now define the AMG of a method as the result of our abstract interpretation, starting from the initial state:

Definition 5.6 (AMG of a method)**.** The AMG of a method m, *i.e.*, $\Theta(m)$ is the simplified AMG in the final state $Q_f = (\rho_f, s_f, G_f)$ obtained after solving the equations system (5.3.1) with $(Q^{init}_m, \Gamma^{init}_m)$ as initial state:

$$\Theta(m) = sim(G_f).$$

The function sim removes unnecessary nodes in G_f.

The analysis simplifies the resulting AMG by removing unreachable nodes and edges; in other words, nodes and edges which are not critical for the correctness and which are not necessary for the AMG of the method. We discuss this simplification in Section 5.4.6, page 129.

Invoke The result of the abstract execution of an `invoke` bytecode is the composition of the AMG G_1 before invocation and the AMG $\Theta(m_2)$ of the invoked method, based on an initial mapping relation $\sim_\iota$, which puts in relation actual arguments $(a_{n_{m_2}} :: \cdots :: a_0)$ to parameters of m_2. The abstract semantics rule for the method invocation is:

$$\frac{PC_{m_1}[pc] = \texttt{invoke}\ \ m_2 \qquad Q = (\rho, a_{n_{m_2}} :: \cdots :: a_0 :: s, G_1) \qquad G_2 = \Theta(m_2)}{Q' = (\rho, map_{\sim_\iota}(ret(G_2)) :: s, G_1 \oplus_{\sim_\iota} G_2)}$$

The graph composition operation is performed by the $\oplus$ operator, starting from an initial mapping relation $\sim_\iota$. Note that the mapping relation $\sim_\iota$ depends on the calling context, (Q, Γ) and on the program counter pc; a correct definition of $\sim_\iota$ would have the form $\sim_\iota^{Q,\Gamma,pc}$ complicating readability. For simplification, the mapping relation is denoted by $\sim_\iota$.

If the exact type of a_0 is not statically known, it is impossible to find which of the possible callees is called in a specific execution of the `invoke` instruction. To deal with such a situation, we conservatively join the AMGs computed for all possible callees.

Besides the graph composition, the `invoke` instruction pushes nodes on the stack mapped to nodes returned by the invoked method, denoted by $map_{\sim_\iota}(ret(G_2))$. Intuitively, the set $ret(G_2)$ contains nodes returned by m_2 (*i.e.*, fields of node $n^r_\top$ in G_2), while the function $map_{\sim_\iota}(u)$ returns nodes mapped to nodes in $ret(G_2)$. We precisely define these functions later, after the definition of the mapping relation.

Return To indicate the return of the method we use a dummy node, $n^r_\top$, with a single field, r. Nodes returned by the method through α`return` bytecode are fields of $n^r_\top$. The abstract semantics rule of the α`return` instruction is:

$$\frac{(\rho, u :: s, (V, E \cup E^c))}{(\rho, \epsilon, (V, E \cup (E^c \cup \{(n^r_\top, e, \langle r, t\rangle) | (e, t) \in u\} \cup \{(n^r_\top, e', \langle r, \mathbf{i}\rangle) | e' \in \Gamma\}))} \alpha\texttt{return}$$

The method returns elements on the top of the stack (e, such that $(e, t) \in u$) and elements in the context Γ ($e' \in \Gamma$), as the return may depend on the control flow of the method.

AMGs composition outline The composition of two AMGs is performed by the $\oplus$ operator and it has two steps:

1. First, the analysis computes a mapping function $\sim$: $Nodes \times Nodes \times \mathcal{F}$ between nodes of G_1 and nodes of the AMG of the invoked method G_2.

2. Next, the analysis uses the mapping function $\sim$ to combine the two AMGs, G_1 and G_2. The composition translates edges between two nodes in G_2 to edges between mapped nodes in G_1.

In the next paragraphs we present each step in detail. Subsequently, we prove that method invocation instructions, `invoke` and α`return`, fit into the monotone data flow framework. Finally we show how an AMG can be simplified to keep only relevant information and we discuss native methods and inheritance issues.

5.4.3 The mapping relation

In the context of graph composition, the mapping relation is needed to put in relation *(i)* initially, actual arguments and formal parameters, and *(ii)* the rest of the nodes starting from the initial mapping. We first informally describe the meaning of a mapping relation and afterwards we define it formally in the context of method invocation.

A mapping relation $\sim$: $Nodes \times Nodes \times \mathcal{F}$ between the two AMGs G_1 and G_2 puts in relation the nodes from the AMG of the caller, G_1, to nodes from the AMG of the callee, G_2. More exactly, $\sim$ puts in relation nodes from the final graph to nodes from G_2, but as we assume that all nodes are known from the start, the nodes of G_1 and nodes of the final graph are identical.

Mapping relation definition As the graph is enriched with implicit flows and the edges are labeled with the field and the type of the flow, the mapping relation must also take into account the the type of flow (elements in $\mathcal{F}$). We recall that the set of flows is defined as $\mathcal{F} = \{\mathtt{d}, \mathtt{i}\}$. Hence, given two graphs $G_1 = (V_1, E_1 \cup E_1^c)$ and $G_2 = (V_2, E_2 \cup E_2^c)$, the mapping relation $\sim$ is defined on $V_1 \times V_2 \times \mathcal{F}$. Let $v_1 \in V_1$, $v_2 \in V_2$ and $t \in \mathcal{F}$. If $(v_1, v_2, t) \in \sim$, we read v_1 *maps to* v_2 *through a flow of type* t. For convenience, we use the equivalent notation $v_1 \overset{t}{\sim} v_2$.

The relation $\overset{\mathtt{d}}{\sim}$ (or $\sim \cap V_1 \times V_2 \times \{\mathtt{d}\}$) maps the nodes without taking into account the implicit flow. This relation is possible only between nodes of the same type (reference or primitive). If $a \overset{\mathtt{d}}{\sim} b$, either both a and b are primitive nodes ($a \in \mathcal{V}(V_1)$ and $b \in \mathcal{V}(V_2)$), or they are reference nodes ($a \in \mathcal{O}(V_1)$ and $b \in \mathcal{O}(V_2)$).

The relation $\overset{\mathtt{i}}{\sim}$ (or $\sim \cap V_1 \times V_2 \times \{\mathtt{i}\}$) adds support for implicit flow. In contrast to $\overset{\mathtt{d}}{\sim}$, the relation $\overset{\mathtt{i}}{\sim}$ is not necessary between nodes of the same type. If $a \overset{\mathtt{i}}{\sim} b$, then

- either a is a primitive node ($a \in \mathcal{V}(V_1)$) and b is a reference node ($b \in \mathcal{O}(V_2)$); in this case, nodes related to b through direct flow implicitly depend on a (if $a' \overset{\mathtt{d}}{\sim} b$, then a' depends on a through implicit flow); this case is illustrated by rule 2b of Definition 5.9 (Graph composition): each time the composition adds an edge from a' labeled with $\langle f, \mathtt{d}\rangle$ (which corresponds to modifying a'), an edge from a' to a labeled with $\langle f, \mathtt{i}\rangle$ is also added;

- or both a and b are primitive nodes ($a \in \mathcal{V}(V_1)$ and $b \in \mathcal{V}(V_2)$); for example, $n_{\perp}^{cxt} \overset{\mathtt{i}}{\sim} n_{\perp}^{cxt}$; this means that for all nodes b' that implicitly depend on b, then all nodes mapped to b' implicitly depend on a.

The relation $a \overset{\mathtt{i}}{\sim} b$ with a being a reference node in $\mathcal{O}(V_1)$ is impossible, as the analysis generates implicit dependencies only on primitive nodes (according to AMGs properties stated in Section 5.2.5, page 107).

Now that we explained the intuition behind the mapping relation, we define it formally.

Construction of mapping relation The construction of the mapping relation starts from an initial relation; next, the relation is extended by matching edges (cf. constraints in Definition 5.8).

For `invoke` bytecode, the initial relation $\sim_\iota$ maps the actual arguments to formal parameters of the called method.

Definition 5.7 (Initial mapping relation for `invoke`)**.** If $Q = (\rho, a_{n_{m_2}} :: \cdots :: a_0 :: s, G_1 = (V_1, E_1 \cup E_1^c))$ and Γ is the abstract state before the invocation of method m_2 at instruction pc in m_1, and $G_2 = (V_2, E_2 \cup E_2^c)$ is the AMG of the called method, then the initial relation $\sim_\iota$ is defined as follows:

$$\begin{aligned}
\sim_\iota : V_1 &\rightarrow V_2 \times \mathcal{F} & \\
u &\overset{t}{\sim}_\iota n_j^p \text{ if } (u,t) \in a_j & (5.4.1)\\
u &\overset{\mathtt{d}}{\sim}_\iota v \text{ if } u = ren_{m_1,m_2}(v, pc) & (5.4.2)\\
u &\overset{\mathtt{i}}{\sim}_\iota n_\perp^{cxt} \text{ if } u \in \Gamma & (5.4.3)\\
n_\perp^{\mathtt{null}} &\overset{\mathtt{d}}{\sim}_\iota n_\perp^{\mathtt{null}}. & (5.4.4)
\end{aligned}$$

Constraint 5.4.1 puts in relation the actual arguments ($a_{n_{m_2}} :: \cdots :: a_0$) to formal parameters in G_2 ($n_{m_2}^p :: \cdots :: n_0^p$). Constraint (5.4.2) maps inside nodes of G_2 to corresponding renamed nodes. Constraint (5.4.3) maps nodes in the calling context Γ to the node $n_\perp^{cxt}$, while the last constraint (5.4.4) maps the unique null node.

Based on an initial relation, we now define the *closure* of a mapping relation between two AMGs G_1 and G_2. The closure extends the initial mapping relation by relating nodes based on the AMGs (*e.g.*, the fields f of two related nodes are also related).

Definition 5.8 (Closure of a mapping relation)**.** For two AMGs, $G_1 = (V_1, E_1 \cup E_1^c)$ and $G_2 = (V_2, E_2 \cup E_2^c)$, and a given initial relation $\sim_\iota \subseteq V_1 \times V_2 \times \mathcal{F}$ we define the mapping relation $\sim_\iota^{G_1,G_2} \subseteq V_1 \times V_2 \times \mathcal{F}$ as the closure of $\sim_\iota$ on G_1 and G_2 and thus as the least relation such that :

1. $\sim_\iota \subseteq \sim_\iota^{G_1,G_2}$,
2. If $v_1 \overset{\mathtt{d}}{\sim}_\iota^{G_1,G_2} v_2$, $(v_1, v_1', \langle f, t\rangle) \in E_1 \cup E_1^c$, and $(v_2, v_2', \langle f, t\rangle) \in E_2 \cup E_2^c$ then $v_1' \overset{t}{\sim}_\iota^{G_1,G_2} v_2'$,
3. If $v_1 \overset{\mathtt{d}}{\sim}_\iota^{G_1,G_2} v_2$, $v_1 \overset{\mathtt{d}}{\sim}_\iota^{G_1,G_2} w_2$, $(v_2, v_2', \langle f, \mathtt{d}\rangle) \in E_2 \cup E_2^c$, $(w_2, w_2', \langle f, \mathtt{d}\rangle) \in E_2 \cup E_2^c$ and $v_1' \overset{\mathtt{d}}{\sim}_\iota^{G_1,G_2} v_2'$ then $v_1' \overset{\mathtt{d}}{\sim}_\iota^{G_1,G_2} w_2'$,
4. If $(u, v_1, \langle f, \mathtt{d}\rangle) \in E_1 \cup E_1^c$, $(u, v_1', \langle f, \mathtt{i}\rangle) \in E_1 \cup E_1^c$ and $v_1 \overset{\mathtt{d}}{\sim}_\iota^{G_1,G_2} w$ then $v_1' \overset{\mathtt{i}}{\sim}_\iota^{G_1,G_2} w$,
5. If $v_1 \overset{\mathtt{i}}{\sim}_\iota^{G_1,G_2} v_2$, $(v_2, v_2', \langle f, \mathtt{d}\rangle) \in E_2 \cup E_2^c$, then $v_1 \overset{\mathtt{i}}{\sim}_\iota^{G_1,G_2} v_2'$.

The first constraint of the definition initializes the relation $\sim_\iota^{G_1,G_2}$, while the others extend it by matching edges. As more and more mappings are discovered, the mapping function goes deeper

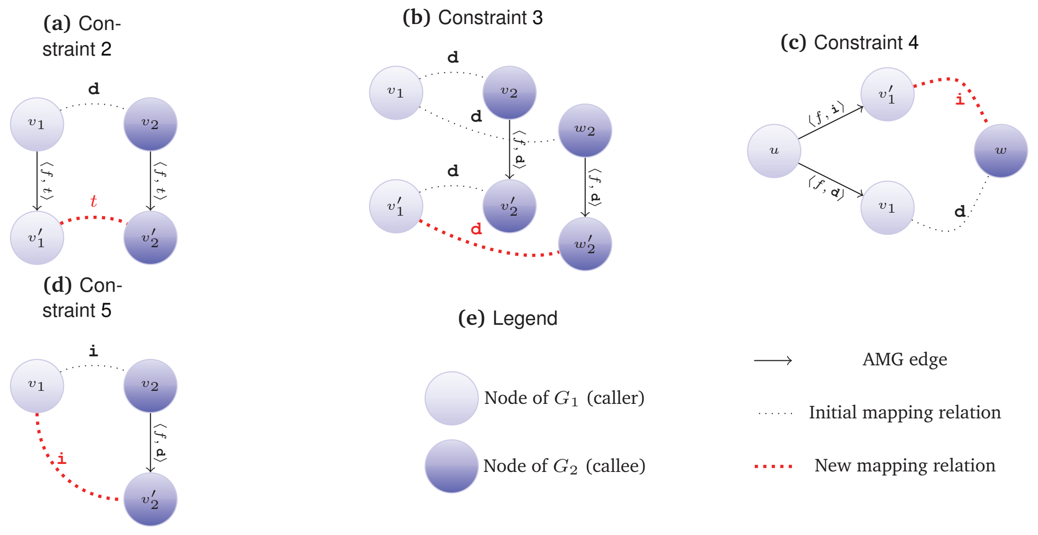


Figure 5.10: Closure of a mapping relation

and deeper into the AMGs, and more matching edges can be applied. Accordingly, Constraints 2, 3, 4 and 5 are repeatedly applied until no further progress is possible, and a fixed point is reached.

The constraints are depicted in Figure 5.10. The second constraint puts in relation the same fields of two nodes which are already related (Figure 5.10a). The third constraint is needed (Figure 5.10b) to solve well known aliasing problems. In a nutshell, it detects relations between aliased nodes; if two nodes v_2 and w_2 are related to the same node v_1, then they might be aliased nodes, hence fields f of v_2 and w_2 might also be aliased. As a consequence, nodes related to field f of v_2 must also be related to fields f of w_2. A detailed example showing the utility of this constraint is presented in Appendix A.2, at page 165. The fourth constraint (Figure 5.10c) is needed in order to correctly take into account the implicit flow. The last rule (Figure 5.10d) propagates the implicit dependency: if a node implicitly depends on a node v_1, then all its fields, fields of fields, etc. also implicitly depend on v_1.

(a) Source code

```
class B { int g;}

class A { B b;
   void m1(int p1) {
       ...
   i:  iload 0
   j:  invoke m2
       ...
   }

   void m2(){
      ...
   }
}
```

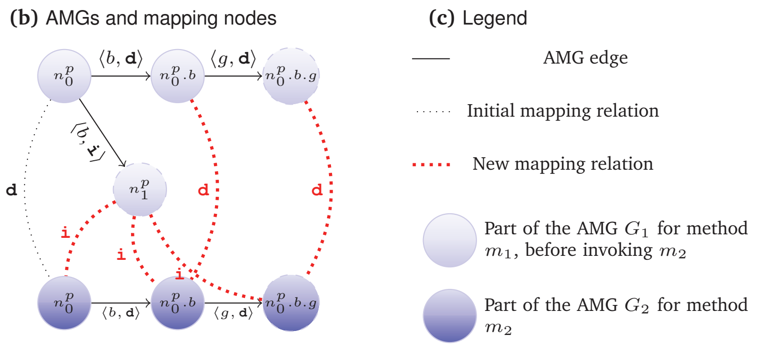


Figure 5.11: Mapping example

Example 5.3. Let us consider the example in Figure 5.11. The initial mapping relation, $\sim_\iota$, maps the actual argument with which m_1 invokes m_2 (`this` or n_0^p) to the first parameter of m_2 (n_0^p):

$n_0^p \overset{\mathtt{d}}{\sim}_\iota n_0^p$. By applying Constraint 2, which puts in relation the same fields of mapped nodes, we obtain: $n_0^p.b \overset{\mathtt{d}}{\sim}_\iota n_0^p.b$ and $n_0^p.b.g \overset{\mathtt{d}}{\sim}_\iota n_0^p.b.g$.

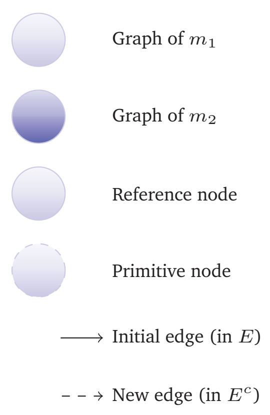
In the AMG of method m_1, the field b of node n_0^p implicitly depends on n_1^p. Hence, all subsequent changes to the nodes related to the field b implicitly depend on n_1^p. Our mapping takes into consideration this reasoning in two steps: first, it puts in relation the nodes n_1^p to n_0^p, by applying Constraint 4: $n_1^p \overset{\mathtt{i}}{\sim}_\iota n_0^p$; second, it will map n_1^p to all fields of n_0^p, according to Constraint 5. The second step leads to the mappings: $n_1^p \overset{\mathtt{i}}{\sim}_\iota n_0^p.b$ and $n_1^p \overset{\mathtt{i}}{\sim}_\iota n_0^p.b.g$.

(a) Source code

```
class B { int g;}

class A {
   B b;

   void m1(int p1,B p2){
      if(p1 != 0)
         this.b = p2;
      this.m2();
   }

   void m2() {
      this.b.g = 5;
   }

...
}
```

```
m1
   0: iload 1
   1: ifeq 5
   2: aload 0
   3: iload 2
   4: putfield b_A
   5: aload 0
   6: invoke m2

m2
   0: aload 0
   1: getfield b_A
   2: bipush 5
   3: putfield g_B
```

(b) Legend



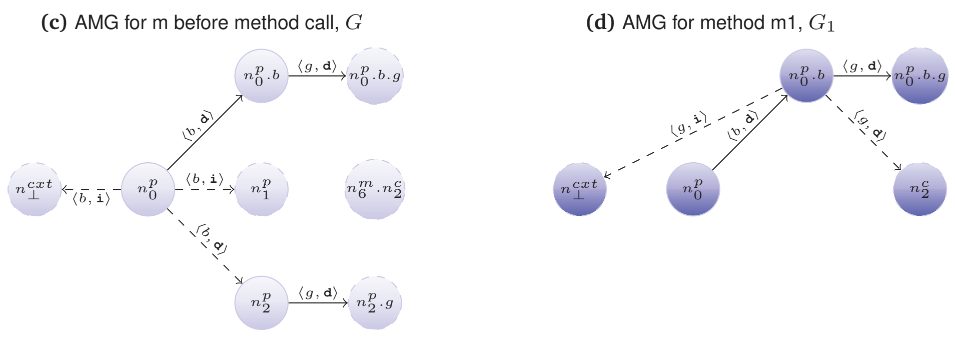

(c) AMG for m before method call, G
$n_0^p.b$
$\langle g, \mathtt{d}\rangle$
$n_0^p.b.g$
$\langle b, \mathtt{d}\rangle$
$n_\perp^{cxt}$
$\langle b, \mathtt{i}\rangle$
n_0^p
$\langle b, \mathtt{i}\rangle$
n_1^p
$n_6^m.n_2^c$
$\langle b, \mathtt{d}\rangle$
n_2^p
$\langle g, \mathtt{d}\rangle$
$n_2^p.g$
(d) AMG for method m1, G_1
$n_0^p.b$
$\langle g, \mathtt{d}\rangle$
$n_0^p.b.g$
$\langle g, \mathtt{i}\rangle$
$\langle b, \mathtt{d}\rangle$
$\langle g, \mathtt{d}\rangle$
$n_\perp^{cxt}$
n_0^p
n_2^c

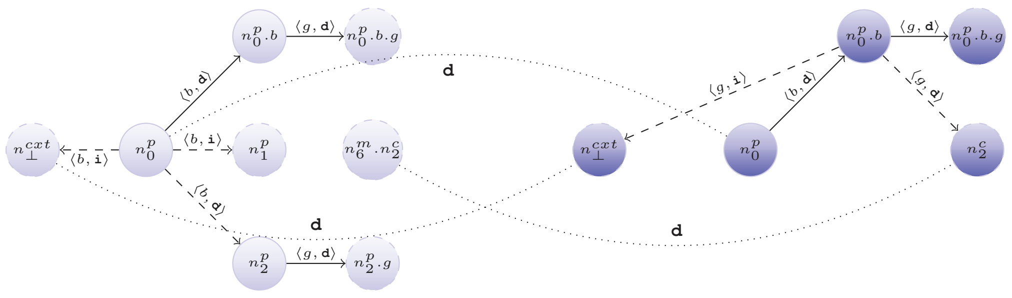
Figure 5.12: AMGs for Example 5.4

Example 5.4. Let us consider the example in Figure 5.12. Method m_1 in class A calls method m_2. Let $G_1 = (V_1, E_1 \cup E_1^c)$ be the graph of m_1 before the call (depicted in Figure 5.12c) and $G_2 = (V_2, E_2 \cup E_2^c)$ be the graph of m_2 (Figure 5.12d). The call from instruction 7 invokes the method m_2 with the argument n_0^p (`this`), hence the initial mapping $\sim_\iota : V_1 \times V_2 \times \mathcal{F}$ is defined as:

$$\sim_\iota : \quad \begin{array}{rcl} n_0^p & \overset{\mathtt{d}}{\sim}_\iota & n_0^p \\ n_\perp^{cxt} & \overset{\mathtt{i}}{\sim}_\iota & n_\perp^{cxt} \\ n_6^m.n_2^c & \overset{\mathtt{d}}{\sim}_\iota & n_2^c. \end{array}$$

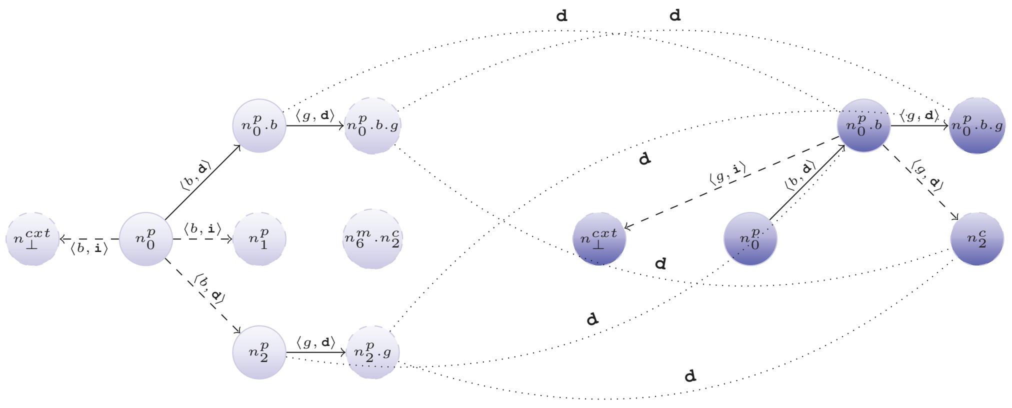
The initial mapping puts in relation the actual arguments to the first parameter in m_2 (n_0^p), and the context nodes. The node $n_6^m.n_2^c$ represents the renamed constant node n_2^c, created by instruction

(a) Node mapping for the initial relation $\sim_\iota$



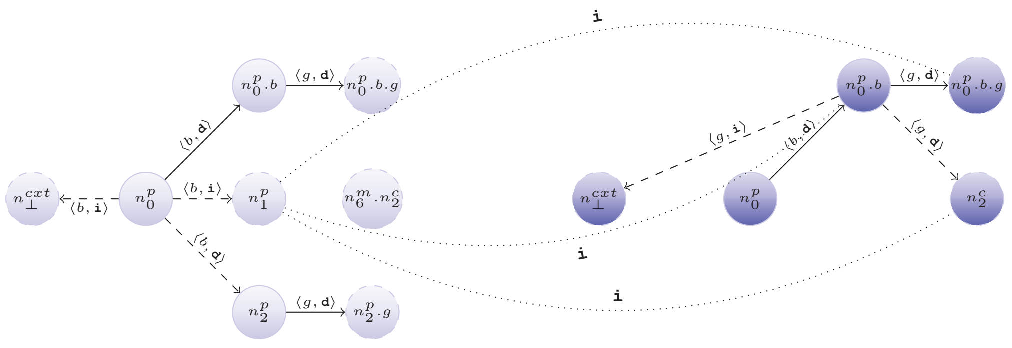
$n_\perp^{ext}$
$\langle b, \mathtt{i}\rangle$
n_0^p
$\langle b, \mathtt{d}\rangle$
$n_0^p.b$
$\langle g, \mathtt{d}\rangle$
$n_0^p.b.g$
$\langle b, \mathtt{i}\rangle$
n_1^p
$n_6^m.n_2^c$
$\langle b, \mathtt{d}\rangle$
n_2^p
$\langle g, \mathtt{d}\rangle$
$n_2^p.g$
d
d
d
$n_\perp^{ext}$
$\langle g, \mathtt{i}\rangle$
n_0^p
$\langle b, \mathtt{d}\rangle$
$n_0^p.b$
$\langle g, \mathtt{d}\rangle$
$n_0^p.b.g$
$\langle g, \mathtt{d}\rangle$
n_2^c

(b) Node mapping added by Constraint 2 of mapping closure (Definition 5.8)



d
d
$n_0^p.b$
$\langle g, \mathtt{d}\rangle$
$n_0^p.b.g$
$n_\perp^{ext}$
$\langle b, \mathtt{i}\rangle$
n_0^p
$\langle b, \mathtt{d}\rangle$
$\langle b, \mathtt{i}\rangle$
n_1^p
$n_6^m.n_2^c$
$\langle b, \mathtt{d}\rangle$
n_2^p
$\langle g, \mathtt{d}\rangle$
$n_2^p.g$
d
d
d
d
$n_\perp^{ext}$
$\langle g, \mathtt{i}\rangle$
n_0^p
$\langle b, \mathtt{d}\rangle$
$n_0^p.b$
$\langle g, \mathtt{d}\rangle$
$n_0^p.b.g$
$\langle g, \mathtt{d}\rangle$
n_2^c

(c) Node mapping added by Constraints 4 and 5 of mapping closure (Definition 5.8)



i
$n_0^p.b$
$\langle g, \mathtt{d}\rangle$
$n_0^p.b.g$
$n_\perp^{ext}$
$\langle b, \mathtt{i}\rangle$
n_0^p
$\langle b, \mathtt{d}\rangle$
$\langle b, \mathtt{i}\rangle$
n_1^p
$n_6^m.n_2^c$
$\langle b, \mathtt{d}\rangle$
n_2^p
$\langle g, \mathtt{d}\rangle$
$n_2^p.g$
i
i
$n_\perp^{ext}$
$\langle g, \mathtt{i}\rangle$
n_0^p
$\langle b, \mathtt{d}\rangle$
$n_0^p.b$
$\langle g, \mathtt{d}\rangle$
$n_0^p.b.g$
$\langle g, \mathtt{d}\rangle$
n_2^c

Figure 5.13: Mapping relation for Example 5.4

`bipush` at instruction `2` in method m_2 (called at instruction `6`). Obviously, $n_6^m.n_2^c$ is related with the node n_2^c in m_2. The initial mapping relations are depicted in Figure 5.13a by dotted edges.

Next, we compute the closure of initial relation $\sim_\iota$, $\sim_\iota^{G,G_1}$. By applying Constraint 2 of Definition 5.8, we map the fields of nodes which are already related; as $n_0^p \overset{\mathbf{d}}{\sim}_\iota n_0^p$, we map the field b, which leads to mapping relations 5.4.5 and 5.4.6 (Figure 5.13b). As $n_0^p.b \overset{\mathbf{d}}{\sim} n_0^p.b$, we can map their fields g and obtain the mapping relations 5.4.7 and 5.4.8. Similarly, we map the field g of n_2^p and

$n_0^p.b$ and obtain the mapping relations 5.4.9 and 5.4.10.

$$n_0^p \overset{\mathtt{d}}{\sim}_\iota n_0^p \quad\Rightarrow\quad n_0^p.b \overset{\mathtt{d}}{\sim} n_0^p.b \tag{5.4.5}$$

$$n_2^p \overset{\mathtt{d}}{\sim} n_0^p.b \tag{5.4.6}$$

$$n_0^p.b \overset{\mathtt{d}}{\sim} n_0^p.b \quad\Rightarrow\quad n_0^p.b.g \overset{\mathtt{d}}{\sim} n_0^p.b.g \tag{5.4.7}$$

$$n_0^p.b.g \overset{\mathtt{d}}{\sim} n_2^c \tag{5.4.8}$$

$$n_2^p \overset{\mathtt{d}}{\sim} n_0^p.b \quad\Rightarrow\quad n_2^p.g \overset{\mathtt{d}}{\sim} n_0^p.b.g \tag{5.4.9}$$

$$n_2^p.g \overset{\mathtt{d}}{\sim} n_2^c \tag{5.4.10}$$

The edge $(n_0^p, n_1^p, \langle b, \mathtt{i}\rangle)$ in the graph G_1 of m_1 illustrates that field b of n_0^p implicitly depends on n_1^p. Hence, all changes to field b of n_0^p (and fields of fields etc.) also implicitly depend on n_1^p (Figure 5.13c). To reflect this, Constraint 4 puts in relation n_1^p with all nodes in G_2 related, through direct flow, to fields b of n_0^p ($n_0^p.b$ and n_2^p). As $n_0^p.b \overset{\mathtt{d}}{\sim} n_0^p.b$, then

$$n_1^p \overset{\mathtt{i}}{\sim} n_0^p.b. \tag{5.4.11}$$

In the same way, $n_2^p \overset{\mathtt{d}}{\sim} n_0^p.b$, hence we obtain the same relation 5.4.11. We note that the implicit mapping is between a primitive node and a reference node, in contrast to direct mapping which relates nodes of the same type.

Constraint 5 propagates the implicit mapping to fields of fields b of n_0^p etc. Hence, it puts in relation n_1^p with all nodes in G_2 related, through direct flow, to fields of fields b of n_0^p ($n_0^p.b.g$ and $n_2^p.g$). As $n_0^p.b.g \overset{\mathtt{d}}{\sim} n_0^p.b.g$, we obtain the mapping relation 5.4.12, and as $n_0^p.b.g \overset{\mathtt{d}}{\sim} n_2^c$, we obtain the mapping relation 5.4.13.

$$n_1^p \overset{\mathtt{i}}{\sim} n_0^p.b.g \tag{5.4.12}$$

$$n_1^p \overset{\mathtt{i}}{\sim} n_2^c. \tag{5.4.13}$$

Computing the return of the callee The abstract semantics of `invoke` instruction pushes the set $map_{\sim_\iota}(ret(G_2))$ on the stack, representing the nodes mapped to nodes returned by the callee.

The function $ret(G_2)$ contains nodes returned by m_2, *i.e.*, fields of node $n_\top^r$ in G_2. Formally,

$$ret(G_2) = \{(u,t) \mid u \in adj_{G_2}(n_\top^r, \langle r, t\rangle)\}.$$

The function $map_{\sim_\iota}$ returns the nodes mapped to $ret(G_2)$ through the closure of the initial mapping relation, *i.e.*, $\sim_\iota^{G_1,G_2}$. Formally, the mapping of a set of nodes $(v,t) \in e$ through the relation $\sim_\iota^{G_1,G_2}$ is

$$map_{\sim_\iota}(e) = \{(u,t_2) \mid (v,t) \in e \quad\wedge\quad u \overset{t_1}{\sim}_\iota^{G_1,G_2} v \quad\wedge\quad t_2 = t \sqcap t_1\}.$$

The type of the node (u,t_2) is the meet between t and t_1, *i.e.*, $t_2 = t \sqcap t_1$, according to the set of flow types $\mathcal{F} = \{\mathtt{d}, \mathtt{i}\}$ and to the order relation $\mathtt{i} \sqsubseteq \mathtt{d}$. The intuition behind it is that if the flow type resulting from two different flow types is always equal to the smallest (the weakest) of the flows. For example, if $t = \mathtt{d}$ and $t_1 = \mathtt{i}$, then $t = \mathtt{i}$.

5.4.4 Combining two AMGs

Once we have the mapping relation, we combine the AMG for the program point before the `invoke`, G_1, with the AMG of the callee, G_2, to obtain the AMG for the program point right after the `invoke`.

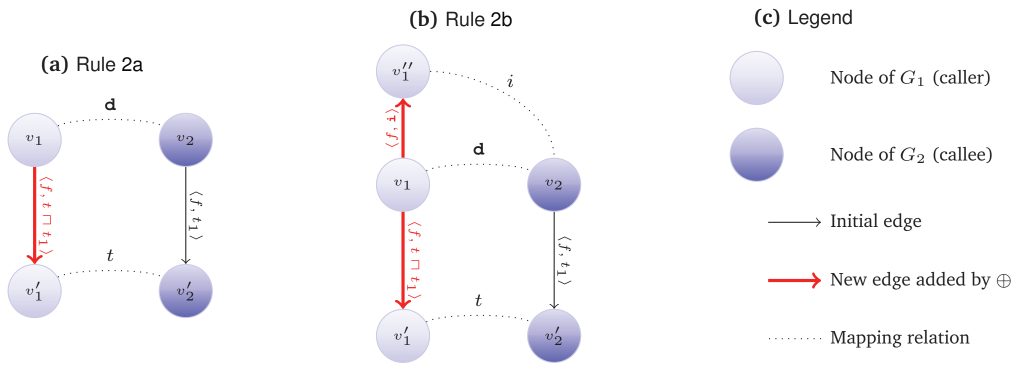


Figure 5.14: Rules for combining two AMGs

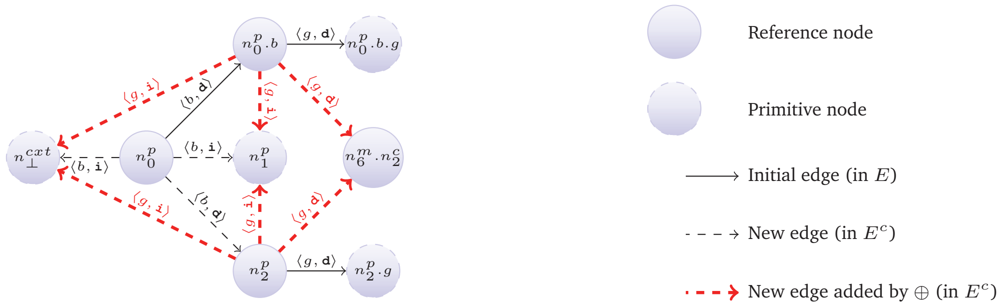


Figure 5.15: Combined AMGs for Example 5.4, $G \oplus_{\sim_\iota} G_1$

The combination is done by the $\oplus_{\sim_\iota}$ operator. Intuitively, $\oplus$ returns the union between G_1 and the projection of G_2 through the mapping $\sim_\iota^{G_1,G_2}$.

Definition 5.9 (Graphs composition)**.** Given two AMGs, $G_1 = (V_1, E_1 \cup E_1^c)$ and $G_2 = (V_2, E_2 \cup E_2^c)$, and an initial relation $\sim_\iota \subseteq V_1 \times V_2 \times \mathcal{F}$, we define $G = (V_1, E_1 \cup E^c) = G_1 \oplus_{\sim_\iota} G_2$ as the least graph such that:

1. $E_1^c \subseteq E^c$,
2. If $v_1 \overset{\mathbf{d}}{\sim}_\iota^{G_1,G_2} v_2$, $v_1' \overset{t}{\sim}_\iota^{G_1,G_2} v_2'$, $v_1 \neq n_\perp^{\mathtt{null}}$ and $(v_2, v_2', \langle f, t_1\rangle) \in E_2^c$
   a) then $(v_1, v_1', \langle f, t \sqcap t_1\rangle) \in E^c$,
   b) if $v_1'' \overset{\mathbf{i}}{\sim}_\iota^{G_1,G_2} v_2$ then $(v_1, v_1'', \langle f, \mathbf{i}\rangle) \in E^c$.

The composition extends G_1 with edges between surviving nodes in G_2. Thus $G_1 \subseteq G$. The first constraint initializes edges in G, while the second one (depicted in Figure 5.14a) translates edges between nodes in G_2 to edges between nodes related in G. The new edge is typed with $t \sqcap t_1$, as the flow type of the new edge is the smallest of the two flow types which generated the edge. The third constraint, depicted in Figure 5.14b, propagates implicit flows: if node v_2 in G_2 is implicitly mapped to v_1'', then all changes to v_2 (all edges created in G_2) implicitly depend on v_1''. Hence, when an edge starting from v_2 is translated to G, an implicit edge to v_1'' must also be translated.

Example 5.5 (Example 5.4 continued)**.** Figure 5.15 shows the result of the graph composition $G_1 \oplus_{\sim_\iota} G_2$ in the context of Example 5.4 (Figure 5.12). The construction of the AMG is straightforward by applying the rules of Definition 5.9. Edges added by the composition are depicted by extra thick dashed lines. We give the added edges and the rules applied to obtain them:

$$\begin{array}{ll}
\text{rule 2a}: & (n_0^p.b, n_6^m.n_2^c, \langle g, \mathbf{d}\rangle) \\
\text{rule 2b}: & (n_0^p.b, n_1^p, \langle g, \mathbf{i}\rangle) \\
\text{rule 2a}: & (n_2^p, n_6^m.n_2^c, \langle g, \mathbf{d}\rangle) \\
\text{rule 2b}: & (n_2^p, n_1^p, \langle g, \mathbf{i}\rangle) \\
\text{rule 2a}: & (n_0^p.b, n_\perp^{cxt}, \langle g, \mathbf{i}\rangle) \\
\text{rule 2a}: & (n_2^p, n_\perp^{cxt}, \langle g, \mathbf{i}\rangle).
\end{array}$$

5.4.5 Compatibility with the monotone framework

In order to keep the compatibility within the monotone framework, we need to show that the transfer rules for `invoke` and α`return` are monotone.

Lemma 5.10 (Monotonicity of transformation functions (cont. Lemma 5.5)). *The transformation rule of* `invoke` *instruction is monotonic both in state Q (the state before the call) and in G_2 (the AMG of the invoked method).*

Proof. The monotonicity in Q is straightforward: starting from a smaller AMG, we obtain a smaller closure for the mapping relation and we add less edges than if we start from a bigger graph. The monotonicity in G_2 is also straightforward: with a smaller graph, we can compute less mapping relations and, hence, we add less edges in the resulted graph (*i.e.*,, $G_1 \oplus_{\sim_\iota} G_2$). □

Lemma 5.11 (Monotonicity of transformation functions (cont. Lemma 5.5)). *The transformation rule of* α`return` *instruction is monotonic w.r.t. ordering relation $\sqsubseteq$.*

Proof. We consider two pairs

$$(Q_1, \Gamma_1) = ((\rho_1, u_1 :: s_1, (V_1, E_1 \cup E_1^c)), \Gamma_1) \qquad (Q_2, \Gamma_2) = ((\rho_2, u_2 :: s_2, (V_2, E_2 \cup E_2^c)), \Gamma_2)$$

from the property space $\mathcal{S}$ such that $(Q_1, \Gamma_1) \sqsubseteq (Q_2, \Gamma_2)$, and let

$$Q_1' = (\rho_1, \epsilon, E_1 \cup (E_1^c \cup E_{u_1} \cup E_{\Gamma_1})) \qquad Q_2' = (\rho_2, \epsilon, E_2 \cup (E_2^c \cup E_{u_2} \cup E_{\Gamma_2}))$$

be the result of applying abstract semantics of α`return` on (Q_1, Γ_1) and (Q_2, Γ_2) respectively, where

$$E_u = \{(n_\top^r, e, \langle r, t\rangle) | (e, t) \in u\} \qquad E_\Gamma = \{(n_\top^r, e', \langle r, \mathbf{i}\rangle) | e' \in \Gamma\}.$$

We need to prove that $Q_1' \sqsubseteq Q_2'$, hence that $E_{u_1} \subseteq E_{u_2}$ and $E_{\Gamma_1} \subseteq E_{\Gamma_2}$.

By hypothesis, $u_1 \subseteq u_2$ holds which immediately leads to $E_{u_1} \subseteq E_{u_2}$. Also by hypothesis, $\Gamma_1 \subseteq \Gamma_2$, hence $E_{\Gamma_1} \subseteq E_{\Gamma_2}$. □

The monotonicity of the `invoke` and α`return` transformation rules guarantees on the one hand the termination of the intra-procedural analysis and on the other hand the termination of inter-procedural analysis (this is due to fact that the rule is monotone in G_2).

5.4.6 AMG simplification

Once the analysis completed, the AMGs can be simplified in order to reduce their size. The simplification removes nodes and edges insignificant for a method (recall that in our analysis all the nodes are computed statically, at the beginning of the analysis, and they are included in the initial graph). Therefore, it must be performed once the inter-procedural analysis has ended. An earlier simplification of the AMG might lead to a loss of nodes which are still needed by the analysis.

We define *nodes significant for a method* as the set of nodes reachable from parameters of the method (hence from the subgraph G_m^p), from static fields (hence from G^s) or from the dummy node $n_\top^r$. The rest of the nodes are local to the method and are invisible to an external caller.

Considering the AMG of a method m, G_m, the simplified graph, $sim(G_m)$ is formally defined as:

$$sim(G_m) = G_m\lfloor G_m^p \cup G^s \rfloor \cup G_m\lfloor n_\top^r \rfloor$$

where $G\lfloor G' \rfloor$ represents the subgraph of G reduced to nodes reachable from nodes in G'. The simplification eliminates nodes allocated in a method and which are solely used for local needs (*e.g.*, to perform some computations).

5.4.7 Discussion

Native methods and I/Os

Native methods are used when a Java application needs to access some platform-dependent features not supported by Java (*e.g.*, the I/Os) or to use a library written in other language. Native methods bring many benefits to Java as well as security concerns. As they are written in other languages than Java, they cannot be analyzed by our model or any other Java security tool. To support them, native methods must be hand-annotated, just as flow signatures, and their AMG will belong to our trusting computing base.

Native methods represent the only place when a read/write to/from an I/O can occur. To model I/Os, we introduce a special node $n_\top^{io}$ and a special field io. Writing a node u into an I/O is represented by an edge $(n_\top^{io}, u, \langle io, t\rangle)$ in the AMG, while reading from an I/O to the field f of a node u is represented by an edge $(u, n_\top^{io}, \langle f, t\rangle)$. For simplicity, we use a single node to abstract all read/write sites; separating different sites is straightforward: the user (who manually defines the AMG of the native method) can add a unique identifier j to each node, *i.e.*, n_j^{io}.

Inheritance and virtual invocation

To deal with virtual calls and to simplify the model and the soundness proof, we chose to conservatively take all AMGs matching the call site. Obviously, we can improve precision by adapting a technique similar to the one used in the embedded model: we perfom an exact type computation, and, if the exact type is known, then the exact AMG is used; if not, a *global AMG* is used. The global AMG is a door to openness and it is computed similarly with the global flow signature: either from the class hierarchy by computing the union of all AMGs of overwriting methods, or manually defined (as for native methods). Later, new overriding methods can be accepted in the system if their AMG is compatible with the global AMG of the class hierarchy to which they belong.

5.5 Analysis applications

The main goal of our analysis was to build a sound framework for secure information flow. Finally, we obtain a more general framework which can successfully be used for other program analysis domains. We now show how our analysis applies to information flow, but also other applications such as points-to and purity analyses.

5.5.1 Secure information flow

Most of the previous works on non-interference require the lattice security model of information flow to be known from the beginning. Our AMG contains points-to and control flow dependency information, without any information concerning security policies. Once computed, the AMG can be labeled with security levels and non-interference can be checked.

Let $(\mathcal{L}, \sqcup, \sqsubseteq)$ be a lattice of security levels; for simplicity, let us consider $\mathcal{L} = \{low, high\}$, $low \sqsubseteq high$. We can now define security levels on types: let $\lambda_t : \mathit{Field} \times \mathit{Class} \to \mathcal{L}$ be a function that associates a security level with a field of a class. Let us consider a method m, its AMG $\Theta(m) = (V, E \cup E^c)$ and the initial subgraph of parameters $G_m^p = (V^p, E^p \cup \emptyset)$. A security function is a function that associates security levels to input values (the primitive nodes) and to return values (primitive nodes reached from the return node $n_\top^r$ and denoted by the set $R^r = \mathit{Reach}_{\Theta(m)}(n_\top^r)$): $\lambda_\iota : \mathcal{V}(V^p \cup R^r) \to \mathcal{L}$ such that if v is the field f of a class C, then $\lambda_\iota(v) = \lambda_t(f, C)$.

Informally, we say that a method is *secure* if it respects the following properties:

1. no confidential data flows to a static field; static fields are considered as having the lowest security level, *i.e.*, $\perp$,

2. values accessible from parameters or return value on paths contain at least a field of higher security level than their own. This corresponds to our convention we used to define the secret/public part of an object in Section 3.2.1, page 28 (Equations 3.2.1 and 3.2.2). In this convention, the secret part of an object contains all access paths that contain at least one secret field. We extend this convention to a general security lattice, $(\mathcal{L}, \sqcup, \sqsubseteq)$. If $\ell \in \mathcal{L}$ is a security level, than the part of an object with security level ℓ contains all access paths to primitive fields that contain at least a field with security level ℓ and all other security levels ℓ' on the path are smaller than ℓ, *i.e.*, $\ell' \sqsubseteq \ell$.

Formally, we define the secure information flow as follows:

Theorem 5.12 (Secure information flow). *Let m be a method, $G_m^p = (V^p, E^p, \emptyset)$ the initial subgraph of its parameters, $\Theta(m) = (V, E \cup E^c)$ its AMG, and λ_ι a security function. Let $R^r = \mathit{Reach}_{\Theta(m)}(n_\top^r)$. m has secure information flow with respect to $\mathcal{L}$ if*

1. *for every node $v \in \mathcal{V}(V^p \cup R^r)$ such that $\perp$[5]$\sqsubset \lambda_\iota(v)$, there is no path $n_C^s \xrightarrow{\langle f_1, t_1 \rangle} o_1 \ldots o_{k-1} \xrightarrow{\langle f_k, t_k \rangle} v$,*

2. *and for every node $v \in \mathcal{V}(V^p \cup R^r)$ and every path $o_0 \xrightarrow{\langle f_1, t_1 \rangle} o_1 \ldots o_{k-1} \xrightarrow{\langle f_k, t_k \rangle} v$, $\exists i$ such that $\lambda_\iota(v) \sqsubseteq \lambda_t(f_i, \mathit{Type}(o_i))$.*

The advantage of our approach is that security annotations must not be known a priori. Changing a security level does not require a new analysis. Note that we can also have more precise policies on instances: if o and o' have the same type, $o.f$ and $o'.f$ can be given different security levels using a function $\lambda_e : E \to \mathcal{L}$ instead of λ_t. It is more precise but requires the user to precise all the policies.

Example 5.6 (Example 5.1 continued). Recall the Example 5.1, page 115; its AMG is depicted in Figure 5.8, page 116. Let us consider a security lattice $\mathcal{L} = \{low, high\}$. For example, we can define two security policies:

- $\lambda_t(sal, Income) = high$ and $\lambda_t(tax, Income) = high$, as depicted in Figure 5.16a,
- $\lambda_t(sal, Income) = high$ and $\lambda_t(tax, Income) = low$, as depicted in Figure 5.16b.

The program is secure w.r.t. the first security policy, but it is unsecure w.r.t. to the second security policy. The insecurity arises from the path $n_0^p \xrightarrow{\langle tax, \mathbf{d} \rangle} n_0^p.sal$, as $\lambda_\iota(n_0^p.sal) = high$ and $\lambda_t(tax, Income) = low$. Hence, a $high$ input value is accessible to the low part of the parameter n_0^p.

[5] $\perp$ designates here the infimum of the security lattice $\mathcal{L}$

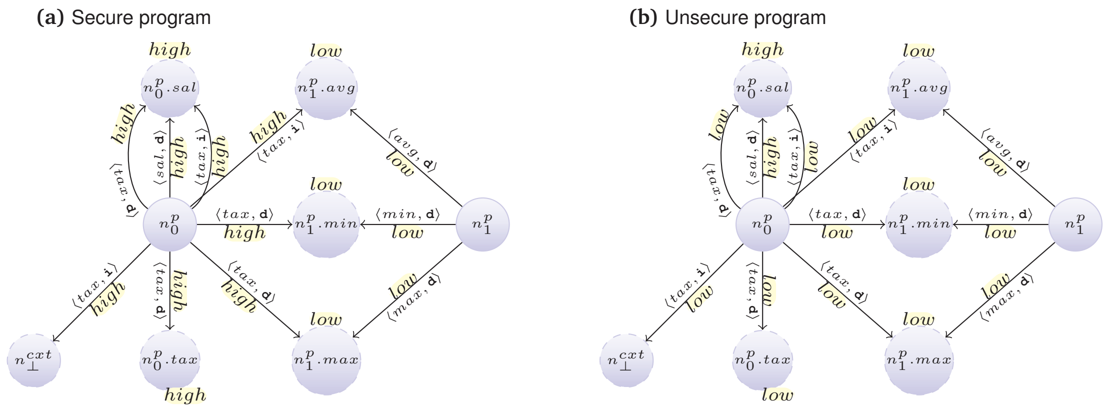


Figure 5.16: AMG of method `tax` labeled with security levels (Example 5.1 cont.)

5.5.2 Points-to analysis

In object oriented languages, like Java [LY99], the analysis of information flow is related to the analysis of references. *Points-to analysis* [And94], or pointer analysis, is a static analysis which computes for every object the set of objects to which it may point at runtime. Points-to analysis has been developed for applications such as escape analysis [SR01, WR99], shape analysis [SRW99] or optimizations (stack allocation, synchronization removal, etc). It gives more accurate memory dependence analysis and data flow analysis. A more precise points-to analysis allows more aggressive optimization and scheduling.

Here, we show how our analysis can be applied for one points-to application, *i.e.*, stack allocation. We believe that our analysis can successfully be applied to other applications and optimizations, but this is out of the scoope of this thesis.

One of the main applications of points-to analysis is *stack allocation* [CGS⁺03]: if an object is not reachable after the end of its allocating method, *i.e.*, the method that allocated it through a `new` instruction, then it can be allocated in the stack frame of the method instead of the heap (which is garbage collected). This optimization has several benefits: less overhead for garbage collection, as the objects allocated in stack are implicitly deallocated when the method's execution ends, and better memory access times.

As our algorithm is a pointer analysis extended with primitive values, using our framework for points-to applications is straightforward.

In our analysis, *stack allocation* techniques can be applied for inside nodes. Given a method m, inside nodes abstract objects allocated in m and have the form n^n_{pc}. An inside node n^n_{pc} captures all objects allocated at site pc. An object can be allocated in the local stack if the inside node abstracting it is not reachable from the parameters of the method, from the return value of the method, or from a static variable. Formally, if G is the AMG of m and pc an allocation site, then n^n_{pc} can be allocated in the stack of m if

$$Reach_G(G^p \cup G^s) \cup Reach_G(\{n^r_\top\}) \cap \{n^n_{pc}\} = \emptyset.$$

Recall that G^p is the initial graph of method parameters while G^s is the graph containing all static fields.

5.5.3 Purity analysis

Purity analysis has become important as the number of software constantly increases. Hence, their verification is crucial in order to ensure correctness. Specification languages [LBR06] use assertions which describe program behaviour or program properties. The use of method calls in such assertions raises several problems: the methods should not modify the objects used in the program, they must only have an observational purpose. Such methods, which have no externally visible side-effects, are called *pure methods*.

We use the same definition of method purity as in JLM (Java Modeling Language) [LBR06] and also used by Salcianu in [Sal06]:

Definition 5.13 (Pure method)**.** A method is *pure* if any of its executions

- does not mutate static fields,
- does not perform I/O operations,
- does not mutate methods parameters.

Note that pure methods allow mutations of newly allocated objects. Moreover, a pure method can invoke an impure method. Sun *et al.* have defined this notion of purity as *observational purity* [BNS04, BNSS06, Sun08]. The definition allows to allocate and throw exceptions; otherwise the definition would be too strict, as almost all JVM instructions can throw runtime exceptions.

In our analysis, method purity can be defined as a property of the AMG of the method. Intuitively, the execution of a method does not mutate static fields and parameters if it does not add edges initiating from static or from parameter nodes. To verify this property, we need the initial graph of the method, before the start of the method analysis. Let G_m^{init} be the initial AMG of a method m. Recall that $G_m^p = (V_m^p, E_m^p \cup \emptyset)$, $G_m^p \subset G_m^{init}$ is the initial subgraph rooted from parameters and $G^s = (V^s, E^s, \emptyset)$, $G^s \subset G_m^{init}$ is the initial static graph of all classes. If G_m is the AMG of m, we can define the purity predicate as follows:

Lemma 5.14 (Purity predicate)**.** *Consider a method m, its AMG G_m. The method is* pure, i.e.*, the predicate $pure(m)$ holds, if*

$$G_m^p = G_m\lfloor G_m^p\rfloor \quad \wedge \quad G^s = G_m\lfloor G^s\rfloor \quad \wedge \quad Reach_{G_m}(n_\top^{io}) \cap (V_m^p \cup V^s) = \emptyset.$$

Hence, a method is pure if the initial subgraphs rooted by parameters and static nodes have not been changed by the execution of the method and if they have not been written to I/Os (reading from I/O leads to a mutation, hence we do not need to check it separately).

Information flow analysis has already been used in purity analysis by Bannet *et al.* [BNSS06]. They state that observational purity is a novel application of information flow analysis in software verification. The problem on observational purity is reduced to a non-interference problem: internal data that the side-effect is on are treated as having security level $high$, while other classes have low security level. If the non-interference holds, *i.e.*, there is no leak of information from $high$ to low, then the internal data do not mutate.

5.6 Conclusion

In this chapter, we showed the definition and construction of the AMGs for JVM methods. The graph contains dependencies between JVM objects and primitive values. The main goal was to build a graph for information flow verification, but, as a result, our construction can be used for many other program analysis techniques, such as points-to and purity analysis. The main contribution of this chapter is the formal definition of inter-procedural analysis, as modularity is an essential requirement in a computing world evolving towards mobility and untrusted code downloading. Moreover, security policies and AMG construction are completely separated: policies are specified and verified *a posteriori*. In the next chapter, we show the soundness of our construction by proving a non-interference theorem.

6 Soundness of the dependency analysis

Contents

This chapter proves the soundness of the dependency analysis presented in previous chapter. Most of this section is dedicated to the proof of the following theorem:

Theorem 6.1 (Correctness of non-interference hints)**.** *Consider a method m, and let $\Theta(m) = (V, E)$ be the AMG of a method m. Let $\overline{a}$ be a concrete object and $\overline{b}$ a primitive input value in the concrete execution of m, and a and b their abstractions in the abstract model. If ni$_{\Theta(m)}(a, b)$,* i.e.*, $\overline{a}$ and $\overline{b}$ do not interfer, then changing the input value $\overline{b}$ does not affect the output value of $\overline{a}$,* i.e.*, the abstraction of $\overline{a}$ does not change.*

The text in the theorem is not very formal. In particular, we do not say what we mean by *concrete execution*, *abstraction relation*, *input value*, *output value* etc. We clarify all these notions along the proof. Moreover, we define additional models (*e.g.*, a concrete execution model, an intermediate model) and we prove additional results (*e.g.*, memory abstraction, state variation) needed to prove the non-interference theorem.

6.1 Proof outline

In order to study the correctness of the analysis, we needed to define a concrete execution model for the JVM and to prove that if a non-interference property in the abstract model holds, then the non-interference property in the concrete model also holds. The definition of the concrete model requires to define the memory heap, concrete objects and concrete semantics for the analyzed language. While the abstract semantics has an intuitive, abstract view about the execution of a program, the concrete semantics has a precise, operational view. The concrete semantics precisely defines the execution of a program, while the abstract semantics gives an approximation which holds for every possible execution of the program.

In order to take in consideration method calls, most of the previous approaches[Mye99a] rely on the call graph. This requires the entire class set to be known from the begining, which is not possible in an open environment. As we target open devices, we define a modular approach for the inter-procedural analysis. Proving the soundness of the modular approach is a complex task and it represents an important and sensitive part of the proof: we must show that graph composition safely approximates concrete method invocation.

We split the prove in two parts: *(i)* we first prove the correctness of the intra-procedural analysis by considering programs without method invocation; *(ii)* in the second part, we complete the proof by showing the soundness of the inter-procedural analysis.

(i) In order to prove the correctness of our construction during the intra-procedural analysis, we need to investigate the relation between the AMG resulted from our analysis and the concrete state resulted from the concrete semantics. Due to the fact that we consider primitive values in our graph, we cannot create a relation between them in a single step. Again, we split the proof in two parts:

1. First, we prove the correctness of our AMG restricted to references; the restraint graph (points-to graph) is related directly to the concrete model through an abstraction relation.

2. Next, we concentrate on primitive values and we show that our construction is correct w.r.t. to non-interference.

(ii) The proof for inter-procedural analysis must show the correctness of the abstract execution of the `invoke` instruction (which combines the AMGs of the caller and of the callee). Unfortunately, due to the significant difference between the abstract and the concrete semantics of the `invoke` instruction, we cannot investigate the proof and the relation between the two semantics in a single step. Hence, we introduce an *intermediate layer* between concrete semantics and dependency analysis. The intermediate semantics is similar to the abstract semantics, except for the invocation model: the intermediate semantics steps into the callee and it corresponds to unfolding method calls, just as the concrete semantics does. Based on this intermediate layer, we split the proof of inter-procedural analysis in two steps:

1. First, we proof the correctness of the intermediate layer.

2. Then, we show that the AMG of a method resulted from the abstract semantics contains at least the same edges as the AMG resulted from the intermediate layer.

6.2 Concrete model and concrete semantics

To simplify the proofs, we restrict ourselves to a subset of the instruction set considered for the abstract model. The initial instruction set is depicted in Figure 3.1 page 25, while the restricted instruction set (which we will consider for the concrete model) is depicted in Figure 6.1. The subset keeps the main features of JVM, such as pointer manipulation, method call, jumps, etc. The subset is just a simplification of the initial intruction set (*e.g.*, it does not contain bytecodes dealing with static variables and arrays as they are a particular case of `getfield` and `putfield` and their proofs are similar).

The concrete model precisely defines the execution of JVM programs. The set of JVM values is defined as $Jv = Val \cup Obj$ where Val is the set of primitive values (restricted to int), and Obj the set of objects, including the special value `null`. We extend the function $Type$ to the concrete model: $Type : Obj \rightarrow Class$ returns the type of an object.

`prim` *op*	primitive operation taking two operands, pushing the result on the stack
`pop`	pop the top of the stack
`bipush` n	push the primitive value n on the stack
`aconst_null`	push null on the stack
`new` C	creates new object of type C in the memory
`goto` a	jump to address a
`ifeq` a	jump to address a if the top of the stack is equal to zero
α`load` x	push the content of the local variable x on the stack
α`store` x	pop the top of the stack and store it into the local variable x
`getfield` $f_{C'}$	load the field $f_{C'}$ of the top of the stack on the stack
`putfield` $f_{C'}$	store the top of the stack in the field $f_{C'}$ of an object on the stack
`invoke` $m_{C'}$	virtual invocation of method $m_{C'}$
α`return`	return an object and exit the method

α denotes the type: `i` for primitive types, `a` for references

Figure 6.1: Subset of the instruction set in the concrete model

6.2.1 Memory model

We define the memory heap as a directed graph[1]. The memory is represented by a *memory graph* G which is a triple $G = (V, E, \varsigma)$ defined in the following. A node of V designates a location: V is partitioned into $\mathcal{O}(V) \subseteq Obj$ for locations containing object references and $\mathcal{V}(V)$ for locations containing values of Val. As for the AMGs, for a memory graph $G = (V, E, \varsigma)$ we use $\mathcal{O}(G)$ and $\mathcal{V}(G)$ instead of $\mathcal{O}(V)$ and $\mathcal{V}(V)$ respectively. As values contained in locations may change during the execution, we use an injective function $\varsigma : \mathcal{V}(V) \longrightarrow Val$ that labels each node from $\mathcal{V}(V)$ with the value stored in this location; edges represent field references and are labeled with field names (from $Field$). The unique node `null` and nodes of $\mathcal{V}(V)$ (primitive values) are leaves. For each primitive field of an object, the memory has a location (node) containing (labeled with) the value of the field, and this vertex is the target of a unique edge.

We consider an allocator function $G' = new(o, C)$ which creates a new graph structure for an object named o of type C; o is the root of the graph. If the object is allocated at instruction i, then o has the form C_i^k, where k is a counter such that C_i^k is uniquely used in G. The graph G' contains the initial values of the new created object (vertices containing 0 for fields of primitive type and edges to `null` for fields of type object).

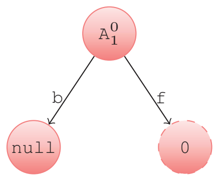
For a memory graph $G = (V, E, \varsigma)$, $G[u \mapsto l]$ designates the graph $(V, E, \varsigma[u \mapsto l])$. We extend naturally the definition of union, inclusion, $G\lfloor u \rfloor$, and $G[(u, f) \mapsto v]$ to memory graphs. Moreover, we denote $G_1 \equiv G_2$ if the memory graphs G_1 and G_2 are isomorphic (considering both edge and vertex labelling).

Example 6.1 (Memory graph example). Figure 6.2a shows the result of the allocator function for $\mathtt{A}_1^0$, thus after executing instruction `1`. Figure 6.2b shows the memory graph after executing the entire code from Figure 6.2. Affecting the object $\mathtt{B}_1^2$ to the field b of $\mathtt{A}_1^0$ (bytecodes `6-8`) consists of creating a new edge labelled with `b` from $\mathtt{A}_1^0$ to $\mathtt{B}_1^2$, while the assignment of the value 3 to the field `f` of $\mathtt{A}_1^0$ (bytecodes `12-14`) changes the label of the node reached from $\mathtt{A}_1^0$ with an `f`-edge.

```
class A {
  int f;
  B b;
}

class B {
  int g;
}
```

```
0 : new A            10: aload  1
1 : astore 0         11: putfield b_A
2 : new B            12: aload  0
3 : astore 1         13: bipush 3
4 : new A            14: putfield f_A
5 : astore 2         15: aload  2
6 : aload  0         16: aload  0
7 : aload  1         17: getfield f_A
8 : putfield b_A     18: putfield f_A
9 : aload  2
```

(a) After 0



(b) After 18

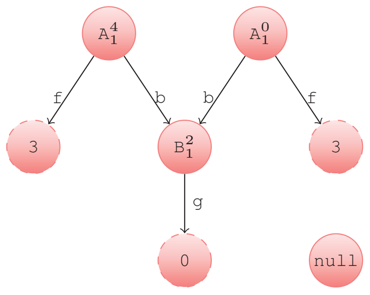
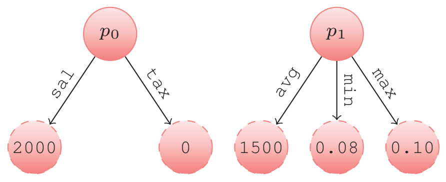


Figure 6.2: Memory Example

(a) After 0



(b) After 18

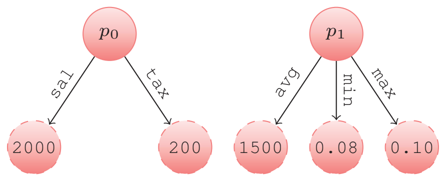


Figure 6.3: Memory Graph for `tax` method

6.2.2 Operational semantics

A (concrete) state of our machine is a pair $Q = (fr, G)$, where fr represents a stack of frames corresponding to a dynamic chain of method calls (with the current frame on the top), and G is the memory graph. A frame has the form (pc, ρ, s), where $pc \in P_m$ is the program counter, $\rho : \chi_m \to Jv$ represents a value assignment of local variables, and s is a stack with elements in Jv. The operational semantics of Java bytecode, denoted by $\overline{instr_b}(Q)$, is presented in Figure 6.4.

A state $Q = ((i, \rho, s) :: fr, G)$ is a *good initial* state for a block B, if $B[i]$ is defined and executing the block B starting from the state Q terminates in a state denoted by $\overline{instr_B}(Q)$.

For example, the `putfield` bytecode has two rules depending on the type of the manipulated field: changing the value of a primitive field means changing the label of a vertex, whereas changing an object field consists of changing the edge to the vertex containing the new pointed object. Figure 6.3 shows memory graphs of the `tax` method (the source code was described in Example 5.1, page 115 and depicted in Figure 5.8, page 116). Figure 6.3a shows the memory graph at the beginning of the method, while Figure 6.3b shows the memory graph after executing the entire method. Note that the `putfield` bytecode at instruction `18`, which assigns a value to the field `tax`, actually modifies the label of the memory location corresponding to the `tax` field.

[1]The advantage of this representation is the independence between objects and actual locations where these objects are allocated. However, the graph representing the memory is isomorphic to any address assignment in an execution. This independence property is crucial when comparing memories for two executions of the same method for different input values.

$$\frac{P_m[i] = \texttt{prim}\ op \qquad n = op(n_1, n_2)}{((i, \rho, n_1 :: n_2 :: s) :: fr, G) \longrightarrow ((i+1, \rho, n :: s) :: fr, G)} \qquad \frac{P_m[i] = \texttt{goto}\ \ a}{F \longrightarrow ((a, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{aconst\_null}}{F \longrightarrow ((i+1, \rho, \texttt{null} :: s) :: fr, G)} \qquad \frac{P_m[i] = \texttt{pop}}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{bipush}\ \ n}{F \longrightarrow ((i+1, \rho, n :: s) :: fr, G)} \qquad \frac{P_m[i] = \texttt{new}\ \ C \qquad o = C_i^{\mathsf{fresh}(C_i, G)} \qquad G' = new(o, C)}{F \longrightarrow ((i+1, \rho, o :: s) :: fr, G \cup G')}$$

$$\frac{P_m[i] = \texttt{ifeq}\ \ a \qquad n! = 0}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G)} \qquad \frac{P_m[i] = \texttt{ifeq}\ \ a \qquad n == 0}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((a, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \alpha\texttt{load}\ \ x}{F \longrightarrow ((i+1, \rho, \rho(x) :: s) :: fr, G)} \qquad \frac{P_m[i] = \alpha\texttt{store}\ \ x}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho[x \mapsto n], s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{getfield}\ \ f_{C'} \qquad n \neq \texttt{null} \qquad n' \in adj_G(n, f_{C'}) \qquad n' \in Obj}{((i, \rho, n :: s) :: fr, G) \longrightarrow ((i+1, \rho, n' :: s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{getfield}\ \ f_{C'} \qquad n \neq \texttt{null} \qquad n' \in adj_G(n, f_{C'}) \qquad n' \notin Obj}{((i, \rho, n :: s) :: fr, (V, E, \varsigma)) \longrightarrow ((i+1, \rho, \varsigma(n') :: s) :: fr, (V, E, \varsigma))}$$

$$\frac{P_m[i] = \texttt{putfield}\ \ f_{C'} \qquad n \neq \texttt{null} \qquad v \in Val \qquad n' \in adj_G(n, f_{C'})}{((i, \rho, v :: n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G[n' \mapsto v])}$$

$$\frac{P_m[i] = \texttt{putfield}\ \ f_{C'} \qquad n \neq \texttt{null} \qquad v \notin Val}{((i, \rho, v :: n :: s) :: fr, G) \longrightarrow ((i+1, \rho, s) :: fr, G[(n, f_{C'}) \mapsto v])}$$

$$\frac{P_m[i] = \texttt{invoke}\ \ m_{C'} \qquad o \neq \texttt{null}}{((i, \rho, p_{n_m} :: \cdots :: p_1 :: o :: s) :: fr, G) \longrightarrow ((0, \{0 \mapsto o, 1 \mapsto p_1 \ldots n_m \mapsto p_{n_m}\}, \epsilon) :: (i, \rho, s) :: fr, G)}$$

$$\frac{P_m[i] = \texttt{areturn}}{((i, \rho, v :: s) :: (i', \rho', s') :: fr, G) \longrightarrow ((i'+1, \rho', v :: s') :: fr, G)}$$

where $F = ((i, \rho, s) :: fr, G)$. $\texttt{prim}\ op$ stands for primitive operations with two parameters. The function $\mathsf{fresh}(c, G)$ returns a natural k such that c^k is not used as a vertex label in G.

Figure 6.4: A subset of operational semantics rules

6.3 Non-interference for concrete model

Our analysis computes an abstraction on how the value of some objects may depend or not on the value of others. We formalize this notion of dependency through non-interference: roughly, at the method level, the non-interference between an object o and an input value ι (of primitive type) can be stated as "changing the value of ι does not affect the value of o". The "value" of an object

```
void m (A p1, A p2) {
  int i=1;
  int j=0;
  int k=2;
  p1.f = i;
  p2.f = k-p1.f-j;
}
```

```
0:   bipush 1          13:  aload 2
1:   istore 3          14:  iload 5
2:   bipush 0          16:  aload 1
3:   istore 4          17:  getfield f_A
5:   bipush 2          20:  isub
6:   istore 5          21:  iload 4
8:   aload 1           23:  isub
9:   iload 3           24:  putfield f_A
10:  putfield f_A
```

(a) memory graph **(b)** AMG

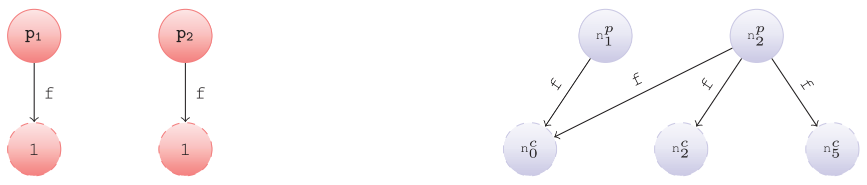


Figure 6.5: Relations Example

is defined by the values of its fields of primitive type and recursively, by the "values" of its object fields. An input value of a block is a value of primitive type chosen in the concrete state before the execution of the block; formally:

Definition 6.2 (Set of input values)**.** Let $Q = ((i, \rho, s) :: fr, G)$ be a state. Then, the set of *input values* of Q is $\mathcal{I}(Q) = \mathcal{V}(G) \cup \{x \in \chi_m \mid \rho(x) \in Val\}$.

Definition 6.3 (Value-change)**.** The value-change of a state $Q = ((i, \rho, s) :: fr, G)$ and an input value ι from $\mathcal{I}(Q)$ is a state $Q' = ((i, \rho', s) :: fr, G')$ such that for some value a of primitive type,

- either $\iota \in \mathcal{V}(G)$, $\rho = \rho'$ and $G' = G[\iota \mapsto a]$,
- or $\iota \in \{x \in \chi_m \mid \rho(x) \in Val\}$, $\rho' = \rho[\iota \mapsto a]$ and $G' = G$.

Definition 6.4 (Non-interference)**.** For blocks B and each state Q_1, let G_1, G_2 be the memory graphs of Q_1 and $\overline{instr}_B(Q_1)$ respectively. For any input value ι from $\mathcal{I}(Q_1)$ and for any object node o in G_1, *ι does not interfere in B with o* if for any value-change Q_1' of Q_1 and ι, one has $G_2\lfloor o \rfloor \equiv G_2'\lfloor o \rfloor$, G_2' being the memory graph of $\overline{instr}_B(Q_1')$.

6.4 Relations between abstract and concrete models

Notation. As, in the next sections, we compare concrete and abstract worlds, throughout the remainder of this thesis, by $\overline{e}$ we denote a concrete element and by e an abstract element. Note that e is just an abstract element, and not necessarily the abstraction of $\overline{e}$.

We first relate the abstract and the concrete semantics formally: we consider an abstraction function to relate memory graphs restricted to objects nodes and AMGs. Because of the allocation site model, it is possible to relate concrete nodes to abstract nodes in a non ambiguous manner. However, for complete graphs (including primitive nodes), we cannot define an abstract relation as it is not possible to associate uniquely a concrete primitive node with an abstract one.

An example is described in Figure 6.5. The execution of the method `m` leads to the memory graph in Figure 6.5a, while our analysis computes the AMG in Figure 6.5b for `m`. For example, it is not possible to decide which node to associate with the concrete dashed node: ${n_0}^c$, ${n_2}^c$, or ${n_5}^c$ since the value `1` is the result of a calculus that implies several values. This is the reason why the abstraction relation is only defined for the restriction of graphs to their object part.

Definition 6.5 (Abstraction function)**.** Let $\overline{W}$ and W be two sets of nodes. A relation α from $\overline{W}$ to W is an *abstraction function* if it is a mapping that respects the allocation site model: for all k, all the C_i^k are mapped to n_i^n, and $\alpha(\texttt{null}) = n_\perp^{\texttt{null}}$.

Definition 6.6 (α-abstraction)**.** Let $\overline{G} = (\overline{V}, \overline{E}, \varsigma)$ be a memory graph and G be an abstract graph. Let α be an abstraction function between the sets of nodes $\mathcal{O}(\overline{G})$ and $\mathcal{O}(G)$. Then, G is an α-abstraction of $\overline{G}$, denoted by $\overline{G} \lhd^\alpha G$ if $\alpha(\overline{G}) \subseteq G$, where $\alpha(\overline{G})$ is the graph $(\{\alpha(\overline{u}) \mid \overline{u} \in \mathcal{O}(\overline{G})\}, \{(\alpha(\overline{u}), \alpha(\overline{v}), \langle f, \mathbf{d}\rangle) \mid \overline{u}, \overline{v} \in \mathcal{O}(\overline{G}) \wedge (\overline{u}, \overline{v}, f) \in \overline{E}\})$. [2]

The abstraction function is carried over concrete and abstract states.

Definition 6.7 (State abstraction)**.** Let α be an abstraction function. Given a concrete state $\overline{Q} = ((pc, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ and an abstract state $Q = (\rho, s, G)$, Q is an α-abstraction of $\overline{Q}$ (denoted by $\overline{Q} \lhd^\alpha Q$) if $\overline{G} \lhd^\alpha G$, $\overline{s} \lhd^\alpha s$, and $\overline{\rho} \lhd^\alpha \rho$, with $\overline{s} \lhd^\alpha s$ if s,$\overline{s}$ are both empty or if $\overline{s} = \overline{v} :: \overline{s_1}$, $s = v :: s_1$, $\overline{s_1} \lhd^\alpha s_1$, and $(\alpha(\overline{v}), \mathbf{d}) \in v$ if $\overline{v} \in \mathcal{O}(\overline{G})$.

The local variables abstraction, denoted by $\overline{\rho} \lhd^\alpha \rho$, is defined similarly to stack abstraction. We now extend the abstraction relation to take into account primitive nodes which correspond to input values.

Definition 6.8 (α-abstraction extension)**.** Let $\overline{Q} = ((i, \overline{\rho}, \overline{s}) :: fr, (\overline{V}, \overline{E}, \varsigma))$ be a concrete state, and $Q = (\rho, s, (V, E))$ such that $\overline{Q} \lhd^\alpha Q$. Then, $\alpha_Q : \mathcal{I}(\overline{Q}) \longrightarrow \wp(\mathcal{V}(V))$ is the unique extension of α in $\overline{Q}$ if

$$\begin{array}{ll} \forall \overline{\iota} \in \mathcal{I}(\overline{Q}) \cap \mathcal{V}(\overline{V}), & \alpha_Q(\overline{\iota}) = \{\iota \mid \exists (\alpha(\overline{u}), \iota, \langle f, \mathbf{d}\rangle) \in E, \text{with} (\overline{u}, \overline{\iota}, f) \in \overline{E}\} \\ \forall \overline{\iota} \in \mathcal{I}(\overline{Q}) \cap \chi_m, & \alpha_Q(\overline{\iota}) = \{e \mid \langle e, \mathbf{d}\rangle \in \rho(\overline{\iota}))\} \end{array}$$

Definition 6.9. Let $Q = (\rho, s, (V, E))$ be an abstract state. Let $\beta \subseteq \mathcal{V}(V)$. For any $o \in \mathcal{O}(V)$, we have *free*(o, β, Q) if one of the following conditions holds:

- $\exists\ (u, o, \langle f, \mathbf{d}\rangle) \in E$ and $\beta \cap adj_{(V,E)}(u, \langle f, \mathbf{i}\rangle) = \emptyset$,
- $\exists x$ such that $(o, \mathbf{d}) \in \rho(x)$ and $\nexists (\iota, \mathbf{i}) \in \rho(x)$ with $\iota \in \beta$,
- $\exists i$ such that $(o, \mathbf{d}) \in s[i]$ and $\nexists (\iota, \mathbf{i}) \in s[i]$ with $\iota \in \beta$.

The function *free*(o, β, Q) expresses that o is not bound to any ι of β in Q, meaning that at least one presence of o on stack, local variables or in AMG does not implicitly depend on β. In the non-interference theorem, this condition ensures the existence of o (with $\alpha(\overline{o}) = o$) after a value-change, as varying ι may influence the execution path, and thus the objects being created.

Theorem 6.10 (Non-interference)**.** *Let B be an instruction block, $\overline{Q}$ be a concrete state, and Q be an abstract state such that $\overline{Q} \lhd^\alpha Q$. Let $\overline{G}$ be the memory graph of $\overline{Q}$ and G' be the abstract graph of $instr_B(Q, \Gamma)$. For a reference node $\overline{o}$ in $\mathcal{O}(\overline{G})$ and an input value $\overline{\iota}$ from $\mathcal{I}(\overline{Q})$ such that free$(\alpha(\overline{o}), \alpha_Q(\overline{\iota}), Q)$, if Reach$_{G'}(\alpha(\overline{o}) \cap \alpha_Q(\overline{\iota})) = \emptyset$ then $\overline{\iota}$ does not interfere with $\overline{o}$ in B.*

This is the main theorem of the correctness proof. Next sections are devoted to additional definitions and lemma needed by the proof, and finally we conclude with the proof of Theorem6.10.

6.5 Soundness of intra-procedural analysis

We now prove the correctness of the AMG construction, as presented in Section 5.3, with respect to non-interference: if there is no dependency (path in the abstract graph) between an object and an input value, then changing the input value in the concrete graph will not affect the concrete graph of the object. Thus, this section is devoted to the proof of the non-interference theorem in a program without method calls.

[2]Recall that the AMG has the property that there are only direct flows (edged labeled $\langle f, \mathbf{d}\rangle$) between references nodes.

6.5.1 Points-to correctness

We first show that when the AMGs are restricted to references, our analysis is a sound points-to analysis. We rely on the dataflow framework which is slightly adapted to take into account the flow information of the Γ's.

Proposition 6.11 (Points-to correctness)**.** *Let B be a block, $\overline{Q}$ be a good initial (concrete) state for B, α be an abstraction function, and Q be an abstract state. Then, for any set of abstract values Γ, $\overline{Q} \lhd^\alpha Q$ implies $\overline{instr}_B(\overline{Q}) \lhd^\alpha instr_B(Q, \Gamma)$.*

Proof. We split the proof of Proposition 6.11 in two parts: first we prove the local soundness (the theorem holds for blocks containing a single instruction). Later we address structured blocks containing many instructions. □

Single instruction block (local soundness)

We prove the points-to correctness for single instruction blocks by case analysis on each instruction. The complete proof is presented in Appendix A.3.1, page 166.

Points-to correctness for blocks

The proof relies on the general theorem stating the correctness of the global soundness provided that the local correctness holds. However, our system of equations is defined for a pair (Q, Γ) and the computation of the values of Γ's is not performed structurally on the control flow graph but based on regions instead.

Our proof will simply show that the restriction to objects of AMGs from the computed abstract states Q are independent from the Γ's. Therefore, we will be able to define a new system of equation $\mathcal{E}'_B$ for a block B that differ from $\mathcal{E}_B$ only on the interpretation of the functionals appearing in the system, that becomes independent of Γ. But it turns out that the points-to graphs in the states computed for $\mathcal{E}_B$ and $\mathcal{E}'_B$ are identical.

Given a memory graph $(\overline{V}, \overline{E}, \varsigma)$, we define by $\Pi_{Obj}((\overline{V}, \overline{E}, \varsigma))$ the subgraph of $(\overline{V}, \overline{E})$ restricted to objects as $(\mathcal{O}(\overline{V}), \{(\overline{u}, \overline{v}, f) \mid \overline{u}, \overline{v} \in \mathcal{O}(\overline{V})$ and $(\overline{u}, \overline{v}, f) \in \overline{E}\})$. We extend this definition to AMGs: $\Pi_{Obj}((V, E)) = (\mathcal{O}(V), \{(u, v, \langle f, \mathbf{d}\rangle) \mid u, v \in \mathcal{O}(V)$ and $(u, v, \langle f, \mathbf{d}\rangle) \in E\})$.

For two AMGs, G and G', we write $G \doteq G'$ if and only if G and G' are equal when restricted to their object part, *i.e.*, $\Pi_{Obj}(G) = \Pi_{Obj}(G')$. This relation $\doteq$ is also defined on sets of typed-flow vertices as : $S_1 \doteq S_2$ if $\{e \mid \langle e, \mathbf{d}\rangle \in S_1 \wedge e \in \mathcal{V}^{\mathcal{O}}\} = \{e \mid \langle e, \mathbf{d}\rangle \in S_2 \wedge e \in \mathcal{V}^{\mathcal{O}}\}$, where $\mathcal{V}^{\mathcal{O}}$ is a set of vertices. This relation is extended to abstract states as follows: $Q = (\rho, s, G) \doteq (\rho', s', G') = Q'$ if $G \doteq G'$ and

- for all x in χ_m, $\rho(x) \doteq \rho'(x)$
- either $s = s' = \varepsilon$ or $s = u :: s_t$, $s' = u' :: s'_t$, $u \doteq u'$ and $s_t \doteq s'_t$.

Proposition 6.12. *For any concrete state $\overline{Q}$, for any abstract states, Q and Q', if $\overline{Q} \lhd^\alpha Q$ and $Q \doteq Q'$ then $\overline{Q} \lhd^\alpha Q'$.*

Proof. Let $\overline{Q} = ((i, \overline{\rho}, \overline{s}) :: fr, \overline{G})$, $Q = (\rho, s, G)$ and $Q' = (\rho', s', G')$. From the definition 6.7 of state abstraction, $\overline{Q} \lhd^\alpha Q'$, if $\overline{s} \lhd^\alpha s', \overline{\rho} \lhd^\alpha \rho'$ and $\overline{G} \lhd^\alpha G'$. By hypothesis, $\overline{Q} \lhd^\alpha Q$ thus $\overline{s} \lhd^\alpha s, \overline{\rho} \lhd^\alpha \rho$ and $\overline{G} \lhd^\alpha G$. Moreover $Q \doteq Q'$ thus $s \doteq s', \rho \doteq \rho'$ and $G \doteq G'$.

Let us first prove that $\overline{G} \lhd^\alpha G'$. From $G \doteq G'$ we have $\Pi_{Obj}(G) = \Pi_{Obj}(G')$. Since $\overline{G} \lhd^\alpha G$, $\alpha(\overline{G}) \subseteq G$. From the Definition 6.5 of abstraction of a memory graph and the definition of $\Pi_{Obj}(G)$, we deduce the following property $\alpha(\overline{G}) \subseteq \Pi_{Obj}(G)$. As $\Pi_{Obj}(G) = \Pi_{Obj}(G')$, then $\alpha(\overline{G}) \subseteq \Pi_{Obj}(G') \subseteq G'$. Thus $\overline{G} \lhd^\alpha G'$.

Let us now prove that $\overline{s} \lhd^{\alpha} s'$. Let $\overline{s} = \overline{u} :: \overline{s}_t$, $s = u :: s_t$ and $s' = u' :: s'_t$. As $\overline{s} \lhd^{\alpha} s$, thus $\overline{s}_t \lhd^{\alpha} s_t$ and $(\alpha(\overline{u}), \mathbf{d}) \in u$ if $\overline{u} \in \mathcal{O}(\overline{G})$. If $\overline{u} \in \mathcal{O}(\overline{G})$, then $\alpha(\overline{u}) \in \mathcal{O}(G)$.

By hpothesis, $s \doteq s'$. From the definition of $s \doteq s'$ if $s = s' = \epsilon$ or, $u \doteq u'$ and $s_t \doteq s'_t$. Moreover, the two sets of typed-flow nodes, u and u', are equal accoding to $\doteq$ relation if $\{e \mid \langle e, \mathbf{d}\rangle \in u \wedge e \in \mathcal{O}(G)\} = \{e \mid \langle e, \mathbf{d}\rangle \in u' \wedge e \in \mathcal{O}(G')\}$. Thus, if $\alpha(\overline{u}) \in \mathcal{O}(G)$ and $(\alpha(\overline{u}), \mathbf{d}) \in u$ then $(\alpha(\overline{u}), \mathbf{d}) \in u'$. Hence $\overline{s} \lhd^{\alpha} s'$.

The proof for local variables array is similar. □

Proposition 6.13. *For all abstract states Q_1, Q'_1, Q_2, Q'_2,*

- *if $Q_1 \doteq Q_2$ and $Q'_1 \doteq Q'_2$ then $Q_1 \sqcup Q'_1 \doteq Q_2 \sqcup Q'_2$,*
- *for any bytecode b, for any Γ_1, Γ_2, if $Q_1 \doteq Q_2$ then*

$$instr_b(Q_1, \Gamma_1) \doteq instr_b(Q_2, \Gamma_2)$$

Proof. For the first point, as $\sqcup$ is defined component-wise for stacks and mappings ρ, the only part to be proved is for graphs. Let $Q_1 = (\rho_1, s_1, G_1)$, $Q_2 = (\rho_2, s_2, G_2)$, $Q'_1 = (\rho'_1, s'_1, G'_1)$ and $Q'_2 = (\rho'_2, s'_2, G'_2)$.

If $G_1 \doteq G_2$ then the two graphs are equal when restricted to their object part, thus $\Pi_{Obj}(G_1) = \Pi_{Obj}(G_2)$. Moreover, $\Pi_{Obj}(G'_1) = \Pi_{Obj}(G'_2)$, hence $\Pi_{Obj}(G_1) \cup \Pi_{Obj}(G'_1) = \Pi_{Obj}(G_2) \cup \Pi_{Obj}(G'_2)$.

We must prove that $\Pi_{Obj}(G_1 \cup G'_1) = \Pi_{Obj}(G_2 \cup G'_2)$. It is enough to prove that, for two AMGs G and G', $\Pi_{Obj}(G \cup G') = \Pi_{Obj}(G) \cup \Pi_{Obj}(G')$.

The proof lies on the property of AMGs stating that any node $u \in \mathcal{V}(G)$ is a leaf. Thus, an AMG contains only edges between two reference nodes or edges from a reference node to a primitive node: there is no edge from a primitive node to a reference node. When making the union of two AMGs, the only edges between objects can come from the two AMGs. Thus, the restriction to objects part of the union of two graphs is equal to the union of the restrictions of the two graphs to objects: $\Pi_{Obj}(G \cup G') = \Pi_{Obj}(G) \cup \Pi_{Obj}(G')$.

For the second point, we make a case analysis on each bytecode instruction in Appendix A.3.2, page 168. □

Let us consider the system of equations $\mathcal{E}_B$. Each right-hand side can be seen as a functional over (Q_i, Γ_i), the unknowns. Let us define the system $\mathcal{E}'_B$ with $(Q_0, \varnothing)$ for initial state:

$$(Q_i, \Gamma_i) = \begin{cases} (Q_0 \sqcup_{j \in pred(i)} instr_{B[j]}(Q_j, \Gamma_j), & \text{if } i = \mathsf{Entry}(B) \\ \quad \{u \mid \langle u, t\rangle \in v \text{ with } Q_k = (\rho, v :: s, G), k \in cxt(i)\}) & \\ (\sqcup_{j \in pred(i)} instr_{B[j]}(Q_j, \Gamma_j), & \text{otherwise} \\ \quad \{u \mid \langle u, t\rangle \in v \text{ with } Q_k = (\rho, v :: s, G), k \in cxt(i)\}) & \end{cases}$$

We define for a block B and the system $\mathcal{E}'_B$, the mapping $instr'_B(Q, \Gamma)$ as we did for $instr_B(Q, \Gamma)$ and $\mathcal{E}_B$.

Obviously, for the least solution of $\mathcal{E}'_B$, we have that $\Gamma_i = \varnothing$ for all i. Therefore, the functionals in the right-hand side and thus, the solution of the system are independent from the Γ'. As Proposition 6.11 establishes the local soundness for all Γ's, in particular for $\varnothing$, we have that the global soundness holds for the system $\mathcal{E}'_B$, that is

$$\overline{Q} \lhd^{\alpha} Q \text{ implies } \overline{instr}_B(\overline{Q}) \lhd^{\alpha} instr'_B(Q, \varnothing) \tag{6.5.1}$$

However, the system $\mathcal{E}_B$ involves the Γ's which influence on the values of the Q's but not on their object part as we show now.

Proposition 6.14. *Provided that their respective initial states satisfies $Q \doteq Q'$, then for the respective least solution of $\mathcal{E}_B$ and $\mathcal{E}'_B$, we have that $Q_i \doteq Q'_i$ for all i.*

Proof. We rely here on the iterative construction of the least solution of equation systems. Starting from the valuation assigning the least value of the lattice to all the unknowns, that is the (Q_i, Γ_i)'s and the (Q'_i, Γ'_i)'s respectively, we compute a new value for these valuations (at the left-hand side of equalities) by using the old values in the right-hand side. Eventually, a fixed point is reached during that computation.

Then, the proof goes by induction on n, the minimal number of iterations required to reach a fixed point in both system using Proposition 6.13 for the induction step. □

Proposition 6.15. *For any block B, for any Γ_1, Γ_2, if $Q_1 \doteq Q_2$ then $instr_B(Q_1, \Gamma_1) \doteq instr'_B(Q_2, \Gamma_2)$*

Proof. Straightforward from Propositions 6.14 and 6.13. □

By combining the global soundness (6.5.1), Propositions 6.15 and 6.12, we get Proposition 6.11 as result. This ensures correctness for AMG restricted to references.

We now prove the correctness of the primitive edges (implicit and direct flow) as a non-interference theorem relying on the correctness for the points-to analysis.

6.5.2 Correctness for primitive values

To prove the non-interference theorem, according to Definition 6.4, we need to make values vary at some program point according to Definition 6.3 and to check the impact of this variation on objects at another program point. Thus, we first define the notion of *state variation* that captures how a concrete execution state, corresponding to a program point i, might change when an input value ι has changed in the past of this execution. Definition 6.16 contains an over estimation of the set of states that the JVM can reach after this change: the main impact is on the memory graph, but the local variables and stack can also be affected. The correctness of the definition is proved later in Proposition 6.18.

Definition 6.16 (State variation)**.** Let $\overline{Q} = ((i, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ and $\overline{Q'} = ((i, \overline{\rho}', \overline{s}') :: fr, \overline{G}')$ be two concrete states. Let $Q = (\rho, s, G)$ be an abstract state. Let α be an abstraction function. Let $\beta \subseteq \mathcal{V}(G)$ be a set of primitive nodes.

Then $\overline{Q}$ and $\overline{Q'}$ are **state variation** from each other with respect to Q and β, denoted by $\overline{Q} \overset{Q,\beta}{\longleftrightarrow} \overline{Q'}$, if $\overline{Q} \lhd^\alpha Q$, $\overline{Q'} \lhd^\alpha Q$ and

- $\forall x, \overline{\rho}(x) = \overline{\rho}'(x) \vee \exists \iota \in \beta$ such that $(\iota, t) \in \rho(x)$,
- $\forall i, \overline{s}[i] = \overline{s}'[i] \vee \exists \iota \in \beta$ such that $(\iota, t) \in s[i]$,
- for all $\overline{v} \in \mathcal{O}(\overline{G}) \cup \mathcal{O}(\overline{G}')$, either $\overline{v} \in \mathcal{O}(\overline{G}) \cap \mathcal{O}(\overline{G}')$ or $\neg free(\alpha(\overline{v}), \beta, Q)$,
- for all $\overline{v}$ in $\mathcal{O}(\overline{G}) \cap \mathcal{O}(\overline{G}')$, for all fields f from $Type(\overline{v})$,
  - either there exists a unique node $\overline{w}$ such that $(\overline{v}, \overline{w}, f)$ is an edge from $\overline{G}$ and from $\overline{G}'$ and, moreover, if $\overline{w} \in \mathcal{V}(\overline{G}')$, then $\varsigma(\overline{w}) = \varsigma'(\overline{w})$,
  - or some edge $(\alpha(\overline{v}), \iota, \langle f, t \rangle)$ exists in G for some ι in β.

From state variation definition, we can state that the variation of a state regarding to a set β does not impact the objects $\overline{o}$ which have no dependence to any node of β.

Lemma 6.17. *Let α be an abstraction function, $\overline{Q}= ((i, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ a concrete state, and $Q=(\rho, s, G)$ an abstract state with $\overline{Q} \lhd^{\alpha} Q$. Let $\beta \subseteq \mathcal{V}(G)$, and $\overline{o} \in \mathcal{O}(\overline{G})$ such that there exists no path from $\alpha(\overline{o})$ to any node of β in G. Then, in any state variation $\overline{Q}' \overset{Q,\beta}{\longleftrightarrow} \overline{Q}$ with $\overline{Q}' = ((i, \overline{\rho}', \overline{s}') :: fr, \overline{G}')$, if $\overline{o} \in \mathcal{O}(\overline{G}')$, we have $\overline{G}\lfloor \overline{o} \rfloor \equiv \overline{G}'\lfloor \overline{o} \rfloor$.*

Proof. The differences between $\overline{G} = (\overline{E}, \overline{V}, \varsigma)$ and $\overline{G}' = (\overline{E}', \overline{V}', \varsigma')$ are given by rules of Definition 6.16. If $\overline{o} \in \mathcal{O}(\overline{G}')$, then, for any reference node $\overline{v}$ of the subgraph rooted by $\overline{o}$, there exists an edge $(\overline{u}, \overline{v}, f) \in \overline{E}$ such that $\overline{u}$ belongs to the subgraph rooted by $\overline{o}$ with:

- $\beta \cap adj_G(\alpha(\overline{u}), \langle f, t \rangle) = \emptyset$ because $Reach_G(\alpha(\overline{o})) \cap \beta = \emptyset$,
- $\overline{u} \in \mathcal{O}(\overline{G}')$ because either $\overline{u} = \overline{o}$, or $\overline{u}$ belongs to $\mathcal{O}(\overline{G}')$ for the same reason.

Thus, $\overline{v} \in \mathcal{O}(\overline{G}')$ and $(\overline{u}, \overline{v}, f) \in \overline{E}'$.

For the leaves of the subgraph rooted by $\overline{o}$, the value of $\overline{v}$ is randomized if there is an edge $(\overline{u}, \overline{v}, f) \in \overline{E}$ and a flow from $\alpha(\overline{u})$ to an element of β labelled with f in G. As by hypothesis there is no path from $\alpha(\overline{o})$ to any element of β in G, it is obvious that this modification does not affect the subgraph rooted by $\overline{o}$. □

Proposition 6.18 states the state variation correctness, by claiming that changing an input value and executing a block with a state variation leads us to a state variation.

Proposition 6.18 (State variation correctness)**.** *Let B be an instruction block. For all concrete states $\overline{Q_1}, \overline{Q_1'}$, for all abstract state Q_1, for all Γ_1,*

If $\overline{Q_1'} \overset{Q_1,\beta}{\longleftrightarrow} \overline{Q_1}$ then $\overline{instr_B}(\overline{Q_1'}) \overset{instr_B(Q_1,\Gamma_1),\beta}{\longleftrightarrow} \overline{instr_B}(\overline{Q_1})$.

We split the proof of Proposition 6.18 in two parts: first we prove the theorem holds for blocks containing a single instruction. Later we address blocks containing many instructions.

State variation correctness for single instruction block

We prove the state variation correctness for single instruction block by case analysis on each instruction, in Appendix A.3.3, page 170.

State variation correctness for blocks

Let us now prove the state variation correctness. The blocks we consider have a single entry point and a single exit point, thus, they are sets of instructions closed by region (this is the case of any method).

Remember the definition of a block B: it is a method m or a connected subgraph of the control flow graph CF_m with a single entry point, a single exit point such that each vertex of B is reachable from $Entry(B)$. Thus we can deduce the following property:

Property 6.19. *Let B be a block of a method m. Then, for each $i \in B$ either $reg(i) \cup \{IPD(i)\} \subseteq B$ or $reg(i) \cup \{IPD(i)\} \cap B = \emptyset$ or $i = Exit(B)$.*

We want to show the State variation correctness.

Proof. Let B be an instruction block, $\overline{Q}_k$ a concrete state, Q_k an abstract state with $\overline{Q}_k \lhd^{\alpha} Q_k$, $\overline{Q}_l = \overline{instr}_B(\overline{Q}_k)$, and $Q_l = instr_B(Q_k, \Gamma_k)$.

Let $\overline{Q}_k' \overset{Q_k,\beta}{\longleftrightarrow} \overline{Q}_k$ and $\overline{Q}_l' = \overline{instr}_B(\overline{Q}_k')$, we want to show that $\overline{Q}_l' \overset{Q_l,\beta}{\longleftrightarrow} \overline{Q}_l$.

Let us consider the sequences of concrete states corresponding to both executions of B: $T = \overline{Q}_{i_0}\overline{Q}_{i_1} \ldots \overline{Q}_{i_n}$ and $T' = \overline{Q}_{j_0}'\overline{Q}_{j_1}' \ldots \overline{Q}_{j_m}'$ with $i_0 = j_0 = k$ and $i_n = j_m = l$. Both traces may have different length and use different paths. We will show that traces end at the same instruction by states that are variations from each other by induction on the length of the longest trace.

Let us consider the pair of prefixes of T and T' such that P and P' are respectively the longuest prefix of T and T' such that:

- they respectively start with $\overline{Q}_{i_0}$ and $\overline{Q}'_{j_0}$,
- P ends with the first $\overline{Q}_{i_k}$ such that $i_k = ipd(i_0)$ and $i_{k+1} \notin reg(i_0)$,
- P' ends with the first $\overline{Q}'_{j_k}$ such that $j_k = ipd(i_0)$ and $j_{k+1} \notin reg(i_0)$.

Case 1. If length of P is 1 and length of P' is 1, then $i_1 = j_1$ and we have proved by case study on bytecodes that $\overline{Q}_{i_1} \overset{Q'_{j_1},\beta}{\longleftrightarrow} \overline{Q}'_{j_1}$. Since $\overline{Q}_{i_1} \dots \overline{Q}_{i_n}$ and $\overline{Q}'_{j_1} \dots \overline{Q}'_{j_m}$ are shortest than T and T' with $i_1 = j_j$ and $i_n = j_m = k$, we are done.

Case 2. If $i_1 \neq j_1$, then $P_m[i_0] = \mathtt{ifeq}$, and $\overline{Q}_{i_0} = ((i_0, \overline{\rho}_{i_0}, \overline{\iota} :: \overline{s}_{i_0}) :: fr, \overline{G}_{i_0})$ with $\alpha(\overline{\iota}) \in \beta$.

Thus, for each $l \in reg(i_0)$, $\alpha(\overline{\iota}) \in \Gamma_l$ and we can deduce that for each $\overline{Q_{i_l}} = ((i, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ that belongs to P, with $Q_{i_l} = (\rho, s, G)$, we have:

- $\forall x, \overline{\rho}(x) = \overline{\rho}_{i_0}(x) \vee (\iota, t) \in \rho(x)$,
- $\forall i, \overline{s}[i] = \overline{s}_{i_0}[i] \vee (\iota, t) \in s(x)$,
- for all $\overline{v} \in \mathcal{O}(\overline{G}) \setminus \mathcal{O}(\overline{G}_{i_0})$, $\neg free(\alpha(\overline{v}), \beta, \overline{Q}_{i_l})$,
- for all $\overline{v}$ in $\mathcal{O}(\overline{G}) \cap \mathcal{O}(\overline{G}_{i_0})$, for all fields f from $Type(\overline{v})$,
  - either there exists a unique node $\overline{w}$ such that $(\overline{v}, \overline{w}, f)$ is an edge from $\overline{G}$ and from $\overline{G}_{i_0}$ and, moreover if $\overline{w} \in \mathcal{V}(\overline{G}')$, then $\varsigma(\overline{w}) = \varsigma_{i_0}(\overline{w})$,
  - or an edge $(\alpha(\overline{v}), \alpha(\overline{\iota}), \langle f, t \rangle)$ exists in G.

and we have the same properties for each $\overline{Q}_{j_l} = ((i, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ that belongs to P'.

Thus, before the execution of the instruction $ipd(i_0)$, we have these properties for $\overline{Q}_{ipd(i_0)}$ and $\overline{Q}'_{ipd(i_0)}$ and then it is obvious that $\overline{Q}_{ipd(i_0)} \overset{Q_{ipd(i_0)},\beta}{\longleftrightarrow} \overline{Q}'_{ipd(i_0)}$.

Since the rest of T and T' are shortest than T and T', we are done. □

We can now conclude with the proof of the non-interference theorem.

Proof of Theorem 6.10. Let B be an instruction block. Let $\overline{Q_1} = ((i, \overline{\rho_1}, \overline{s_1}) :: fr_1, \overline{G_1})$ be a concrete state, $Q_1 = (\rho_1, s_1, G_1)$ an abstract state such that $\overline{Q_1} \lhd^{\alpha} Q_1$, $\overline{Q_2} = \overline{instr}_B(\overline{Q_1})$, and $Q_2 = instr_B(Q_1, \Gamma_1)$. Let $\overline{o} \in \mathcal{O}(\overline{G_1})$ be a reference node and $\overline{\iota} \in \mathcal{I}(\overline{Q_1})$ be an input value. Let us consider a value-change $\overline{Q'_1} = ((i, \overline{\rho'_1}, \overline{s_1}) :: fr_1, \overline{G'_1})$ of $\overline{Q_1}$ and $\overline{\iota}$. According to Definition 6.3, either $\overline{\iota}$ is a value node of $\overline{G_1}$ and then $\overline{G'_1} = \overline{G_1}[\overline{\iota} \mapsto \overline{v}]$, or $\overline{\iota}$ is a local variable and then $\overline{\rho'_1} = \overline{\rho_1}[\overline{\iota} \mapsto \overline{v}]$ (for some $\overline{v} \in Val$). Let us denote $\beta = \alpha_{Q_1}(\overline{\iota})$, then, the first case corresponds to the graph variation rule in Definition 6.16, the second case to the local variables variation rule of Definition 6.16, thus $\overline{Q'_1} \overset{Q_1,\beta}{\longleftrightarrow} \overline{Q_1}$. We denote $\overline{Q'_2} = \overline{instr}_B(\overline{Q'_1})$. According to Proposition 6.18, we have $\overline{Q'_2} \overset{Q_2,\beta}{\longleftrightarrow} \overline{Q_2}$. As $free_{\alpha}(\alpha(\overline{o}), \alpha_{\overline{Q_1}}(\overline{\iota}), \overline{Q_1})$, graph variation rule of Definition 6.16 says that $\overline{o} \in \mathcal{O}(\overline{G'_1}) \cap \mathcal{O}(\overline{G'_2})$. As by hypothesis, $Reach_{G_2}(\alpha(\overline{o}) \cap \alpha_{\overline{Q_1}}(\overline{\iota})) = \emptyset$, by applying Lemma 6.17, we obtain $\overline{G_2}\lfloor\overline{o}\rfloor \equiv \overline{G'_2}\lfloor\overline{o}\rfloor$ (for $\overline{G_2}, \overline{G'_2}$ the memory graphs of resp. $\overline{Q_2}, \overline{Q'_2}$), ie $\overline{\iota}$ does not interfer with $\overline{o}$ in B. □

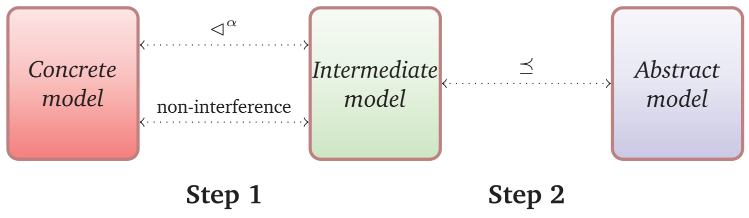


Figure 6.6: Proof schema for inter-procedural analysis

6.6 Soundness of inter-procedural analysis: intermediate model

Showing the correctness of the compositional method invocation directly has proved difficult because of the differences between the concrete and abstract rules; hence, we have chosen a proof in two steps (Proof schema is depicted in Figure 6.6):

Step 1: we introduce an intermediate layer, which is similar to the abstract layer except for the inter-procedural part. This layer is closer to the concrete layer than the abstract layer because it unfolds the inter-procedural control flow graph (ICFG), and steps into method's body, as in a concrete execution. The intermediate layer is proved correct w.r.t. the concrete layer.

Step 2: we prove the correctness of the abstract layer by showing that it gives at least the same results as the intermediate layer. Hence, by transitivity, the abstract layer is correct w.r.t. the concrete layer.

In this section, we present the intermediate layer and we prove its soundness. The intermediate layer is similar to the abstract layer, but handles method invocation differently. Hence, for each bytecode b we define an intermediate rule, denoted by $\widehat{instr_b}$ in the following way:

$$\widehat{instr_b} = \begin{cases} \widehat{\texttt{invoke}} & \text{if } b = \texttt{invoke} \\ \alpha\widehat{\texttt{return}} & \text{if } b = \alpha\texttt{return} \\ instr_b & \text{otherwise} \end{cases}$$

where $\widehat{\texttt{invoke}}$ and $\alpha\widehat{\texttt{return}}$ are the transformation rules that we define in the following paragraphs, and $instr_b$ are the transformation rules of the abstract model, as defined in the intra-procedural analysis in Figure 5.5, page 112. Intuitively, $\widehat{\texttt{invoke}}$ steps into the callee, while $\alpha\widehat{\texttt{return}}$ returns from the callee to the caller.

We first show how the inter-procedural control flow graph is built, then we give the sematics rules for $\widehat{\texttt{invoke}}$ and $\alpha\widehat{\texttt{return}}$. Finally, we prove the correctness of the intermediate model w.r.t. the non-interference, by showing that the two bytecodes that perform method invocation have the same properties as the rest of the bytecode (monotonicity, points-to correctness, state variation correctness).

6.6.1 Construction of the inter-procedural control flow graph

We now present some details about the construction of the inter-procedural control flow graph that we use. The inter-procedural control flow graph [CBC93, LR04] consists of connecting isolated control flow graphs of single methods, by adding edges from the call site of a method m (*e.g.*, instruction i) to its first instruction, and from the return instruction of the callee m to the next instruction in the caller (*e.g.*, $i+1$).

We use the same framework and the same equation system as in the intra-method analysis and we perform a context insensitive analysis. We only have to define the inter-procedural control flow graph.

Context-insensitive ICFG We use an interprocedural control flow graph (ICFG), that is to say, the union of the control flow graphs of all the methods. To built the ICFG, we need to keep disjoint instruction number sets for methods and unique node names in the AMG, hence we use the extended naming strategy defined in the inter-procedural analysis of the abstract model. Therefore, in node names n^c_{pc} and n^n_{pc}, pc encodes the fully qualified name of the method and the line number.

Then, the ICFG of a set of methods $\mathcal{M}$ is a graph built by connecting each `invoke` instruction to the first node of the invoked method and each α`return` to the successor of `invoke` in the calling method.

Overriding and dynamic binding makes almost impossible to guess statically which one of the possible callees is called in a specific execution of the `invoke` instruction. Our analysis conservatively takes into account all possible callees. Let us denote by $m' \leq m$ if m' overrides m. Then, each call site `invoke` m is connected to all methods m' such that $m' \leq m$.

Formally, if $CF_m = (V_m, E_m)$ is the control flow graph of a method m, and $ICFG_{\mathcal{M}} = (V_{\mathcal{M}}, E_{\mathcal{M}})$, then

$$\begin{aligned}
V_{\mathcal{M}} &= \bigcup_{m \in \mathcal{M}} V_m \\
E_{\mathcal{M}} &= \bigcup_{m \in \mathcal{M}} \{(u,v) \mid (u,v) \in E_m, P_m[u] \neq \texttt{invoke m'}\} && (6.6.1)\\
&\cup \ \{(m_1^i, m_{2'}^0) \mid m_1, m_2, m_{2'} \in \mathcal{M}, P_{m_1}[m_1^i] = \texttt{invoke m}_2, m_{2'} \leq m_2\} && (6.6.2)\\
&\cup \ \{(m_{2'}^l, m_1^j) \mid m_1, m_2, m_{2'} \in \mathcal{M}, P_{m_1}[m_1^i] = \texttt{invoke m}_2, \\
&\qquad j = succ_{CF_{m_1}}(m_1^i), P_{m_{2'}}[m_{2'}^l] = \alpha\texttt{return}, m_{2'} \leq m_2\} && (6.6.3)
\end{aligned}$$

The set of nodes of the graph is simply the union of all nodes in the control flows graph of methods $m \in \mathcal{M}$. The edges are built in the following way: equation (6.6.1) keeps all edges except those starting from `invoke` instructions; (6.6.2) adds an edge from each call site m_1^i to the first instruction of all possible invoked methods ($m_{2'}^0$ such that $m_{2'} \leq m_2$); (6.6.3) connects each α`return` instruction in methods $m_{2'} \leq m_2$ to the successor j of each call site invoking m_2.

Example 6.2. Figure 6.7 depicts the inter-procedural control flow graph ICFG for a set of methods $\mathcal{M} = \{m_1, m_2\}$. Note that both call sites for method m_2, m_1^i and m_1^k, connect to the same node, m_2^0. Moreover, the method m_2 is recursive, hence it loops on itself.

6.6.2 Abstract semantics of the method invocation

Now that we have defined the construction of the inter-procedural call graph, we show how the connection between control flow graphs of different methods is integrated in our framework.

To deal with method calls, the JVM keeps a *method call stack*, which contains the frames of methods in the execution chain. When a method m invokes m', the frame of m is pushed on the call stack (in order to remember the return point) and a new frame, which becomes the current frame, is created for m'. When m' finishes its execution, the last frame on the call stack is being popped and it becomes the current frame. Hence, the call stack is very important as it remember the JVM context under which a method is called and the return point when a method finishes executing.

In static analysis of inter-procedural programs, one of the main problems is that the size of the call stack in the concrete execution is not bounded. Thus, with finite abstract domains, the abstract

(a) Code

```
m1
    0:   ...
    i:   invoke m2
    j:   ...
    k:   invoke m2
    l:   areturn


m2
    0:   ...
    i:   ifcpm k
    j:   invoke m2
    j+1: goto l
    k:   ...
    l: areturn
```

(b) ICFG

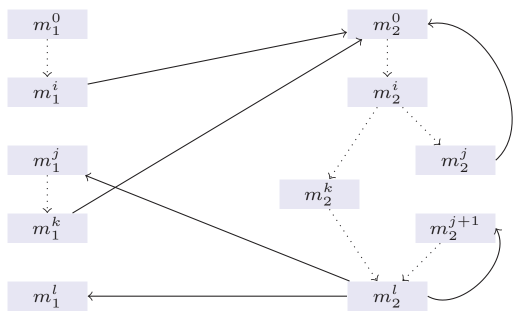


Figure 6.7: Context-insensitive inter-procedural control flow graph example

rule for `invoke` cannot stick to the operational one, as we cannot define a finite abstract call stack. Some works of the literature propose an abstraction of the call stack [JS04].

(a) Original code

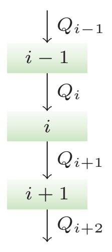
```
i-1  :  ...
i    :  invoke m
i+1  :  ...
```

(b) Transformed code

```
i-1         :  ...
i'          :  bipush 0
a_0         :  ifeq c_0
b_0         :  pop
            :  ...
b_{n_m}     :  pop
b_{n_m+1}   :  aconst_null
b_{n_m+2}   :  goto h(i+1)
c_0         :  invoke m
h(i+1)      :  ...
```

$$
\begin{aligned}
Q_i &= (\rho, u_{n_m} :: \cdots :: u_0 :: s, G)\\
Q_{a_0} &= (\rho, 0 :: u_{n_m} :: \cdots :: u_0 :: s, G)\\
Q_{b_0} &= (\rho, u_{n_m-1} :: \cdots :: u_0 :: s, G)\\
Q_{c_0} &= (\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G)\\
Q_h &= (\rho, \texttt{null} :: s, G)\\
&\Downarrow\\
Q_{i+1} &= (\zeta, u, G') \sqcup Q_h
\end{aligned}
$$

(c) Original Control Flow



(d) Transformed Control Flow

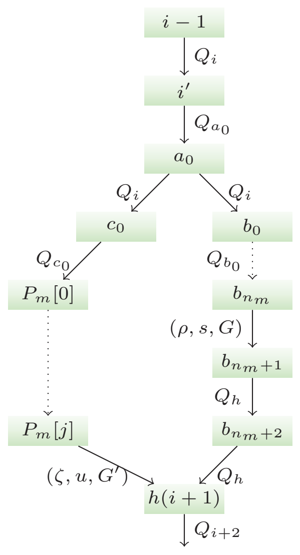


Figure 6.8: Bytecode transformation

To overcome this limitation, we chose an analysis *without any abstraction of the call stack*, thus we use a code transformation which keeps the same finite abstract domain as in the intra-method analysis and which "remembers" the context under which a method was called. To achieve this goal, we replace each `invoke` bytecode by an "`if`" region as depicted in Figure 6.8: in one branch we execute method invocation, while the other branch "remembers" the context under which the method was called. The junction of `if` merges the return of the invoked method (*i.e.*, the heap and the return value) with the initial context.

Moreover, this transformation does not need any new instruction. We only use existing instructions. The only thing needed is to define "adapted" abstract semantics rules for `invoke` and α`return` bytecodes which connect the control flow graphs of methods (where ζ is a special value):

$$\frac{(\rho, u_{n_m} :: \cdots :: u_0 :: s, G)}{(\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G)}\ \widehat{\texttt{invoke}}\ \texttt{m} \qquad \frac{(\rho, u :: s, G)}{(\zeta, u, G)}\ \widehat{\alpha\texttt{return}}$$

This simply means that we connect each $\widehat{\texttt{invoke}}$ instruction to the first node of the invoked method and each $\widehat{\alpha\texttt{return}}$ to the node following the invocation in the calling method. In this way, we occult the call stack and only consider the current frame of the method.

This transformation is syntactical and only used for technical reasons in the analysis of the abstract model: we insert some code and rename the following bytecode numbers with a function h. It does not change the semantics of the program since we only add dead code: the path $b_0 \ldots b_{n_m+2}$ is never executed and is used to "transmit" the current context (the context we had before the invocation), to the immediate successor of the invocation, thus simulating the context save and reload that occur in the concrete execution of an invocation. The equivalent of a context reload is done by the join operator when computing $Q_{h(i+1)}$. We extend the operator to the case where the local variables have the special value ζ: $(\zeta, u, G) \sqcup (\rho, v :: s, G') = (\rho, v :: s, G') \sqcup (\zeta, u, G) = (\rho, u :: s, G \sqcup G')$, thus $Q_{h(i+1)} = (\rho, u :: s, G')$ (since $G \sqcup G' = G'$), which contains the local variables of Q_{a_0}, the stack of Q_{a_0} increased with the return value of `m`, and the memory resulting from the execution of `m`.

This is exactly the state we would obtain by saving the calling context and reloading it after the method execution. Note that the extension of the join operator does not affect the previous uses of the operator since we only introduce the special value ζ in the $\widehat{\alpha\texttt{return}}$ rule. Moreover to have a complete definition of $\sqcup$, we can define the last case: $(\zeta, u, G) \sqcup (\zeta, u', G') = (\zeta, u \sqcup u', G \sqcup G')$.

It is obvious that any concrete execution $Q_0 Q_1 \ldots Q_{i-1} Q_i Q'_i Q_{P_m[0]} \ldots Q_{P_m[j]} Q_{i+1}$ of the original program corresponds to an execution $Q_0 Q_1 \ldots Q_{i-1} Q_{i'} Q_{a_0} Q_{c_0} Q_{P_m[0]} \ldots Q_{P_m[j]} Q_{h(i+1)}$ with $Q_{i+1} = Q_{h(i+1)}$. Thus, we obtain the same state that we would obtain by saving the calling context and reloading it after the method execution.

Note that we keep the usual Java properties: the stack is well typed and has always a fixed size at each program point.

6.6.3 Correctness

The intermediate model is based on the inter-procedural control graph and on a code transformation in order to avoid unbounded call stack. The code transformation fits the previous framework, as it relies on instructions already proved correct and on two new instructions: $\widehat{\texttt{invoke}}$ and $\widehat{\alpha\texttt{return}}$. Hence, in order to show that the intermediate abstract model is sound with respect to the non-interference, it is sufficient to prove that the abstract semantics rules for $\widehat{\texttt{invoke}}$ and $\widehat{\alpha\texttt{return}}$ have the same properties as the rest of bytecode. We only have to define an extension of the relation $\lhd^\alpha$ for an abstraction relation α taking into account the new value ζ.

Definition 6.20 (State abstraction (extension))**.** Let α be an abstraction function. Given a concrete state $\overline{Q} = ((pc, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ and an abstract state $Q = (\rho, s, G)$, Q is an α-abstraction of $\overline{Q}$ (denoted by $\overline{Q} \lhd^\alpha Q$) if $\overline{G} \lhd^\alpha G$ and:

- $\overline{v} \lhd^\alpha s$ with $s = v :: s'$ if $\rho = \zeta$
- $\overline{s} \lhd^\alpha s$, and $\overline{\rho} \lhd^\alpha \rho$, otherwise.

The extension is necessary for the α`return` bytecode, in order to prove the relation $\lhd^\alpha$ for the value returned by the invoked method. This definition is sufficient, since the relation $\lhd^\alpha$ for the rest of the calling context is being insured by the code transformation presented in the previous paragraphs.

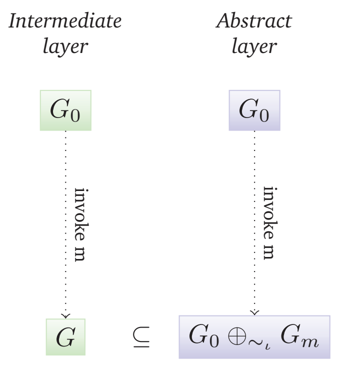


Figure 6.9: Relation between intermediate model and abstract model

For the same reasons, we need to define an extension for the state variation definition: for the α`return` bytecode, and thus the special value ζ, the state variation must hold only for the returned value.

Definition 6.21 (State variation (extension)). Let α be an abstraction function. Let $\overline{Q} = ((pc, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ be a concrete state, $Q = (\rho, s, G)$ an abstract state such that $\overline{Q} \lhd^\alpha Q$. Let $\beta \subseteq \mathcal{V}(G)$ be a set of primitive nodes. Then $\overline{Q'}$ is a **state variation** of $\overline{Q}$ with respect to Q and β, denoted by $\overline{Q}' = ((i, \overline{\rho}', \overline{s}') :: fr, \overline{G}') \overset{Q,\beta}{\longleftrightarrow} \overline{Q}$, if $\overline{G}' \overset{G,\beta}{\longleftrightarrow} \overline{G}$ and:

- $\overline{v}' \overset{s,\beta}{\longleftrightarrow} \overline{v}$ with $\overline{s} = \overline{v} :: \overline{s}''$ and $\overline{s}' = \overline{v}' :: \overline{s}'''$ if $\rho = \zeta$
- $\overline{s'} \overset{s,\beta}{\longleftrightarrow} \overline{s}$, and $\overline{\rho'} \overset{\rho,\beta}{\longleftrightarrow} \overline{\rho}$, otherwise.

We still need to prove that the abstract semantics rules for $\widehat{\texttt{invoke}}$ and $\widehat{\alpha\texttt{return}}$ have the same properties as the rest of bytecode: the monotonicity (cf. Lemma 5.5 page 115), the local soundness for the points-to correctness (Proposition 6.11 at page 142), the $\doteq$ relation (Propositon 6.13 at page 143) needed to prove the points-to correctness for blocks, and the state variation correctness (Proposition 6.18 at page 145).

The proofs are done on case analysis for each bytecode, and are similar to the proofs for other bytecodes, therefore we detail them in Appendix A.4, page 173.

6.7 Soundness of inter-procedural analysis: abstract model (compositional)

Proof outline This section is dedicated to the Step 2 (Figure 6.6, page 147) of the correctness proof for the inter-procedural analysis. While in the previous section we show the correctness of the intermediate layer (which is based on ICFG), here we prove the correctness of the abstract layer (compositional approach) by showing that the abstract layer gives at least the same results as the intermediate layer.

Intuitively, as illustrated in Figure 6.9, if G_0 is the AMG in the abstract state before the method call, G the AMG obtained in the intermediate layer, G_m the AMG of invoked method, $\sim_\iota$ a initial mapping relation between G_0 and G_m then:

$$G \subseteq G_0 \oplus_{\sim_\iota} G_m$$

where $G_0 \oplus_{\sim_\iota} G_m$ is the graph obtained after method invocation in the abstract layer.

For simplicity, we redefine the relation above using the $\preceq$ operator in the following way:

Definition 6.22 (The $\preceq^{\sim_\iota}$ relation for AMGs)**.** Given three AMGs, $G = (V, E \cup E^c)$, $G' = (V', E' \cup E^{c\prime})$ [3] and G_0, such that $G_0 \subseteq G$ and an initial relation $\sim_\iota \subseteq V \times V'$, then:

$$G \preceq^{\sim_\iota}_{G_0} G' \qquad \text{if} \qquad G \subseteq G_0 \oplus_{\sim_\iota} G'.$$

In order to prove the soundness of abstract layer:

1. we first prove that it fits the equation system $\mathcal{E}_B$, hence the $\preceq$ relation respects the following two properties:

   the monotonicity of abstract transfer functions w.r.t. to $\preceq$ relation : if $G \preceq^{\sim_\iota}_{G_0} G'$ then $instr_b(G) \preceq^{\sim_\iota}_{G_0} instr_b(G')$ [4]. The monotonicity enforces correctness at bytecode level, by showing that executing apart each instruction and composing the result with the state before execution G gives at least the same result as if the instruction was applied on G.

   the preservation of union : if $G_1 \preceq^{\sim_\iota}_{G_0} G'_1$ and $G_2 \preceq^{\sim_\iota}_{G_0} G'_2$ then $G_1 \cup G_2 \preceq^{\sim_\iota}_{G_0} G'_1 \cup G'_2$.

   As the AMG, stack and local variables interplay, the relation $\preceq$, and the two properties above must be extended to the stack and local variables array, and hence to the entire abstract state. In Section 6.7.1 we define formally the $\preceq$ relation on abstract states, then we proof in Section 6.7.2 some technical lemmas, and finally, in Section 6.7.3, we proof the two properties described above.

2. then, in Section 6.7.4, we compare the abstract layer to the intermediate layer by showing that, given the same initial state Q_0 before method invocation, the result of method invocation in the intermediate layer, *i.e.*, Q_1, and the result of method invocation in the abstract layer, *i.e.*, Q_2, then $Q_1 \sqsubseteq Q_2$.

6.7.1 Definition of the $\preceq$ relation

Let us now give a complete definition of $\preceq$ relation. We first define the relation $\preceq$ on sets of abstract values. If $\sim \subseteq A \times B \times \mathcal{F}$, we first define the $\preceq$ relations on sets in A and B (this is needed to define the order relation on Γ), and secondly we define the $\preceq$ relation on sets in $A \times \mathcal{F}$ and $B \times \mathcal{F}$ (this relation is needed to order the stack and the local variables array).

Definition 6.23 (The $\preceq$ relation on simple sets)**.** Given two sets A and B and two subsets, $A' \subseteq A$ and $B' \subseteq B$, and a mapping relation $\sim \subseteq A \times B \times \mathcal{F}$, we say that $A' \preceq^{\sim}_{t} B'$ if for all $a \in A'$ there exists $b \in B'$ such that $a \overset{t}{\sim} b$.

Definition 6.24 (The $\preceq$ relation on typed sets)**.** Given two sets A and B and two subsets, $A' \subseteq (A \times \mathcal{F})$ and $B' \subseteq (B \times \mathcal{F})$, and a mapping relation $\sim \subseteq A \times B \times \mathcal{F}$, we say that $A' \preceq^{\sim} B'$ if:

1. for all $(a, \mathtt{d}) \in A'$ there exists $(b, \mathtt{d}) \in B'$ such that $a \overset{\mathtt{d}}{\sim} b$,
2. for all $(a, \mathtt{i}) \in A'$
   a) either there exists $(b, \mathtt{i}) \in B'$ such that $a \overset{\mathtt{i}}{\sim} b$,
   b) or there exists $(b, \mathtt{d}) \in B'$ and $(a', \mathtt{d}) \in A'$ such that $a \overset{\mathtt{i}}{\sim} b$ and $a' \overset{\mathtt{d}}{\sim} b$.

[3]Recall that, for technical reasons, during the inter-procedural analysis we define the AMG as $G = (V, E \cup E^c)$ where E denotes the edges from the initial graph while E^c the edges generated by the abstract semantics rules and by the composition.

[4]$instr_b$ is defined on abstract states and not on graph. As the $\preceq$ relation has not yet been defined on abstract states, we use here $instr_b(G)$, to ease the comprehension.

The $\preceq$ relation on typed sets is needed to define the $\preceq$ relation on stack and local variables array. Rule 1 of Definition 6.24 stands for alias (direct) mapping, while rule 2 takes into account the implicit flow; an element $(a, \mathtt{i})$ can be found on the stack in two cases:

- a belongs to the context of execution ($a \in \Gamma$), and, in this case, a must be mapped through implicit flow to an element b which also belongs to the context of execution (rule (2a): there exists $(b, \mathtt{i}) \in B'$ such that $a \overset{\mathtt{i}}{\sim} b$),
- an object o implicitly depends on a; this means that there exists an object o', two edges $(o', o, \langle f, \mathtt{d}\rangle)$ and $(o', a, \langle f, \mathtt{i}\rangle)$. In the same time, the mapping relation says that all elements b directly mapped to o ($o \overset{\mathtt{d}}{\sim} b$), are also mapped to a ($a \overset{\mathtt{i}}{\sim} b$). If the code is executed inline[5], the instruction `putfield` will load $(o, \mathtt{d})$ and $(a, \mathtt{i})$ on the stack, while, if the code is executed by a method, the instruction will only load b on the stack. The rule (2b) describes this situation: if $(a, \mathtt{i})$ is on the stack, then the object that implicitly depends on a must also be on the stack.

Using the relation $\preceq$ on typed sets of elements, we can now define the $\preceq$ relation for stacks, which is a component wise operation:

Definition 6.25 (The $\preceq$ relation on stacks)**.** Given two stacks, s_1 and s_2, with elements in $S_1 \times \mathcal{F}$ and $S_2 \times \mathcal{F}$ respectively, and a mapping relation $\sim \subseteq S_1 \times S_2 \times \mathcal{F}$, we say that $s_1 \preceq^{\sim} s_2$ if both stacks are empty or, for $s_1 = u_1 :: s'_1$ and $s_2 = u_2 :: s'_2$, $u_1 \preceq^{\sim} u_2$ and $s'_1 \preceq^{\sim} s'_2$.

The $\preceq$ relation on local variables arrays is defined similarly: two local variables arrays are in $\preceq$ relation if each component of the array respect this relation.

Definition 6.26 (The $\preceq$ relation for abstract states)**.** Given three abstract states $Q = (\rho, s, G = (V, E \cup E^c))$, $Q' = (\rho', s', G' = (V', E' \cup E^{c\prime}))$ and $Q_0 = (\rho_0, s_0, G_0 = (V, E_0 \cup E^c{}_0))$, such that $Q_0 \sqsubseteq Q$, and an initial relation $\sim_\iota \subseteq V \times V' \times \mathcal{F}$, we say that $Q \preceq^{\sim_\iota}_{Q_0} Q'$, if

1. $s \preceq^{\sim_\iota^{G,G'}} s'$,
2. $\rho \preceq^{\sim_\iota^{G,G'}} \rho'$,
3. $G \preceq^{\sim_\iota}_{G_0} G'$.

6.7.2 Technical lemmas

Lemma 6.27. *Consider two sets A and B, a mapping relation $\sim \subseteq A \times B \times \mathcal{F}$, and four subsets $A', A'' \subseteq (A \times \mathcal{F})$ and $B', B'' \subseteq (B \times \mathcal{F})$.*
If $A' \preceq^{\sim} B'$ and $A'' \preceq^{\sim} B''$, then $(A' \cup A'') \preceq^{\sim} (B' \cup B'')$.

Proof. We show that for all $(u, t) \in (A' \cup A'')$, with $t \in \mathcal{F}$, Definition 6.24 holds. There are two cases:

1. either $(a, t) \in A'$,
2. or $(a, t) \in A''$.

We consider only the first case, the second is similar. As $A' \preceq^{\sim} B'$, then for each $(a, \mathtt{d}) \in A'$ there exists $(b, \mathtt{d}) \in B' \subseteq (B' \cup B'')$ such that $a \overset{\mathtt{d}}{\sim} b$.
Also, for each $(a, \mathtt{i}) \in A \subseteq (A' \cup A'')$

- either there exists $(b, \mathtt{i}) \in B' \subseteq (B' \cup B'')$ such that $a \overset{\mathtt{i}}{\sim} b$,

[5]We use this term to designate code executed during the ICFG analysis.

- or there exist $(b, \mathtt{i}) \in B' \subseteq (B' \cup B'')$, $(a', \mathtt{d}) \in A' \subseteq (A' \cup A'')$ such that $a \overset{\mathtt{i}}{\sim} b$ and $a' \overset{\mathtt{d}}{\sim} b$.

□

Lemma 6.28. *Let $G_1 = (V_1, E_1 \cup E^c{}_1)$, $G'_1 = (V_1, E'_1 \cup E^{c\prime}{}_1)$, $G_2 = (V_2, E_2 \cup E^c{}_2)$, $G'_2 = (V_2, E'_2 \cup E^{c\prime}{}_2)$ be four AMGs and $\sim_\iota$: $V_1 \times V_2 \times \mathcal{F}$ and a mapping relation.*
If $G_1 \subseteq G'_1$ and $G_2 \subseteq G'_2$ then $\sim_\iota{}^{G_1,G_2} \subseteq \sim_\iota^{G'_1,G'_2}$.

Proof. Straightforward, by construction: as $G_1 \subseteq G'_1$ and $G_2 \subseteq G'_2$, then all the mapping relations generated by the closure of $\sim_\iota$ on G_1 and G_2 will also be generated by the closure of $\sim_\iota$ on G'_1 and G'_2 set $\sim_\iota{}^{G_1,G_2}$. □

Lemma 6.29. *Consider three graphs $G_0 = (V_0, E_0 \cup E^c{}_0)$, $G_1 = (V, E_1 \cup E^c{}_1)$, $G_2 = (V, E_2 \cup E^c{}_2)$, and an initial relation $\sim_\iota$: $V_0 \times V \times \mathcal{F}$.*
If $G_1 \subseteq G_2$, then $G_0 \oplus_{\sim_\iota} G_1 \subseteq G_0 \oplus_{\sim_\iota} G_2$.

Proof. Let $G_0 \oplus_{\sim_\iota} G_1 = G'_1 = (V_0, E_0 \cup E^{c\prime}{}_1)$ and $G_0 \oplus_{\sim_\iota} G_2 = G'_2 = (V_0, E_0 \cup E^{c\prime}{}_2)$.
We need to show that $E^{c\prime}{}_1 \subseteq E^{c\prime}{}_2$, that is to say, for each $(u, v, \langle f, t \rangle) \in E^{c\prime}{}_1$, we have $(u, v, \langle f, t \rangle) \in E^{c\prime}{}_2$. If $(u, v, \langle f, t \rangle) \in E^{c\prime}{}_1$ then

- either $(u, v, \langle f, t \rangle) \in E^c{}_0 \subseteq E^{c\prime}{}_2$,
- or the edge was added by the composition, according to Definition 5.9.

If the edge was added by the composition, (by the $\oplus_{\sim_\iota}$ operator applied on G_0 and G_1), then

- either, according to Definition 5.9, there exist $u \overset{\mathtt{d}}{\sim}_\iota{}^{G_0,G_1} u'$, $v \overset{t_2}{\sim}_\iota{}^{G_0,G_1} v'$, $(u', v', \langle f, t_1 \rangle) \in E^{c\prime}{}_1 \subseteq E^{c\prime}{}_2$, with $t = t_1 \sqcap t_2$. Since $G_1 \subseteq G_2$, by applying Lemma 6.28, we obtain $\sim_\iota{}^{G_0,G_1} \subseteq \sim_\iota{}^{G_0,G_2}$. Hence, $u \overset{\mathtt{d}}{\sim}_\iota{}^{G_0,G_2} u'$, $v \overset{t_2}{\sim}_\iota{}^{G_0,G_2} v'$, and, by applying Definition 5.9, $(u, v, \langle, f, t \rangle) \in E^{c\prime}{}_2$,
- or, if $t = \mathtt{i}$ ($(u, v, \langle f, \mathtt{i} \rangle) \in E^{c\prime}{}_1$), then, there exist $u \overset{\mathtt{d}}{\sim}_\iota{}^{G_0,G_1} u'$, $u_1 \overset{t_2}{\sim}_\iota{}^{G_0,G_1} v'$, $v \overset{\mathtt{i}}{\sim}_\iota{}^{G_0,G_1} u'$ $(u', v', \langle f, t_1 \rangle) \in E^{c\prime}{}_1 \subseteq E^{c\prime}{}_2$. Since $\sim_\iota{}^{G_0,G_1} \subseteq \sim_\iota{}^{G_0,G_2}$, we have $u \overset{\mathtt{d}}{\sim}_\iota{}^{G_0,G_2} u'$, $u_1 \overset{t_2}{\sim}_\iota{}^{G_0,G_2} v'$, $v \overset{\mathtt{i}}{\sim}_\iota{}^{G_0,G_2} u'$, and, by applying Definition 5.9, $(u, v, \langle, f, \mathtt{i} \rangle) \in E^{c\prime}{}_2$.

□

6.7.3 Proofs of $\preceq$ properties

Having defined the $\preceq$ on abstract states, we can now prove the properties of the relation: the monotonicity of abstract transfer functions w.r.t. $\preceq$ and the preservation of union by $\preceq$. These properties are needed to integrate the method calls into the equation system $\mathcal{E}_B$.

Lemma 6.30 (Monotonicity of transfer rules w.r.t. to $\preceq$). *Let b be an instruction. For all abstract states $Q = (\rho, s, G = (V, E \cup E^c))$ and $Q' = (\rho', s', G' = (V', E' \cup E^{c\prime}))$, an initial state $Q_0 = (\rho_0, s_0, G_0 = (V, E \cup E^c{}_0))$ such that $Q_0 \sqsubseteq Q$, and an initial relation $\sim_\iota \subseteq V \times V' \times \mathcal{F}$, and for all Γ and Γ' such that $Q \preceq^{\sim_\iota} Q'$ and $\Gamma \preceq^{\sim_\iota}_{\mathtt{i}} \Gamma'$, we have*

$$instr_b(Q, \Gamma) \preceq^{\sim_\iota}_{Q_0} instr_b(Q', \Gamma').$$

Proof. By case analysis, on each instruction type. See Appendix A.5.1 □

Lemma 6.31 (Preservation of union by $\preceq$ relation). *Let $Q_1 = (\rho_1, s_1, G_1 = (V, E_1 \cup E^c{}_1))$, $Q_2 = (\rho_2, s_2, G_2 = (V, E_2 \cup E^c{}_2))$, $Q_1' = (\rho_1', s_1', G_1' = (V', E_1' \cup E^{c\prime}{}_1))$, $Q_2' = (\rho_2', s_2', G_2' = (V', E_2' \cup E^{c\prime}{}_2))$ be four abstract states, an initial abstract state $Q_0 = (\rho_0, s_0, G_0 = (V, E_0 \cup E^c{}_0))$ and an initial relation $\sim_\iota$: $V \times V' \times \mathcal{F}$.*

If $Q_1 \preceq^{\sim_\iota}_{Q_0} Q_1'$ and $Q_2 \preceq^{\sim_\iota}_{Q_0} Q_2'$, then $Q_1 \sqcup Q_2 \preceq^{\sim_\iota}_{Q_0} Q_1' \sqcup Q_2'$

Proof.

$$Q_1 \sqcup Q_2 = (\rho_1 \sqcup \rho_2, s_1 \sqcup s_2, G_1 \cup G_2)$$

$$Q_1' \sqcup Q_2' = (\rho_1' \sqcup \rho_2', s_1' \sqcup s_2', G_1' \cup G_2')$$

From the Definition 5.8 of closure of a mapping relation, ${\sim_\iota}^{G_1,G_1'}$ is the closure of $\sim_\iota$ on G_1 and G_1', while ${\sim_\iota}^{G_2,G_2'}$ is the closure of $\sim_\iota$ on G_2 and G_2'.

From the Definition 6.26:

- if $Q_1 \preceq^{\sim_\iota}_{Q_0} Q_1'$ then $s_1 \preceq^{{\sim_\iota}^{G_1,G_1'}} s_1'$, $\rho_1 \preceq^{{\sim_\iota}^{G_1,G_1'}} \rho_1'$ and $G_1 \preceq^{\sim_\iota}_{G_0} G_1'$,
- if $Q_2 \preceq^{\sim_\iota}_{Q_0} Q_2'$ then $s_2 \preceq^{{\sim_\iota}^{G_2,G_2'}} s_2'$, $\rho_2 \preceq^{{\sim_\iota}^{G_2,G_2'}} \rho_2'$ and $G_2 \preceq^{\sim_\iota}_{G_0} G_2'$,
- $Q_1 \sqcup Q_2 \preceq^{\sim_\iota}_{Q_0} Q_1' \sqcup Q_2'$ if $s_1 \sqcup s_2 \preceq^{{\sim_\iota}^{G_1 \cup G_2, G_1' \cup G_2'}} s_1' \sqcup s_2'$, $\rho_1 \sqcup \rho_2 \preceq^{{\sim_\iota}^{G_1 \cup G_2, G_1' \cup G_2'}} \rho_1' \sqcup \rho_2'$ and $G_1 \cup G_2 \preceq^{\sim_\iota}_{G_0} G_1' \cup G_2'$.

From Lemma 6.28, we obtain ${\sim_\iota}^{G_1,G_1'} \subseteq {\sim_\iota}^{G_1 \cup G_2, G_1' \cup G_2'}$, as $G_1 \subseteq G_1 \cup G_2$ and $G_1' \subseteq G_1' \cup G_2'$.

Let us first prove the $\preceq$ relation on local variables array: $\rho_1 \sqcup \rho_2 \preceq^{{\sim_\iota}^{G_1 \cup G_2, G_1' \cup G_2'}} \rho_1' \sqcup \rho_2'$. We show that for all local variables x, $\rho_1(x) \cup \rho_2(x) \preceq^{{\sim_\iota}^{G_1 \cup G_2, G_1' \cup G_2'}} \rho_1'(x) \cup \rho_2'(x)$. Let $(u, t) \in \rho_1(x) \cup \rho_2(x)$; either $(u, t) \in \rho_1(x)$ or $(u, t) \in \rho_2(x)$. We consider the first case, the second one is similar.

If $t = \mathtt{d}$ (hence $(t, \mathtt{d}) \in \rho_1(x)$), and as $\rho_1 \preceq^{{\sim_\iota}^{G_1,G_1'}} \rho_1'$, from Definition 6.24, there exists $(v, \mathtt{d}) \in \rho_1'(x) \subseteq \rho_1'(x) \cup \rho_2'(x)$ such that $u \overset{\mathtt{d}}{\sim}_\iota^{G_1,G_1'} v$. From ${\sim_\iota}^{G_1,G_1'} \subseteq {\sim_\iota}^{G_1 \cup G_2, G_1' \cup G_2'}$, $u \overset{\mathtt{d}}{\sim}_\iota^{G_1 \cup G_2, G_1' \cup G_2'} v$. Hence, for $t = \mathtt{d}$, the Definition 6.24 holds and $\rho_1(x) \cup \rho_2(x) \preceq^{{\sim_\iota}^{G_1 \cup G_2, G_1' \cup G_2'}} \rho_1'(x) \cup \rho_2'(x)$.

If $t = \mathtt{i}$, there are two cases:

- either there exists $(v, \mathtt{i}) \in \rho_1'(x) \subseteq \rho_1'(x) \cup \rho_2'(x)$ such that $u \overset{\mathtt{i}}{\sim}_\iota^{G_1,G_1'} v$. If $u \overset{\mathtt{i}}{\sim}_\iota^{G_1,G_1'} v$, then $u \overset{\mathtt{i}}{\sim}_\iota^{G_1 \cup G_2, G_1' \cup G_2'} v$ and the definition holds,
- or there exists $(v, \mathtt{d}) \in \rho_1'(x) \subseteq \rho_1'(x) \cup \rho_2'(x)$ and $(u', \mathtt{d}) \in \rho_1(x) \subseteq \rho_1(x) \cup \rho_2(x)$ such that $u \overset{\mathtt{i}}{\sim}_\iota^{G_1,G_1'} v$ and $u' \overset{\mathtt{d}}{\sim}_\iota^{G_1,G_1'} v$. Hence $u \overset{\mathtt{i}}{\sim}_\iota^{G_1 \cup G_2, G_1' \cup G_2'} v$ and $u' \overset{\mathtt{d}}{\sim}_\iota^{G_1 \cup G_2, G_1' \cup G_2'} v$ and $\rho_1(x) \cup \rho_2(x) \preceq^{{\sim_\iota}^{G_1 \cup G_2, G_1' \cup G_2'}} \rho_1'(x) \cup \rho_2'(x)$. □

The proof for stack is similar to local variables array and will not be detailed here.

Let us now proof the $\preceq$ relation for graphs: $G_1 \cup G_2 \preceq^{\sim_\iota}_{G_0} G_1' \cup G_2'$, which corresponds to $G_1 \cup G_2 \subseteq G_0 \oplus_{\sim_\iota} (G_1' \cup G_2')$.

Let

$G_1 = (V, E_1 \cup E^c{}_1)$ $\quad G_2 = (V, E_2 \cup E^c{}_2)$ $\quad G_1 \cup G_2 = (V, (E_1 \cup E_2) \cup (E^c{}_1 \cup E^c{}_2))$

$G_1' = (V', E_1' \cup E^{c\prime}{}_1)$ $\quad G_2' = (V', E_2' \cup E^{c\prime}{}_2)$ $\quad G_1' \cup G_2' = (V', (E_1' \cup E_2') \cup (E^{c\prime}{}_1 \cup E^{c\prime}{}_2))$

Let $G' = G_0 \oplus_{\sim_\iota} (G_1' \cup G_2') = (V, E' \cup E^{c\prime})$. By definition of $\oplus$, $E' = E_0$.

We need to show that $E_1 \cup E_2 \subseteq E'$ and $E^c{}_1 \cup E^c{}_2 \subseteq E^{c\prime}$.

Let $\begin{array}{l} G_0 \oplus_{\sim_\iota} G_1' = (V, E_0 \cup E^{c\prime}{}_{01}) \\ G_0 \oplus_{\sim_\iota} G_2' = (V, E_0 \cup E^{c\prime}{}_{02}). \end{array}$

(a) The $\preceq$ relation on blocks and abstract states

$$\begin{tikzcd}
(Q,\Gamma) \arrow[r, phantom, "\preceq^{\sim_\iota}_{Q_0}"] \arrow[d, dotted, "instr_B"] & (Q',\Gamma') \arrow[d, dotted, "instr_B"] \\
instr_B(Q,\Gamma) \arrow[r, phantom, "\preceq^{\sim_\iota}_{Q_0}"] & instr_B(Q',\Gamma')
\end{tikzcd}$$

(b) The $\preceq$ relation on blocks and AMGs

$$\begin{tikzcd}
G \arrow[r, phantom, "\subseteq"] \arrow[d, dotted, "instr_B"] & G_0 \oplus_{\sim_\iota} G' \arrow[d, dotted, "instr_B"] \\
instr_B(G) \arrow[r, phantom, "\subseteq"] & G_0 \oplus_{\sim_\iota} instr_B(G')
\end{tikzcd}$$

Figure 6.10: The $\preceq$ relation on blocks and abstract states

By hypothesis, $\begin{array}{lcl} G_1 \subseteq G_0 \oplus_{\sim_\iota} G'_1 & \rightarrow & E_1 \subseteq E_0 \\ G_2 \subseteq G_0 \oplus_{\sim_\iota} G'_2 & \rightarrow & E_2 \subseteq E_0. \end{array}$

Hence, $E_1 \cup E_2 \subseteq E_0$, and as $E' = E_0$, then $E_1 \cup E_2 \subseteq E'$.

Moreover, ${E^c}_1 \subseteq {E^c}'_{01}$ and ${E^c}_2 \subseteq {E^c}'_{02}$.

Let us now show that ${E^c}_1 \cup {E^c}_2 \subseteq {E^c}'$: for each $(u, v, \langle f, t\rangle) \in {E^c}_1 \cup {E^c}_2$, then $(u, v, \langle f, t\rangle) \in {E^c}'$. Let us consider the case when $(u, v, \langle f, t\rangle) \in {E^c}_1$ (the case $(u, v, \langle f, t\rangle) \in {E^c}_2$ is similar). From Lemma 6.29, $G_0 \oplus_{\sim_\iota} G'_1 \subseteq G_0 \oplus_{\sim_\iota} (G'_1 \cup G'_2)$. If $(u, v, \langle f, t\rangle) \in {E^c}_1 \subseteq {E^c}'_{01}$ then $(u, v, \langle f, t\rangle) \in {E^c}'$. □

We recall that our methods are defined as blocks. From the two properties above, we can state the monotonicity of the $\preceq$ property on blocks.

Lemma 6.32 (Monotonicity of transfer rules w.r.t. to $\preceq$ and instruction blocks)**.** *Let B be a block. For all abstract states $Q = (\rho, s, G = (V, E \cup E^c))$ and $Q' = (\rho', s', G' = (V', E' \cup {E^c}'))$, an initial state $Q_0 = (\rho_0, s_0, G_0 = (V, E \cup {E^c}_0))$ such that $Q_0 \sqsubseteq Q$, and an initial relation $\sim_\iota \subseteq V \times V' \times \mathcal{F}$, and for all Γ and Γ' such that $Q \preceq^{\sim_\iota} Q'$ and $\Gamma \preceq^{\sim_\iota}_{\mathbf{i}} \Gamma'$, we have*

$$instr_B(Q, \Gamma) \preceq^{\sim_\iota}_{Q_0} instr_B(Q', \Gamma').$$

Proof. Straightforward, from Monotonicity on single instruction blocks (Lemma 6.30) and from Lemma 6.31. □

This property is very important, as it states that the relation $\preceq$ on abstract states is preserved on any type of blocks (see Figure 6.10a). Moreover, the $\preceq$ relation on AMGs gives the expected result for a modularity, as depicted in Figure 6.10b: the graph obtained by executing block B from an initial graph G [6], such that $G \subseteq G_0 \oplus_{\sim_\iota} G'$, is included in the graph obtained from the composition of G_0 with $instr_B(G')$.

6.7.4 Relating the intermediate model and abstract model

If we consider that B is the code of a method m, G' is the initial graph of that method and $instr_B(G') = \Theta(m)$, then we can obtain the modularity property for methods and for invoke instruction. But, this cannot be achieved in a single step, due to the fact that in the intermediate model, we made a code transformation in order to avoid unbounded call stack. Hence, to prove the $\preceq$ relation for `invoke` instruction, we make a transformation on the abstract model as depicted in Figure 6.11: this code transformation (depicted in Figure 6.11b) is similar to the one done for the intermediate model, which will allow us to apply Lemma 6.32; the only difference consists of the

[6] Again, for simplicity we restrict $instr_B$ to AMGs, but it works on abstract states.

transformation rule for $\widehat{invoke}$ bytecode. The new semantics, denoted by `invoke⋆`, is similar to the abstract model from the point of view of AMGs, as it combines the AMGs of the caller and the callee, and it is also similar to the intermediate model from the point of view of operand stack and local variables array:

$$\frac{PC_{m_1}[i] = \texttt{invoke}\star\ \ m_2 \qquad (\rho, u_{n_m} :: \cdots :: u_0 :: s, G_1) \qquad G_2 = \Theta(m_2)}{(\zeta, map_{\sim_\iota}(ret(G_2)), G_1 \oplus_{\sim_\iota} G_2)}$$

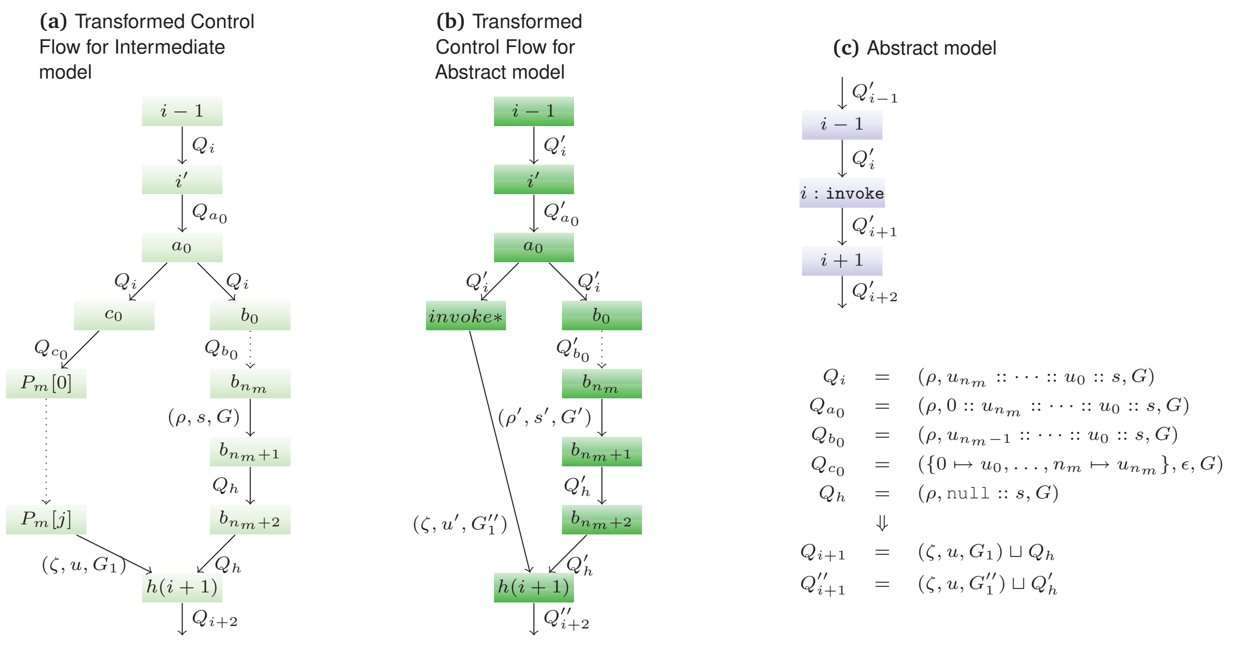

(a) Transformed Control Flow for Intermediate model
(b) Transformed Control Flow for Abstract model
(c) Abstract model
$i-1$
Q_i
i'
Q_{a_0}
a_0
c_0
b_0
Q_{c_0}
Q_{b_0}
$P_m[0]$
b_{n_m}
(ρ, s, G)
b_{n_m+1}
Q_h
$P_m[j]$
b_{n_m+2}
(ζ, u, G_1)
$h(i+1)$
Q_{i+2}
Q'_i
Q'_{a_0}
$invoke\star$
Q'_{b_0}
(ρ', s', G')
Q'_h
(ζ, u', G''_1)
Q''_{i+2}
Q'_{i-1}
$i : \texttt{invoke}$
Q'_{i+1}
$i+1$
Q'_{i+2}
$Q_i = (\rho, u_{n_m} :: \cdots :: u_0 :: s, G)$
$Q_{a_0} = (\rho, 0 :: u_{n_m} :: \cdots :: u_0 :: s, G)$
$Q_{b_0} = (\rho, u_{n_m-1} :: \cdots :: u_0 :: s, G)$
$Q_{c_0} = (\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G)$
$Q_h = (\rho, \texttt{null} :: s, G)$
$\Downarrow$
$Q_{i+1} = (\zeta, u, G_1) \sqcup Q_h$
$Q''_{i+1} = (\zeta, u, G''_1) \sqcup Q'_h$

Figure 6.11: Bytecode transformation

From this code transformation and from Figure 6.11, we can deduce two important results:

1. if $Q_i \preceq^{\sim_\iota}_{Q_0} Q'_i$ and $\Gamma_i \preceq^{\sim_\iota}_{\mathtt{i}} \Gamma'_i$, then, from Lemma 6.32, we obtain that $Q_{i+1} \preceq^{\sim_\iota}_{Q_0} Q''_{i+1}$, where Q_{i+1} is the result of method invocation in the intermediate model and Q''_{i+1} is the result on method invocation in the transformed abstract model;

2. if we apply semantics rule bytecode by bytecode in Figure 6.11b, we obtain that $Q''_{i+1} = Q'_{i+1}$, where Q'_{i+1} is the result of method invocation in the abstract model.

From the two results above, we obtain that, if $Q_i \preceq^{\sim_\iota}_{Q_0} Q'_i$, $\Gamma_i \preceq^{\sim_\iota}_{\mathtt{i}} \Gamma'_i$, m is a method and B the instruction block corresponding to instructions in P_m, $Q'_{i+1} = invoke(Q'_i, \Gamma'_i)$ and $Q_{i+1} = \widehat{instr}_B(Q_i, \Gamma_i)$, then $Q_{i+1} \preceq^{\sim_\iota}_{Q_0} Q'_{i+1}$.
Hence, the compositional approach gives at least the same AMG as the intermediate model.

6.8 Conclusion

In this chapter we proved the soundness of the dependency analysis presented in previous chapter. The soundness of the AMG was splitted in two parts: first, we proved the soundness of the points-to analysis, and then we showed the soundness of the extended graph by stating and proving a non-interference theorem. While the previous models rely on a call graph approach, we present here a modular approach, which is more complex but, in the same time, more adapted to an open environment. Due to its complexity, we have not achieved to prove the soundness of the modular model in a single step. Therefore, we have defined a sound intermediate model, based on ICFG. This model corresponds to the previous approaches based on call graphs. Then, we compare the modular model to the intermediate model and we prove its correctness.

7 Conclusion

Contents

Information flow has been an intensively researched domain during the last decades. But all these efforts lead to a rich theory and a poor utilisation in practice. The main challenge in information flow security is to apply the results in practice and to show their usefulness.

Along this thesis, we have always tried to build a complete solution which addresses information flow issues in embedded systems. We can sum up the goals of this thesis in two major points:

- to give solutions which show the usefulness of information flow analysis. We believe that, in order to successfully apply information flow techniques in practice, both adequate tools and formal proofs for correctness must be provided,
- to adapt this solutions to the specific domain of embedded systems, supporting multiple collaborative applications and dynamic installation.

In the context of open and dynamically evolving systems, the verification must be modular and split in independent parts. Therefore, our models present two essential properties: *(i)* compositionality (*i.e.*, each application can be verified independently) and *(ii)* separation of concerns between the information flow analysis and security policies (*i.e.*, security policies are defined *a posteriori* and changing them does not require to re-analyze the entire system).

The approach followed in this thesis was to provide security guarantees as soon as possible, so that an application can not be installed on the target system if it does not respect the required security properties. In fact, the results of the analysis can be transposed as a typing systems, which allows the use of known verification techniques from this domain. In this sense, we extended the lightweight bytecode verification of Eva Rose and obtained a *lightweight* information flow verification in two steps:

1. first, a phase external to the embedded system, which precisely computes the information flow in terms of types,
2. secondly, the onboard verification of the types previously computed, in order to overcome the complexity of the external algorithm.

This strategy allows the encoding of different properties as a type. In our case, we identify two kind of typing systems: not only the type associated to an abstract value, as usual, but also the type associated to a method. This leads to an analysis more complex than the lightweight bytecode verification. While we applied it here to an information flow analysis, the strategy can be generalized and applied to other security properties, as long as they can be transposed as a type system.

7.1 Contributions

We now detail the contributions of each chapter:

7.1.1 A lightweight information flow model and tool for embedded systems

In Chapters 3 and 4 we addressed security challenges raised by interactions between different software units in the context of a small open environment. We presented a model and a tool dedicated to the target systems. We tried to adapt the model as much as possible to the environment (*i.e.*, limited resources) by making a compromise between the precision of results given an entirely known system and the overhead generated by the analysis onboard (while talking into account the dynamic downloading).

The algorithm is *lightweight*, meaning that, in order to ease the embedded verification, the certification is performed in two phases: an external one, computing information flow information as types, and an embedded one which verifies the computed types. Therefore, the verification is done onboard, the only place where security can be guaranteed, while loading a new software unit.

The embedded verifier was designed as an user-defined class loader, hence it can be used on any system as a plug-in. Apart type verification, it deals with class by class validation (*i.e.*, loaded class needs a class not yet loaded), inheritance (a class is loaded only if it respects the security properties of the parent class) and multiple class loaders.

Security policies The lightweight information flow model enforces non-interference, which a strong security property but, unfortunately, not sufficient to express more complex security policies. To refine non-interference, we defined a domain specific language which describes allowed collaborations between different software units composing the open system. In order to keep the analysis as modular as possible and to avoid code re-analysis, the information flow analysis and security policies enforcement are separated, as policies are specified and verified *a posteriori*, on the target system, based on the results of information flow analysis.

Even if the domain specific language is quite simple, it has enough power to express desired policies in the domain of multiapplicative systems and it can easily be extended to define more complex policies.

As a consequence of the fact that we adapted our model to small open systems, the domain specific language specifies only allowed collaborations *a posteriori*; the sensitive object fields must be specified *a priori* to the information flow analysis, and, for now, we support only two security levels. Increasing the number of security levels will lead to a more precise analysis but more difficult to embed.

Case studies and practical results Throughout our research, we tried to provide a model usable on real Java software and easily integrable on existing platforms. This lead to:

- the development of two functional prototypes, STAN - the external tool computing types, and VSTAN - the embedded verifier defined as a plug-in for any Java Virtual Machine,
- experimental results showing that the tools can be efficiently be used in a domain with restraint resources,
- several practical experiences, especially on Java applications for mobile phones,
- an integrated analysis framework which combines our model with a design security analysis technique.

7.1.2 A sound model for information flow analysis

In Chapter 5 we addressed information flow issues from a theoretical point of view. We defined a formal model which expresses information flow in terms of abstract memory graphs (or AMGs). An AMG is a points-to graph (*i.e.*, a graph which describes relations between references in a program), extended with primitive values and flows arising from primitive assignments and implicit flow. The AMG is computed independently on any security level, hence the analysis is more general than a normal information flow one: several other analysis can be recovered from our model (*e.g.*, alias analysis, purity analysis). Information flow analysis can be considered as a particular case of the framework, obtained by labeling the nodes and the edges with security levels *a posteriori*. The analysis keeps the main features of the embedded model (*i.e.*, modularity, inheritance), the goal being to use it in the context of embedded models.

The model is proved correct with respect to non-interference in Chapter 6. Due to its complexity, the proof schema was split in several parts. First we proved the soundness of the points-to analysis by using a general technique. Then, we proved the correctness of the extended graph by showing a non-interference theorem which states that an output does not depend on an input value if changing the input does not affect the AMG of the output.

The proof of the inter-procedural analysis relies on a composition operator ($\oplus$). We believe that the proof schema can be made generic w.r.t. the composition operator respecting a set of properties.

7.1.3 Discussion

By defining a lightweight analysis for embedded systems and a general sound model, we obtained two models totally opposite in terms of precision:

- the sound model is the most precise model possible, as no approximation and no compromise have been made.
- the embedded model is the "smallest" model possible, pushing the approximations and compromises to the limit. This was imposed by the limited resources of the target systems.

Due to its high precision, the sound model can be declined in a family of models and tools, of different precisions, depending on the target systems and on the needs of the users. The embedded model is the smallest model in this family, with the lowest precision.

7.2 Perspectives

7.2.1 Soundness of the embedded model

As we said before, in order to make information flow analysis appealing, adequate tools must be provided. Moreover, to ensure security, these tools must be proved correct. In the last chapters of this thesis, we define and we prove the soundness of an elegant analysis framework, which we call the dependency model and which is based on AMGs. It is a general framework, as many analysis can be recovered from our model. The embedded model, which we present in the first chapters of this thesis, can also be recovered as an approximation of the dependency model. Hence, to prove the soundness of the embedded model it is enough to prove that it is a correct approximation of the dependency analysis.

The flow signature of a method (which expressed information flows in the embedded model) can be recovered as an approximation of the AMG of a method. In fact, the entire embedded model can be defined as approximation of the dependency model, starting with the abstract domain. As the dependency model has been proved correct w.r.t to non-interference, in order to prove the correctness of the embedded model requires to define an approximation function and to prove its correctness w.r.t. to the AMG.

Hence, proving the soundness of the embedded system is an interesting challenge for two reasons: on the one hand, this will lead to a tool completely proved correct, and on other hand, there are many difficulties that make this approach appealing. The difficulties lies in the approximations of the abstract domain made in the embedded model and the fact that we perform a field-insensitive analysis, while the dependency model unfolds all object fields and performs a field-sensitive and object-sensitive analysis.

7.2.2 Tool support

One direction for future work concerns tool support and case studies.

Reverse Engineering tool The certification must be done onboard due to the fact that the applications are loaded using an unsecured channel and must be adapted to the limited resources of the system. In the same time, the external analysis is supposed to be done on an system offering unlimited resources comparing with a small system. Thus optimization and complexity are not an issue. Moreover, the external resources can be used for other purposes, for example for offering an easy development environment to programmers.

Security must be ensured for different attacks against computing systems, for both deliberate or accidental attacks. Information flow insecurity may arise from malicious, untrusted code or from the programmer's own code. In the later case, the insecurity is due to bad conception of the application or to bad implementation. When the information leak comes from a bad implementation due to human error, it is not always obvious for the developer to correct the application in order to make it safe. The development environment should detect illicit flows and help the developer to correct his mistakes by offering all the necessary information.

The point of failure in the program certification is usually not the real source of information leak. For example, the certification of LoyaltyCard fails while analysing the method `FlyMaroc.makeGetLevel`. But the illegal information flow comes from the implementation of method `getLevel` in class `MHz`, where the computation of fidelity level for `MHz` takes into consideration the points of partners.

To detect the failure source, we can imagine a backward iterative algorithm, which, at each step, tries to detect an information flow in a method. The algorithm is similar to tracking thrown exceptions in Java programs. Let us assume that we have a recursive method $detect(m, f, pc)$ which detects where the flow f occurred in method m by performing a backward analysis starting from the program point pc. If the flow f was created due to another flow f_1, the method $detect(m, f_1, pc)$ is called recursively. If the flow f was created in a method m_1 invoked at pc, the algorithm calls $detect(m_1, f, pc_f)$, where pc_f is the program counter corresponding to the return instruction in method m_1.

This approach is expensive in memory and thus cannot be performed onboard, but it can explore the unlimited resources of the external analyser.

A Addition to proof in Chapter 6

Contents

A.1 Proof of Lemma 5.5 (Monotonicity of the abstract transformation rules)

We consider two pairs (Q_1, Γ_1) and (Q_2, Γ_2) from the property space $\mathcal{S}$ such that $(Q_1, \Gamma_1) \sqsubseteq (Q_2, \Gamma_2)$. Let $Q_1' = instr_b(Q_1, \Gamma_1)$ and $Q_2' = instr_b(Q_2, \Gamma_2)$. We must prove that $Q_1' \sqsubseteq Q_2'$. By TV_Γ we denote
We make a proof by case analysis on each instruction b, based on the transformation rules in Figure 5.5, page 112.

Case: b = **`prim op`**

Let $Q_1 = (\rho_1, v_1 :: v_1' :: s_1, G_1)$ and $Q_2 = (\rho_2, v_2 :: v_2' :: s_2, G_2)$. Since $(Q_1, \Gamma_1) \sqsubseteq (Q_2, \Gamma_2)$, then $\rho_1 \sqsubseteq \rho_2$, $s_1 \sqsubseteq s_2$, $G_1 \subseteq G_2$, $v_1 \subseteq v_2$, $v_1' \subseteq v_2'$ and $TV_{\Gamma_1} \subseteq TV_{\Gamma_2}$. Hence, $(v_1 \cup v_1' \cup TV_{\Gamma_1}) \subseteq (v_2 \cup v_2' \cup TV_{\Gamma_2})$. From the transformation rule, $Q_1' = (\rho_1, v_1 \cup v_1' \cup TV_{\Gamma_1} :: s_1, G_1)$ and $Q_2' = (\rho_2, v_2 \cup v_2' \cup TV_{\Gamma_2} :: s_2, G_2)$, and as the partial order relation holds on each component, we can conclude with $Q_1' \sqsubseteq Q_2'$.

Case: b = **`pop`**

Let $Q_1 = (\rho_1, v_1 :: s_1, G_1)$ and $Q_2 = (\rho_2, v_2 :: s_2, G_2)$, $Q_1' = (\rho_1, s_1, G_1)$ and $Q_2' = (\rho_2, s_2, G_2)$. From hypothesis, $(\rho_1, v_1 :: s_1, G_1) \sqsubseteq (\rho_2, v_2 :: s_2, G_2)$, and thus $(\rho_1, s_1, G_1) \sqsubseteq (\rho_2, s_2, G_2)$.

Case: b = **`ifeq`** a

Let $Q_1 = (\rho_1, v' :: s_1, G_1)$ and $Q_2 = (\rho_2, v'' :: s_2, G_2)$, $Q_1' = (\rho_1, s_1, G_1)$ and $Q_2' = (\rho_2, s_2, G_2)$. From hypothesis, $(\rho_1, v' :: s_1, G_1) \sqsubseteq (\rho_2, v'' :: s_2, G_2)$, and thus $(\rho_1, s_1, G_1) \sqsubseteq (\rho_2, s_2, G_2)$.

Case: b = **goto** a

The abstract semantics rules do not change the input, thus $instr_b(Q_1, \Gamma_1) = Q_1$ and $instr_b(Q_2, \Gamma_2) = Q_2$. Since $Q_1 \sqsubseteq Q_2$, we can conclude straightforward with $instr_b(Q_1, \Gamma_1) \sqsubseteq instr_b(Q_2, \Gamma_2)$.

Case: b = **bipush** n **or** b = **new**iC **or** b = **aconst_null**

The instruction adds an abstract element on the stack (the same abstract element for Q_1 and Q_2). Thus, the monotonicity is preserved.

Case: $b = \alpha$**load x**

From hypothesis, $(\rho_1, s_1, G_1) \sqsubseteq (\rho_2, s_2, G_2)$ and $\Gamma_1 \subseteq \Gamma_2$. Thus, $\rho_1(x) \subseteq \rho_2(x)$, $TV_{\Gamma_1} \subseteq TV_{\Gamma_2}$ and $(\rho_1, \rho_1(x) \cup TV_{\Gamma_1} :: s_1, G_1) \sqsubseteq (\rho_2, \rho_2(x) \cup TV_{\Gamma_2} :: s_2, G_2)$.

Case: $b = \alpha$**store x**

Let $Q_1 = (\rho_1, u_1 :: s_1, G_1)$, $Q_2 = (\rho_2, u_2 :: s_2, G_2)$, $instr_b(Q_1, \Gamma_1) = (\rho_1[x \mapsto u_1'], s_1, G_1)$, where $u_1' = u_1 \cup \{(t, \mathtt{i}) \mid t \in \Gamma_1\}]$, and $instr_b(Q_2, \Gamma_2) = (\rho_2[x \mapsto u_2'], s_2, G_2)$, where $u_2' = u_2 \cup \{(t, \mathtt{i}) \mid t \in \Gamma_2\}]$. To prove the monotonicity, we must show that $u_1' \subseteq u_2'$. As $Q_1 \sqsubseteq Q_2$ and $\Gamma_1 \subseteq \Gamma_2$(from hypothesis), we also have $u_1 \subseteq u_2$ and $\{(t, \mathtt{i}) \mid t \in \Gamma_1\} \subseteq \{(t, \mathtt{i}) \mid t \in \Gamma_2\}$. Hence, $u_1' \subseteq u_2'$ and $instr_b(Q_1, \Gamma_1) \sqsubseteq instr_b(Q_2, \Gamma_2)$.

Case: b = **getfield** $f_{C'}$

Let $Q_1 = (\rho_1, u_1 :: s_1, G_1)$, $Q_2 = (\rho_2, u_2 :: s_2, G_2)$, $instr_b(Q_1, \Gamma_1) = (\rho_1, u_1' :: s_1, G_1)$ and $instr_b(Q_2, \Gamma_2) = (\rho_2, u_2' :: s_2, G_2)$. To prove the monotonicity, we must show that $u_1' \subseteq u_2'$, where $u_1' = \{(e, \mathtt{d}) \mid e \in adj_{G_1}(e', \langle f_{C'}, \mathtt{d}\rangle) \wedge (e', \mathtt{d}) \in u_1\} \cup \{(e, \mathtt{i}) \mid (e, \mathtt{i}) \in u_1\}$. From $u_1 \subseteq u_2$, obviously, $\{(e, \mathtt{i}) \mid (e, \mathtt{i}) \in u_1\} \subseteq \{(e, \mathtt{i}) \mid (e, \mathtt{i}) \in u_2\}$. Let us denote by u_1'' the first set in u_1'. We now show that $u_1'' \subseteq u_2''$, where

$$u_1'' = \{(e, \mathtt{d}) \mid e \in adj_{G_1}(e_1', \langle f_{C'}, \mathtt{d}\rangle) \wedge (e_1', \mathtt{d}) \in u_1\}$$

$$u_2'' = \{(e, \mathtt{d}) \mid e \in adj_{G_2}(e_2', \langle f_{C'}, \mathtt{d}\rangle) \wedge (e_2', \mathtt{d}) \in u_2\}.$$

Since $G_1 \subseteq G_2$ and $u_1 \subseteq u_2$, all the adjacents labeled by $f_{C'}$ of e_1' (where $(e_1', \mathtt{d}) \in u_1$) are also adjacents of e_2' (where $(e_2', \mathtt{d}) \in u_2$). Thus $u_1'' \subseteq u_2''$ and $u_1' \subseteq u_2'$.

Case: b = **putfield** $f_{C'}$

Let $Q_1 = (\rho_1, v_1 :: u_1 :: s_1, (V_1, E_1))$, $Q_2 = (\rho_2, v_2 :: u_2 :: s_2, (V_2, E_2))$. Moreover, let $instr_b(Q_1, \Gamma_1) = (\rho_1, s_1, (V_1, E_1'))$ and $instr_b(Q_2, \Gamma_2) = (\rho_2, s_2, (V_2, E_2'))$. To prove the monotonicity, we must show that $E_1' \subseteq E_2'$. From `putfield` abstract transformation rule in Figure 5.5, $E_1' = E_1 \cup E_1'' \cup E_1'''$, where

$$E_1'' = \{(e, e', \langle f_{C'}, t\rangle) | (e, \mathtt{d}) \in u, (e', t) \in v_1\},$$

$$E_1''' = \{(e, e', \langle f_{C'}, \mathtt{i}\rangle) | (e, \mathtt{d}) \in u_1, e' \in \Gamma \cup V_{\mathtt{i}}^{u_1}\},$$

$$V_{\mathtt{i}}^{u_1} = \{e \mid \langle e, \mathtt{i}\rangle \in u_1\}.$$

The definition of E_2' is similar. By hypothesis, $u_1 \subseteq u_2$ and $v_1 \subseteq v_2$, thus $E_1'' \subseteq E_2''$. Moreover, as $\Gamma_1 \subseteq \Gamma_2$, the inclusion $E_1''' \subseteq E_2'''$ also holds. Thus, $E_1' \subseteq E_2'$.

(a) Source code

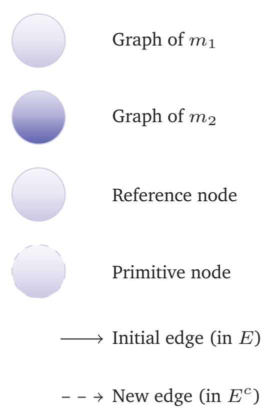
```
class A {
   B b;
   static C s;
   void m1(){
      A v0 = new A();
      B v1 = new B();
      v0.b = v1;
      this.b = v1;
      this.m2(v0);
   }
   ...
}
```

```
class B { C c;  ... }
class C { ... }

class A {
   void m2(A p1){
      B v0 = this.b;
      v0.c = new C();
      p1.b.c=new C();
      A.s = v0.c;
   }
   ...
}
```

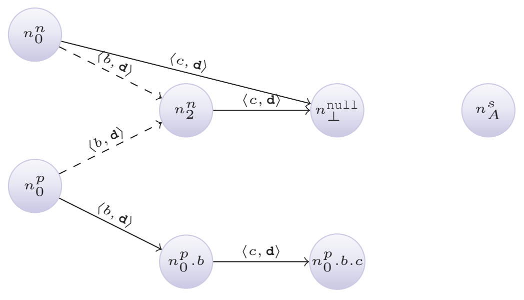
```
m1
  0: new A
  1: astore 1
  2: new B
  3: astore 2
  4: aload 1
  5: aload 2
  6: putfield A_b
  7: aload 0
  8: aload 2
  9: putfield A_b
 10: aload 0
 11: aload 1
 12: invoke A_m2
```

```
m2
  0: aload 0
  1: getfield A_b
  2: astore 2
  3: aload 2
  4: new C
  5: putfield B_c
  6: aload 1
  7: getfield A_b
  8: new C
  9: putfield B_c
 10: aload 2
 11: getfield B_c
 12: putstatic A_s
```

(b) Legend



(c) AMG for m1 before method call, G



(d) AMG for method m2, G_2

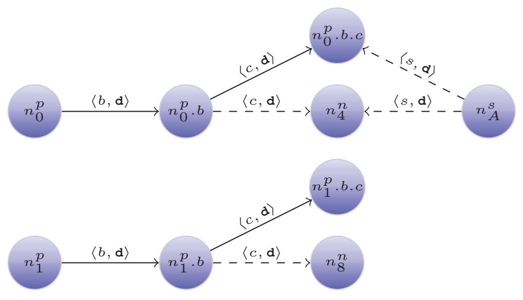

n_0^p
$n_0^p.b$
$n_0^p.b.c$
n_4^n
n_A^s
n_1^p
$n_1^p.b$
$n_1^p.b.c$
n_8^n
$\langle b, \mathbf{d}\rangle$
$\langle c, \mathbf{d}\rangle$
$\langle s, \mathbf{d}\rangle$

Figure A.1: Tricky aliasing example from the litterature

A.2 Tricky aliasing example from litterature

Aliasing is carrefully treated in our analysis. The AMG computed for each method is context-insensitive, hence different aliasing between parameters are available only when the method is invoked; in our analysis, when we compute the mapping relation. Constraint 3 in Definition 5.8, page 123, accounts for aliased nodes. We show how our constraint correctlly detects aliased nodes on an example from litterature.

Let us consider the example in Figure A.1, inspired from the master thesis of Salcianu [Sal01]. The example consists of two method, m_1 and m_2.

Method m_1 creates two new objects and sets the field b of first parameter, n_0^p, and of the new object created at line 0, n_0^n, to n_2^n; in the end, it calls method m_2 passing as arguments n_0^p and n_0^n, which are aliased as their field b points to the same object n_2^n. The AMG of method m_1 before invocation is depicted in Figure A.1c.

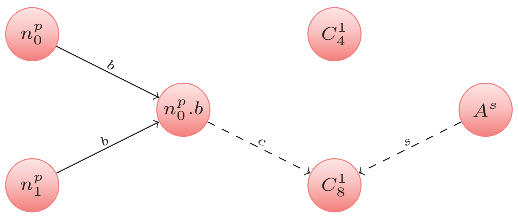


Figure A.2: Concrete memory graph at the end of m_2

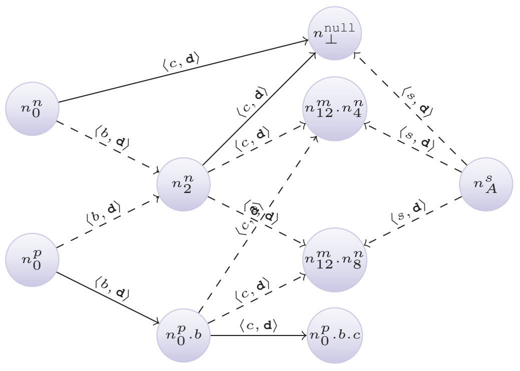

$n_\perp^{\mathtt{null}}$
$\langle c, \mathbf{d} \rangle$
n_0^n
$\langle c, \mathbf{d} \rangle$
$n_{12}^m.n_4^n$
$\langle s, \mathbf{d} \rangle$
$\langle b, \mathbf{d} \rangle$
$\langle c, \mathbf{d} \rangle$
$\langle s, \mathbf{d} \rangle$
n_2^n
n_A^s
$\langle b, \mathbf{d} \rangle$
$\langle c, \mathbf{d} \rangle$
$\langle s, \mathbf{d} \rangle$
n_0^p
$n_{12}^m.n_8^n$
$\langle b, \mathbf{d} \rangle$
$\langle c, \mathbf{d} \rangle$
$n_0^p.b$
$\langle c, \mathbf{d} \rangle$
$n_0^p.b.c$

Figure A.3: AMG at the end of m_1

The context-insensitive AMG for method m_2 is depicted in Figure A.1d. Note that the two parameters are treated as totally unaliased objects. The concrete execution of method m_2 leads to the memory graph in Figure A.2; the static node n_A^s points to n_8^n and not to n_4^n ! Hence, the AMG of m_1 after method invocation (Figure A.3) must also contain an edge from n_A^s to $n_{12}^m.n_8^n$.

To detect aliases, Constraint 3 supposes that two nodes related to the same node might be aliased nodes. In our example, it is obvious[1] that n_2^n is related to both $n_0^p.b$ and $n_1^p.b$:

$$\begin{array}{ccc} n_2^n & \overset{\mathbf{d}}{\sim}_\iota & n_0^p.b \\ n_2^n & \overset{\mathbf{d}}{\sim}_\iota & n_1^p.b. \end{array}$$

As a consequence, nodes $n_0^p.b$ and $n_1^p.b$ might be aliased, and all their field might be aliased as well. Contraint 3 will add the following relations[2]:

$$\begin{array}{ccc} n_{12}^m.n_8^n & \overset{\mathbf{d}}{\sim}_\iota & n_4^n \\ n_{12}^m.n_4^n & \overset{\mathbf{d}}{\sim}_\iota & n_8^n. \end{array}$$

Hence, all edges to n_4^n will be translated to edges to $n_{12}^m.n_8^n$ also. Consequently, our analysis detects the edge $(n_A^s, n_{12}^m.n_8^n, \langle s, \mathbf{d} \rangle)$.

A.3 Soundness of the intra-method analysis

A.3.1 Proof of Proposition 6.11 (Points-to correctness for single instruction block)

We prove the points-to correctness for single instruction block by case analysis on each instruction.

Except for the `new` bytecode, the other intructions do not add new nodes to the memory graph and do not change the location of objects. Thus, for a concrete state $\overline{Q} = ((pc, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ and $\overline{Q'} = \overline{instr_b}(\overline{Q}) = ((pc', \overline{\rho'}, \overline{s'}) :: fr, \overline{G'})$ and b a instruction which does not modify the graph, we have $\mathcal{O}(\overline{G}) = \mathcal{O}(\overline{G'})$.

Case: $b =$ **`prim op`**

From the operational semantics of `prim` op (Figure 6.4, page 139), $\overline{Q} = ((i, \overline{\rho}, \overline{v_1} :: \overline{v_2} :: \overline{s}) :: fr, \overline{G})$ and $\overline{instr_b}(\overline{Q}) = ((i+1, \overline{\rho}, op(\overline{v_1}, \overline{v_2}) :: \overline{s}) :: fr, \overline{G})$. From the abstract semantics rule of `prim`,

[1]From the initial mapping between argumets and parameters and from Constraint 2
[2]For simplification, we show a simplified version of the graphs, and concentrate only on relevant relations.

$Q = (\rho, v_1 :: v_2 :: s, G)$ and $instr_b(Q, \Gamma) = (\rho, v_1 \cup v_2 :: s, G)$. From hypothesis, we have $\overline{G} \lhd^\alpha G$, $\overline{\rho} \lhd^\alpha \rho$, $\overline{s} \lhd^\alpha s$ and since $\overline{v_1}, \overline{v_2} \notin \mathcal{O}(\overline{G})$, we also have $(op(\overline{v_1}, \overline{v_2}) :: \overline{s}) \lhd^\alpha (v_1 \cup v_2 :: s)$, and hence we can conclude with $\overline{instr_b}(\overline{Q}) \lhd^\alpha instr_b(Q, \Gamma)$.

Case: b = **pop**

From operational semantics, $\overline{Q} = ((i, \overline{\rho}, \overline{n} :: \overline{s}) :: fr, \overline{G})$ and $\overline{instr_b}(\overline{Q}) = ((i+1, \overline{\rho}, \overline{s}) :: fr, \overline{G})$. From the abstract rules, $Q = (\rho, v :: s, G)$ and

$instr_b(Q, \Gamma) = (\rho, s, G)$. As we have $\overline{Q} \lhd^\alpha Q$ from hypothesis, it is obvious that $\overline{instr_b}(\overline{Q}) \lhd^\alpha instr_b(Q, \Gamma)$.

Case: b = **ifeq** a

From operational semantics, $\overline{Q} = ((i, \overline{\rho}, \overline{n} :: \overline{s}) :: fr, \overline{G})$ and $\overline{instr_b}(\overline{Q}) = ((j, \overline{\rho}, \overline{s}) :: fr, \overline{G})$, where j is $i+1$ or a. From the abstract rules, $Q = (\rho, v_1 :: s, G)$ and $instr_b(Q, \Gamma) = (\rho, s, G)$. As we have $\overline{Q} \lhd^\alpha Q$ from hypothesis, it is obvious that $\overline{instr_b}(\overline{Q}) \lhd^\alpha instr_b(Q, \Gamma)$.

Case: b = **goto** a Let $\overline{Q} = ((i, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ and $\overline{instr_b}(\overline{Q}) = ((a, \overline{\rho}, \overline{s}) :: fr, \overline{G})$. The abstract semantics of `goto` instruction does not change the input state Q, and thus $instr_b(Q, \Gamma) = Q$. The concrete semantics rule makes a jump to address a and leaves unchanged the rest of the frame $\overline{Q}$. Hence the α-abstraction relation is preserved, and $\overline{instr_b}(\overline{Q}) \lhd^\alpha instr_b(Q, \Gamma)$.

Case: b = **bipush** n

The bytecode adds a primitive value on top of the stack, thus since primitive values are not concerned by α-abstraction, we have $\overline{instr_b}(\overline{Q}) \lhd^\alpha instr_b(Q, \Gamma)$.

Case: b = **aconst_null**

The bytecode pushes `null` on the concrete stack and $n_\perp^{\texttt{null}}$ on the abstract stack. By definition, $\alpha(\texttt{null}) = n_\perp^{\texttt{null}}$, which gives the abstraction relation between stacks.

Case: b = **new** C

The instruction pushes a new object on the stack. By construction, due to the allocation site model, the value pushed by the abstract rule (n_i^n) is the abstraction of the concrete object ($C_i^{\text{fresh}(C_i, \overline{G})}$) pushed on the concrete stack. The concrete graph G is enlarged with the new object, while the AMG already containts the abstract node (we remind that we suppose that all abstract values are computed from the begining).

Case: b = α**load x**

Let $\overline{Q} = ((i, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ and $Q = (\rho, s, G)$. From the operational semantics rule of α`load x` (Figure 6.4), we have $\overline{instr_b}(\overline{Q}) = ((i+1, \overline{\rho}, \overline{\rho}(x) :: \overline{s}) :: fr, \overline{G})$. From the abstract transformation rule of α`load x` (Figure 5.5), we have $instr_b(Q, \Gamma) = (\rho, \rho(x) :: s, G)$. By hypothesis, we have $\overline{Q} \lhd^\alpha Q$, thus by definition, we have $\overline{s} \lhd^\alpha s$, $\overline{\rho} \lhd^\alpha \rho$ and $\overline{G} \lhd^\alpha G$

The local variables array abstraction relation $\overline{\rho} \lhd^\alpha \rho$ implies that for all x such that $\overline{\rho}(x) \in \mathcal{O}(\overline{G})$, we have $(\alpha(\overline{\rho}(x)), \mathbf{d}) \in \rho(x)$. Since $\overline{s} \lhd^\alpha s$, the definition of stack abstraction gives us $(\overline{\rho}(x) :: \overline{s}) \lhd^\alpha (\rho(x) :: s)$. We can conclude with $\overline{instr_b}(\overline{Q}) \lhd^\alpha instr_b(Q, \Gamma)$.

Case: b = α**store** x

From the concrete semantics rule, if $\overline{Q} = ((i, \overline{\rho}, \overline{n} :: \overline{s}) :: fr, \overline{G})$, then $\overline{instr_b}(\overline{Q}) = ((i+1, \overline{\rho}[x \mapsto \overline{n}], \overline{s}) :: fr, \overline{G})$. From the abstract rules, if $Q = (\rho, u :: s, G)$, then $instr_b(Q, \Gamma) = (\rho[x \mapsto u \cup \{(e, \mathbf{i}) \mid e \in \Gamma\}], s, G)$. The α-abstraction on stacks and graphs is straightforward. For the local variables array, we distinguish two cases:

1. either $\overline{n} \in Val$, thus the location x in the local variables array contains a primitive value and the α abstraction does not apply to it,
2. or $\overline{n} \in \mathcal{O}(\overline{G})$; from hypothesis and the α-abstraction definition on stack, we have $\alpha(\overline{n}) \in u$, thus $(\overline{\rho}(x), \mathbf{d}) \in \rho(x)$, and, as the other elements of the local variables array do not change, $\overline{\rho} \lhd^\alpha \rho$.

Moreover, by hypothesis $\overline{s} \lhd^\alpha s$ and $\overline{G} \lhd^\alpha G$, thus the local soundness holds.

Case: b = **putfield** $f_{C'}$

Let $\overline{Q} = ((i, \overline{\rho}, \overline{v} :: \overline{u} :: \overline{s}) :: fr, \overline{G})$ and $Q = (\rho, v :: u :: s, G)$. We consider the case when $\overline{v} \notin Val$ (the case $\overline{v} \in Val$ is straightforward, as the edges between references do not change and neither does the stack nor the local variables array).

From the operational semantics rule of `putfield` (Figure 6.4), we have $\overline{instr_b}(\overline{Q}) = ((i+1, \overline{\rho}, \overline{s}) :: fr, \overline{G'})$. From the abstract transformation rule of `putfield` (Figure 5.5), we have $instr_b(Q, \Gamma) = (\rho, s, G')$. From the hypothesis, $\overline{s} \lhd^\alpha s$, $\overline{\rho} \lhd^\alpha \rho$.

We still need to prove that $\overline{G'} \lhd^\alpha G'$. Since $\overline{G} \lhd^\alpha G$ and because we only add and never delete edges to/from G, all we need to prove is that the edge added by $\overline{instr_b}$ respects the abstraction property i.e., the abstraction of the edge added by $\overline{instr_b}$ in $\overline{G}$ must be in G.

As, by definition, $\overline{G'} = \overline{G}[(\overline{u}, f_{C'}) \mapsto \overline{v}]$, the only edge added to $\overline{G'}$ is $(\overline{u}, \overline{v}, f_{C'})$. From $\overline{Q} \lhd^\alpha Q$ and the stack abstraction definition, we have $(\alpha(\overline{v}), \mathbf{d}) \in v$ and $(\alpha(\overline{u}), \mathbf{d}) \in u$.

The abstract transformation rule of `putfield` adds to G' edges having the form $(n', n, \langle f_{C'}, \mathbf{d}\rangle)$, $\forall n' \in u$ and $\forall n \in v$.

Thus, it also adds the edge $(\alpha(\overline{u}), \alpha(\overline{v}), \langle f_{C'}, \mathbf{d}\rangle)$. Thus $\overline{G'} \lhd^\alpha G'$.

Case: b = **getfield** $f_{C'}$

From the semantics rule of `getfield`, the local variables array and the memory graph do not change. Thus, we only need to prove the α-abstraction for the stack. Let $\overline{Q} = ((i, \overline{\rho}, \overline{n} :: \overline{s}) :: fr, \overline{G})$ and $Q = (\rho, u :: s, G)$. Let $\overline{n'} = adj_{\overline{G}}(\overline{n}, f_{C'})$; we distinguish two cases:

1. either $\overline{n'} \notin Obj$,
2. or $\overline{n'} \in Obj$.

In the first case, the α-abstraction does not apply to primitive values, thus we have the α-abstraction between stacks.

We now consider the second case, $\overline{n'} \in Obj$. By concrete and abstract semantics rules, $\overline{instr_b}(\overline{Q}) = ((i+1, \overline{\rho}, \overline{n'} :: \overline{s}) :: fr, \overline{G})$ and $instr_b(Q, \Gamma) = (\rho, u' :: s, G)$. By hypothesis, we have $\overline{s} \lhd^\alpha s$, hence we still need to show that the α-abstraction is preserved for the top of the resulted stack (between $\overline{n'}$ and u).

From Definition 6.7 of stack abstraction, we must prove that $(\alpha(\overline{n'}), \mathbf{d}) \in u$.

By hypothesis, we also have $\overline{G} \lhd^\alpha G$, which means that $\alpha(\overline{G}) \subseteq G$. Let $\overline{G} = (\overline{V}, \overline{E}, \varsigma)$, $G = (V, E)$ and $\alpha(\overline{G}) = (V^\alpha, E^\alpha)$.

By definition, $E^\alpha = \{(\alpha(\overline{v}), \alpha(\overline{v'}), \langle e, \mathbf{d}\rangle) \mid \overline{v}, \overline{v'} \in \mathcal{O}(\overline{G}) \wedge (\overline{v}, \overline{v}', e) \in \overline{E}\}$. As $E^\alpha \subseteq E$ and $(\overline{n}, \overline{n'}, f_{C'}) \in \overline{E}$, we have $(\alpha(\overline{n}), \alpha(\overline{n'}), \langle f_{C'}, \mathbf{d}\rangle) \in E$, and from the value of u' from the abstract transformation rule of `getfield`, we can conclude with $(\alpha(\overline{n'}), \mathbf{d}) \in u'$.

A.3.2 Proof of Proposition 6.13

Case: b = **prim op**

The instruction pops two operands from the stack and pushes back the result of *op* on these operands. The operands are primitive nodes. Thus the relation $\doteq$ holds for the resulted stack. The local variables array and the graph are not changed.

Case: $b = \texttt{pop}$

From the abstract semantics in Figure 5.5, $Q_1 = (\rho_1, v_1 :: s_1, G_1)$ and $instr_b(Q_1, \Gamma_1) = (\rho_1, s_1, G_1)$. As the only operation performed by this bytecode is a pop from the stack, obviously $instr_b(Q_1, \Gamma_1) \doteq instr_b(Q_2, \Gamma_2)$.

Case: $b = \texttt{ifeq}\ a$

From the abstract semantics in Figure 5.5, we have $Q_1 = (\rho_1, v :: s_1, G_1)$ and $instr_b(Q_1, \Gamma_1) = (\rho_1, s_1, G_1)$. As the only operations are two pops from the stack, obviously $instr_b(Q_1, \Gamma_1) \doteq instr_b(Q_2, \Gamma_2)$.

Case: $b = \texttt{goto}\ a$

The transformation rule does not change the input, thus $instr_b(Q_1, \Gamma_1) = Q_1$ and $instr_b(Q_2, \Gamma_2) = Q_2$. Since $Q_1 \doteq Q_2$, we can conclude straightforward with $instr_b(Q_1, \Gamma_1) \doteq instr_b(Q_2, \Gamma_2)$.

Case: $b = \texttt{bipush}\ iv$ **or** $b = \texttt{aconst\_null}$ **or** $b = \texttt{new\_}iC$

These instructions add an abstract element on the stack, either $({n^c}_i, \mathbf{d})$ or $(n_\perp^{\texttt{null}}, \mathbf{d})$ or $(n_i^n, \mathbf{d})$, which is always the same for Q_1 and Q_2. Thus, the $\doteq$ relation is preserved.

Case: $b = \alpha\texttt{load x}$

From hypothesis, $(\rho_1, s_1, G_1) \doteq (\rho_2, s_2, G_2)$. Thus, $\rho_1(x) \doteq \rho_2(x)$ and thus $(\rho_1, \rho_1(x) :: s_1, G_1) \doteq (\rho_2, \rho_2(x) :: s_2, G_2)$.

Case: $b = \alpha\texttt{store x}$

Let $Q_1 = (\rho_1, u_1 :: s_1, G_1)$, $Q_2 = (\rho_2, u_2 :: s_2, G_2)$, $instr_b(Q_1, \Gamma_1) = (\rho_1[x \mapsto u_1'], s_1, G_1)$, where $u_1' = u_1 \cup \{(t, \mathbf{i}) \mid t \in \Gamma_1\}]$, and $instr_b(Q_2, \Gamma_2) = (\rho_2[x \mapsto u_2'], s_2, G_2)$, where $u_2' = u_2 \cup \{(t, \mathbf{i}) \mid t \in \Gamma_2\}]$. To prove the relation $\doteq$, we must show that $u_1' \doteq u_2'$, which holds because by hypothesis we have $u_1 \doteq u_2$ and the restriction of set $\{(t, \mathbf{i}) \mid t \in \Gamma_1\}$ to objects is the empty set $\emptyset$. Hence, $instr_b(Q_1, \Gamma_1) \doteq instr_b(Q_2, \Gamma_2)$.

Case: $b = \texttt{getfield}\ f_{C'}$

Let $Q_1 = (\rho_1, u_1 :: s_1, G_1)$, $Q_2 = (\rho_2, u_2 :: s_2, G_2)$, $instr_b(Q_1, \Gamma_1) = (\rho_1, u_1' :: s_1, G_1)$ and $instr_b(Q_2, \Gamma_2) = (\rho_2, u_2' :: s_2, G_2)$. To prove the relation $\doteq$, we must show that $u_1' \doteq u_2'$, where $u_1' = \{(e, \mathbf{d}) \mid e \in adj_{G_1}(e', \langle f_{C'}, \mathbf{d}\rangle) \wedge (e', \mathbf{d}) \in u_1\} \cup \{(e, \mathbf{i}) \mid (e, \mathbf{i}) \in u_1\}$. The elements in $\{(e, \mathbf{i}) \mid (e, \mathbf{i}) \in u_1\}$ refer to primitive nodes, thus they do no present any interest here. Let us denote by u_1'' the first set in u_1'. We now show that $u_1'' \doteq u_2''$, where

$$u_1'' = \{(e, \mathbf{d}) \mid e \in adj_{G_1}(e_1', \langle f_{C'}, \mathbf{d}\rangle) \wedge (e_1', \mathbf{d}) \in u_1\}$$

$$u_2'' = \{(e, \mathbf{d}) \mid e \in adj_{G_2}(e_2', \langle f_{C'}, \mathbf{d}\rangle) \wedge (e_2', \mathbf{d}) \in u_2\}.$$

Since $G_1 \doteq G_2$ and $u_1 \doteq u_2$, all the adjacents labeled with $f_{C'}$ of e_1' (where $(e_1', \mathbf{d}) \in u_1$) are also adjacents of e_2' (where $(e_2', \mathbf{d}) \in u_2$). Thus $u_1'' \doteq u_2''$ and $u_1' \doteq u_2'$.

Case: $b = \texttt{putfield}\ f_{C'}$

Let $Q_1 = (\rho_1, v_1 :: u_1 :: s_1, (V_1, E_1))$, $Q_2 = (\rho_2, v_2 :: u_2 :: s_2, (V_2, E_2))$,
$instr_b(Q_1, \Gamma_1) = (\rho_1, s_1, (V_1, E_1'))$ and $instr_b(Q_2, \Gamma_2) = (\rho_2, s_2, (V_2, E_2'))$. To prove the relation $\doteq$, we must show that $(V_1, E_1') \doteq (V_2, E_2')$, thus the `putfield` abstract rule must add the same edges between reference nodes in E_1' and in E_2'.

From abstraction transformation rule in Figure 5.5, $E_1' = E_1 \cup E_1'' \cup E_1'''$, where $E_1'' = \{(e, e', \langle f_{C'}, t\rangle) | (e, \mathbf{d}) \in u, (e', t) \in v_1\}$ and $E_1''' = \{(e, e', \langle f_{C'}, \mathbf{i}\rangle) | (e, \mathbf{d}) \in u_1, e' \in \Gamma \cup V_{\mathbf{i}}^{u}\}$, $V_{\mathbf{i}}^{u_1} = \{e \mid \langle e, \mathbf{i}\rangle \in u\}$. The definition of E_2' is similar. Edges in E_1''' are between references and primitive nodes, thus are not considered by $\doteq$. We must refer only to E_1'' and E_2''. From hypothesis, $u_1 \doteq u_2$ and $v_1 \doteq v_2$, thus $E_1'' \doteq E_2''$. Thus, $(V_1, E_1') \doteq (V_2, E_2')$.

A.3.3 Proof of Proposition 6.18 (State variation correctness for single instruction block)

We prove the state variation correctness for single instruction block by case analysis on each instruction.

Case: b = **`prim op`**

The local variables array, the memory graph and the AMG are not affected by the execution of this bytecode, thus we still need to prove the state variation for the stack.

Let $\overline{Q_1} = ((i, \overline{\rho_1}, \overline{s_1}) :: fr, \overline{G_1})$, with $\overline{s_1} = \overline{n_1} :: \overline{n_2} :: \overline{s_{11}}$, $Q_1 = (\rho, n_1 :: n_2 :: s, G)$, $\overline{Q'_1} = ((i, \overline{\rho'_1}, \overline{s'_1}) :: fr, \overline{G'_1})$. From the operational semantics rule of `prim op` in Figure 6.4, we have $\overline{Q_2} = ((i+1, \overline{\rho_1}, \overline{s_2}) :: fr, \overline{G_1})$, with $\overline{s_2} = op(\overline{n_1}, \overline{n_2}) :: \overline{s_{11}}$ and, from the abstract transformation rule, $Q_2 = (\rho, (n_1 \cup n_2) :: s, G)$.

If there exists $\iota \in \beta$, $t \in \mathcal{F}$ such that $(\iota, t) \in n_1$, then the value on the top of the stack is randomized and $\overline{s'_1} = \overline{s_1}[0 \mapsto x]$ with $x \in Val$. Thus, $\overline{s'_2} = \overline{s_2}[0 \mapsto op(x, \overline{n_2})]$, which corresponds to Definition 6.16, as (ι, t) belongs to the top of the stack in Q_2 ($(\iota, t) \in n_1$ and obviously $(\iota, t) \in (n_1 \cup n_2)$). Thus $\overline{s'_2} \overset{s_1, \beta}{\longleftrightarrow} \overline{s'_1}$.

Case: b = **`pop`**

In both concrete and abstract semantics, the instruction pops a value from the stack. The rest of the stack, the local variables array, the memory graph and the AMG remain unchanged. Thus, if $\overline{Q'_1}$ is the state variation of $\overline{Q_1}$ with respect to Q_1 and β, $\overline{Q'_2} = \overline{instr_b}(\overline{Q'_1})$ and $Q_2 = instr_b(Q_1, \Gamma)$, then $\overline{Q'_2}$ is a state variation of $\overline{Q_2}$ with respect to Q_2 and β.

Case: b = **`goto`** a **or** b = **`ifeq`** a

The proof is similar with the one for `pop` bytecode, as the instruction does not change the local variables array, the memory graph and the AMG and moreover, does not change the stack.

Case: b = **`bipush`** n **or** b = **`new`**iC **or** b = **`aconst_null`**

A constant value is pushed on the stack (for both states $\overline{Q_1}$ and the state variation $\overline{Q'_1}$). The rest of the frame remains unchanged, thus the proof is straightforward.

Case: b = α**`load`** x

Let $Q_1 = (\rho, s, G)$, $Q_2 = (\rho, \rho(x) :: s, G)$ (from the abstract tranformation rules in Figure 5.5), $\overline{Q_1} = ((i, \overline{\rho}, \overline{s}) :: fr, \overline{G})$ and $\overline{Q_2} = ((i+1, \overline{\rho}, \overline{\rho}(x) :: \overline{s}) :: fr, \overline{G})$ (from the semantics rule in Figure 6.4).

Let $\overline{Q'_1} = ((i, \overline{\rho'}, \overline{s'}) :: fr, \overline{G'})$ be the state variation , and $\overline{Q'_2} = ((i+1, \overline{\rho'}, \overline{\rho'}(x) :: \overline{s'}) :: fr, \overline{G'})$.

The bytecode only pushes an element on the stack, thus the graph and local variables array do not change. From hypothesis, we have $\overline{G'} \overset{G, \beta}{\longleftrightarrow} \overline{G}$, thus the AMG in $\overline{Q'_2}$ is a state variation of the graph in $\overline{Q_2}$ with respect to the graph in Q_2 and β. The same reasonement applies to the local variables. We still need to prove the state variation for stacks. We know that $\overline{s'}$ is a state variation of $\overline{s}$ with respect to s and β. The top of the stack in $\overline{Q'_2}$ ($\overline{\rho'}(x)$) agrees with the local variables array variation, and thus it is a state variation of the top of the stack in $\overline{Q_2}$ ($\overline{\rho}(x)$) with respect to the top of the stack in Q_2 ($\rho_2(x)$) and β.

Case: b = α**`store`** x

The instruction stores the top of the stack in the local variable x. The proof is similar to the α`load` x bytecode: the top of the stack in the state variation $\overline{Q'_1}$ agrees with the stack variation definition, thus the local variable x in $\overline{Q'_2}$ agrees with the local variables array variation.

Case: $b =$ **putfield** $f_{C'}$

Let $\overline{Q_1} = ((i, \overline{\rho_1}, \overline{s_1}) :: fr, \overline{G_1})$, $\overline{Q_1'} = ((i, \overline{\rho_1'}, \overline{s_1'}) :: fr, \overline{G_1'})$, with $\overline{s_1} = \overline{v} :: \overline{u} :: \overline{s_2}$ and $\overline{s_1'} = \overline{v'} :: \overline{u'} :: \overline{s_2'}$. From the semantics rule of `putfield`, we have $\overline{Q_2} = ((i+1, \overline{\rho_1}, \overline{s_2}) :: fr, \overline{G_2})$ and $\overline{Q_2'} = ((i+1, \overline{\rho_1'}, \overline{s_2'}) :: fr, \overline{G_2'})$,

Since `putfield` does not affect the local variables and it only pops elements from the stack, the state variation is preserved for the local variables array and for the operand stack.

To completely prove the state variation correctness, we still need to prove the graph variation: $\overline{G_2'} \overset{G_2,\beta}{\longleftrightarrow} \overline{G_2}$.

Let $Q_1 = (\rho_1, s_1, G_1)$ with $s_1 = v :: u :: s_2$.

Depending on the type of the top of the stack $\overline{v}$, there are two semantics rules for the `putfield` instruction (as depicted in Figure 6.4). We consider each case: $\overline{v}$ is a primitive node or a reference node.

Let us first consider that the field manipulated is of primitive type ($\overline{v} \in \mathit{Val}$).

We will make a case study based on the values of $\overline{v}$ and $\overline{u}$, depending on β. We distinguish the following possibilities:

- for $\overline{v} \in \mathit{Val}$:
  - either $\exists \iota \in \beta, t \in \mathcal{F}$ such that $(\iota, t) \in v$
  - or $\forall \iota \in \beta, t \in \mathcal{F} (\iota, t) \notin v$.
- for $\overline{u} \notin \mathit{Val}$:
  - either $\exists \iota' \in \beta$ such that $(\iota', \mathbf{i}) \in u$
  - or $\forall \iota' \in \beta, (\iota', \mathbf{i}) \notin u$.

Based on the combination of cases presented above, we distinguish the following cases for the abstract stack, thus the following possibilities to compute the state $\overline{Q_1'}$.

Case I: $\forall \iota \in \beta, t \in \mathcal{F}, (\iota, t) \notin v$ **and** $\forall \iota' \in \beta, (\iota', \mathbf{i}) \notin u$

In this case, the rules of the state variation do not apply on $\overline{v}$ and $\overline{u}$, thus $\overline{v'} = \overline{v}$ and $\overline{u'} = \overline{u}$. Since `putfield` is executed on the same objects, the same edges are added and since $\overline{G_1'} \overset{G_1,\beta}{\longleftrightarrow} \overline{G_1}$, the conclusion is obvious $\overline{G_2'} \overset{G_2,\beta}{\longleftrightarrow} \overline{G_2}$.

Case II: $\exists \iota \in \beta, t \in \mathcal{F}$ **such that** $(\iota, t) \in v$ **and** $\forall \iota' \in \beta, (\iota', \mathbf{i}) \notin u$

We have a new value for $\overline{v'}$ and $\overline{u}$ is not affected by the state variation ($\overline{u} = \overline{u'}$). According to the concrete semantics, $\overline{G_2} = \overline{G_1}[\overline{n'} \mapsto \overline{v}]$ and $\overline{G_2'} = \overline{G_1'}[\overline{n'} \mapsto \overline{v'}]$, where $\overline{n'} = adj_{\overline{G}}(\overline{u}, f_{C'})$. Applying the abstract semantics of `putfield`, an edge $(\alpha(\overline{u}), \iota, \langle f_{C'}, t\rangle)$ is added to $\overline{G_2}$. Thus, $\overline{G_2'}$ is a graph variation of $\overline{G_2}$ with respect to G_2 and β.

Case III: $\exists \iota' \in \beta$ **such that** $(\iota', \mathbf{i}) \in u$

According to Definition 6.16 of stack variation, $\overline{u} \neq \overline{u'}$ and $\overline{u'} = \alpha^{-1}(o)$, with $(o, \mathbf{d}) \in u$. The transformation rule of `putfield` creates the links $(\alpha(\overline{u}), \iota, \langle f_{C'}, \mathbf{i}\rangle)$ and $(o, \iota, \langle f_{C'}, \mathbf{i}\rangle)$ in G_2. At the concrete level, the semantics will change the value of adjacent of $\alpha^{-1}(o)$ and not $\overline{u}$, which respects the graph variation definition.

We now consider the case when the field $f_{C'}$ is of type reference ($\overline{v} \in \mathit{Obj}$).

The idea is the same as the previous case, when $\overline{v} \in \mathit{Val}$: we will make a case study based on the values of $\overline{v}$ and $\overline{u}$, depending on β. We distinguish the following possibilities:

- for $\overline{v} \in \mathit{Obj}$:
  - either $\exists \iota \in \beta$ such that $(\iota, \mathbf{i}) \in v$
  - or $\forall \iota \in \beta, (\iota, \mathbf{i}) \notin v$.

- for $\overline{u} \in Obj$:
  - either $\exists \iota' \in \beta$ such that $(\iota', \mathbf{i}) \in u$
  - or $\forall \iota' \in \beta, (\iota', \mathbf{i}) \notin u$.

Case I: $\forall \iota \in \beta, (\iota, \mathbf{i}) \notin v$ **and** $\forall \iota' \in \beta, (\iota', \mathbf{i}) \notin u$

In this case, the rules of the state variation do not apply on $\overline{v}$ and $\overline{u}$, thus $\overline{v'} = \overline{v}$ and $\overline{u'} = \overline{u}$. Since putfield is executed on the same objects, the same edges are added and since $\overline{G'_1} \overset{G_1,\beta}{\longleftrightarrow} \overline{G_1}$, the conclusion is obvious $\overline{G'_2} \overset{G_2,\beta}{\longleftrightarrow} \overline{G_2}$.

Case II: $\exists \iota \in \beta$ **such that** $(\iota, \mathbf{i}) \in v$ **and** $\forall \iota' \in \beta, (\iota', \mathbf{i}) \notin u$

According to Definition 6.16 of stack variation, $\overline{v} \neq \overline{v'}$ and $\overline{v'} = \alpha^{-1}(o)$, with $(o, \mathbf{d}) \in v$. The transformation rule of putfield creates the links $(\alpha(\overline{u}), \iota, \langle f_{C'}, \mathbf{d}\rangle)$ and $(\alpha(\overline{u}), o, \langle f_{C'}, \mathbf{d}\rangle)$ in G_2. At the concrete level, the semantics rule will add the edge $(\overline{u}, \alpha^{-1}(o), f_{C'})$ in $\overline{G'_2}$ and not $(\overline{u}, \overline{v}, f_{C'})$, as in $\overline{G_2}$. The α-abstraction is still preserved for $\overline{G'_2}$ and G_2, as the $(\alpha(\overline{u}), o, \langle f_{C'}, \mathbf{d}\rangle) \in G_2$. Thus $\overline{G'_2} \lhd^{\alpha} G_2$ and $\overline{G'_2} \overset{G_2,\beta}{\longleftrightarrow} \overline{G_2}$.

Case III: $\exists \iota \in \beta$ **such that** $(\iota, \mathbf{i}) \in v$ **and** $\exists \iota' \in \beta$ **such that** $(\iota', \mathbf{i}) \in u$

According to stack variation in Definition 6.16, $\overline{u} \neq \overline{u'}$ and $\overline{u'} = \alpha^{-1}(o)$, with $(o, \mathbf{d}) \in u$. In the same way, $\overline{v} \neq \overline{v'}$ and $\overline{v'} = \alpha^{-1}(o')$, with $(o', \mathbf{d}) \in v$.

The execution of putfield creates the edges $(\alpha(\overline{u}), \iota, \langle f_{C'}, \mathbf{d}\rangle)$, $(o, \iota, \langle f_{C'}, \mathbf{d}\rangle)$, $(\alpha(\overline{u}), o', \langle f_{C'}, \mathbf{d}\rangle)$ and $(o, o', \langle f_{C'}, \mathbf{d}\rangle)$ in G_2. At the concrete level, the semantics will add the edge $(\alpha^{-1}(o), \alpha^{-1}(o'), f_{C'})$ in $\overline{G'_2}$ and not $(\overline{u}, \overline{v}, f_{C'})$, as in $\overline{G_2}$. The α-abstraction is still preserved for $\overline{G'_2}$ and G_2, as the $(o, o', \langle f_{C'}, \mathbf{d}\rangle) \in G_2$. Thus $\overline{G'_2} \lhd^{\alpha} G_2$, and $\overline{G'_2} \overset{G_2,\beta}{\longleftrightarrow} \overline{G_2}$.

Case IV: $\forall \iota \in \beta, (\iota, \mathbf{i}) \notin v$ **and** $\exists \iota' \in \beta$ **such that** $(\iota', \mathbf{i}) \in u$

This case is a simplification of the previous one, thus the proof is similar.

Case: $b =$ **getfield** $f_{C'}$

The instruction pops an object and pushes the field $f_{C'}$ of this object on the stack. Thus, to prove the state variation correctness, we must refer only to the new object/value pushed on the stack (the field $f_{C'}$) and prove that it can change only accordingly to stack variation rule of Definition 6.16.

Let $\overline{Q_1} = ((i, \overline{\rho_1}, \overline{u} :: \overline{s_1}) :: fr, \overline{G_1})$, $\overline{Q'_1} = ((i, \overline{\rho'_1}, \overline{u'} :: \overline{s'_1}) :: fr, \overline{G'_1})$. From the concrete semantics rule of getfield: $\overline{Q_2} = ((i+1, \overline{\rho_1}, \overline{v} :: \overline{s_1}) :: fr, \overline{G_1})$, $\overline{Q'_2} = ((i+1, \overline{\rho'_1}, \overline{v'} :: \overline{s'_1}) :: fr, \overline{G'_1})$. Moreover, let $Q_1 = (\rho, u :: s, G)$ and $Q_2 = (\rho, v :: s, G)$.

We can have two cases: 1) for all $\iota \in \beta$, $(\iota, \mathbf{i}) \notin u$ or 2) there exists $\iota \in \beta$ such that $(\iota, \mathbf{i}) \in u$.

1) Let us consider the first case: the top of the stack in $\overline{Q_1}$ does not change, thus $\overline{u'} = \overline{u}$. The getfield bytecode pushes the adjacent of $\overline{u}$ in $\overline{G_1}$ on the stack. Since $\overline{G'_1}$ is a state variation of $\overline{G_1}$ with respect to G_1 and β, G_1 is an α-abstraction of G'_1 ($\overline{G'_1} \lhd^{\alpha} G_1$). Again, we distinguish two cases:

- either $\beta \cap adj_{G_1}(\alpha(\overline{u}), f_{C'}) \neq \emptyset$
- or $\beta \cap adj_{G_1}(\alpha(\overline{u}), f_{C'}) = \emptyset$

In the second case, the adjacent labeled $f_{C'}$ of $\overline{u}$ does not change, thus the instruction pushes the same object/value on the stack for both $\overline{G'_1}$ and $\overline{G_1}$ ($\overline{v} = \overline{v'}$).

We now consider the first case ($\alpha(\overline{u})$ has an adjacent in β in G_1). Than there exists $\iota \in \beta$ and $t \in \mathcal{F}$ and an edge $(\alpha(\overline{u}), \iota, \langle f_{C'}, t\rangle)$ in G_1, We have two possibilities: $f_{C'}$ is of primitive type or is of type reference.

If the field $f_{C'}$ is of primitive type, the value of $adj_{\overline{G_1}}(\overline{u}, f_{C'})$ might change according with the graph variation definition (Definition 6.16). Thus, the value pushed on the stack in $\overline{G'_1}$ will differ from the one in $\overline{G_1}$, but it will respect the stack variation rule, since the abstract value (ι, t) will be pushed on the abstract stack in $\overline{Q_2}$ ($(\iota, t) \in v'$ since $(\alpha(\overline{u}), \iota, \langle f_{C'}, t\rangle) \in G_1 \subseteq G_2$).

If the field $f_{C'}$ is of type reference, the adjacent of u might change. Since $\overline{G_1'} \lhd^\alpha G_1$ and from the points-to correctness (Proposition 6.11), $\overline{instr_b}(\overline{Q_1'}) \lhd^\alpha instr_b(Q_1, \Gamma)$ or $\overline{Q_2'} \lhd^\alpha Q_2$. Thus $\overline{s_2'} \lhd^\alpha s_2$, $(\iota, \mathbf{i}) \in v'$ and $\alpha(v') \in v'$. Hence the stack respects the stack variation definition.

2) Let us now consider the case when there exists $\iota \in \beta$ such that $(\iota, \mathbf{i}) \in u$. Thus the stack variation rule of Definition 6.16 applies to $\overline{u}$, and $\overline{u'} = \alpha^{-1}(o)$, with $(o, \mathbf{i}) \in u$. From the abstract semantics of `getfield`, $(\iota, \mathbf{i})$ is kept on the stack ($(\iota, \mathbf{i}) \in v$). Thus, the value of $\overline{v'}$ will change, but it will respect the stack variation rules of state variation definition (the proof is similar with the previous case, depending on the type of $f_{C'}$, primitive or reference).

A.4 Soundness of inter-procedural analysis: intermediate layer

We now prove that the semantics rules for $\widehat{\texttt{invoke}}$ and $\alpha\widehat{\texttt{return}}$ in the intermediate model have the same properties as the rest of bytecode. We have to add the cases of the two new bytecodes ($\widehat{\texttt{invoke}}$ and $\alpha\widehat{\texttt{return}}$), to the proofs that have been previously done by case analysis on each bytecode.

A.4.1 Addition to proof of Lemma 5.5

We must show that the abstract semantics rules are monotone. We consider two pairs (Q_1, Γ_1) and (Q_2, Γ_2) from the property space $\mathcal{S}$ such that $(Q_1, \Gamma_1) \sqsubseteq (Q_2, \Gamma_2)$, and let $Q_1' = instr_b(Q_1, \Gamma_1)$ and $Q_2' = \widehat{instr}_b(Q_2, \Gamma_2)$. We prove that $Q_1' \sqsubseteq Q_2'$.

By case analysis on each instruction b, based on the transformation rules presented in Section 6.5.

Case: b = **invoke m**

Let $Q_1 = (\rho, u_{n_m} :: \cdots :: u_0 :: s, G)$ and $Q_2 = (\rho', u'_{n_m} :: \cdots :: u'_0 :: s', G')$. From the semantics of $\widehat{\texttt{invoke}}$,

$$\widehat{instr}_b(Q_1, \Gamma_1) = (\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G)$$
$$\widehat{instr}_b(Q_2, \Gamma_2) = (\{0 \mapsto u'_0, \ldots, n_m \mapsto u_{n'_m}\}, \epsilon, G').$$

As $Q_1 \sqsubseteq Q_2$ and $\Gamma_1 \subseteq \Gamma_2$, we have $u_0 \subseteq u'_0 \ldots u_{n_m} \subseteq u'_{n_m}$, $s \sqsubseteq s'$ and $G \subseteq G'$, thus $\widehat{instr}_b(Q_1, \Gamma_1) \sqsubseteq \widehat{instr}_b(Q_2, \Gamma_2)$.

Case: b = α**return**

Let $Q_1 = (\rho, u :: s, G)$ and $Q_2 = (\rho', u' :: s', G')$. From the semantics of $\alpha\widehat{\texttt{return}}$, $\widehat{instr}_b(Q_1, \Gamma_1) = (\zeta, u, G)$ and $\widehat{instr}_b(Q_2, \Gamma_2) = (\zeta, u', G')$.

As $Q_1 \sqsubseteq Q_2$ and $\Gamma_1 \subseteq \Gamma_2$, we have $u \subseteq u'$ and $G \subseteq G'$, thus $\widehat{instr}_b(Q_1, \Gamma_1) \sqsubseteq \widehat{instr}_b(Q_2, \Gamma_2)$.

A.4.2 Addition to proof of Proposition 6.11 (Points-to correctness)

In this section, we prove the local soundness of the intermediate model, that is the abstract semantics rules for $\widehat{\texttt{invoke}}$ and $\alpha\widehat{\texttt{return}}$ respect the local soundness, as defined in Proposition 6.11.

Case: b = **invoke m**

From the operational semantics of `invoke` m (Figure 6.4), we have a concrete state $\overline{Q}$

$$\overline{Q} = ((i, \overline{\rho}, \overline{p_{n_m}} :: \cdots :: \overline{p_1} :: \overline{p_0} :: \overline{s}) :: fr, \overline{G})$$

$$\overline{instr_b}(\overline{Q}) = ((0, \{0 \mapsto \overline{p'_0}, 1 \mapsto \overline{p_1} \ldots n_m \mapsto \overline{p_{n_m}}\}, \epsilon) :: (i, \overline{\rho}, \overline{s}) :: fr, \overline{G}).$$

From the abstract rule of $\widehat{\texttt{invoke}}$ m (defined in Section 6.5),

$$Q = (\rho, u_{n_m} :: \cdots :: u_0 : s, G)$$

$$\widehat{instr}_b(Q, \Gamma) = (\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G).$$

From hypothesis, we have $\overline{G} \lhd^\alpha G$, $\overline{p_0} \lhd^\alpha u_0 \ldots \overline{p_{n_m}} \lhd^\alpha u_{n_m}$. Moreover, the stacks are empty and $\lhd^\alpha$. Thus, $\overline{instr}_b(\overline{Q}) \lhd^\alpha \widehat{instr}_b(Q, \Gamma)$ according to Definition 6.20 .

Case: b = α**return** From the operational semantics of $\alpha\texttt{return}$ in Figure 6.4, we have

$$\overline{Q} = ((i, \overline{\rho}, \overline{v} :: \overline{s}) :: (i', \overline{\rho'}, \overline{s'}) :: fr, \overline{G})$$

$$\overline{instr}_b(\overline{Q}) = ((i' + 1, \overline{\rho'}, \overline{v} :: \overline{s'}) :: fr, \overline{G}).$$

From the abstract rule of $\alpha\widehat{\texttt{return}}$:

$$Q = (\rho, u :: s, G)$$

$$\widehat{instr}_b(Q, \Gamma) = (\zeta, u, G).$$

From hypothesis, $\overline{Q} \lhd^\alpha Q$, hence $\overline{G} \lhd^\alpha G$, $\overline{\rho} \lhd^\alpha \rho$, $\overline{v} \lhd^\alpha u$ and $\overline{s} \lhd^\alpha s$.

Since the local variables array in $\widehat{instr}_b(Q, \Gamma)$ equals to ζ, $\overline{instr}_b(\overline{Q}) \lhd^\alpha \widehat{instr}_b(Q, \Gamma)$ according to Definition 6.20 if $\overline{G} \lhd^\alpha G$ and $\overline{v} \lhd^\alpha u$ which holds by hypothesis.

A.4.3 Addition to proof of Proposition 6.13

We prove the second point of the Proposition for both bytecodes, $\widehat{\texttt{invoke}}$ and $\alpha\widehat{\texttt{return}}$: for any Γ_1, Γ_2, if $Q_1 \doteq Q_2$ then $\widehat{instr}_b(Q_1, \Gamma_1) \doteq \widehat{instr}_b(Q_2, \Gamma_2)$.

Case: b = **invoke m**

Let $Q_1 = (\rho, u_{n_m} :: \cdots :: u_0 :: s, G)$ and $Q_2 = (\rho', u'_{n_m} :: \cdots :: u'_0 :: s', G')$. From the semantics of $\widehat{\texttt{invoke}}$,

$$\widehat{instr}_b(Q_1, \Gamma_1) = (\{0 \mapsto u_0, \ldots, n_m \mapsto u_{n_m}\}, \epsilon, G)$$

$$\widehat{instr}_b(Q_2, \Gamma_2) = (\{0 \mapsto u'_0, \ldots, n_m \mapsto u_{n'_m}\}, \epsilon, G').$$

As $Q_1 \doteq Q_2$, we have $u_0 \doteq u'_0 \ldots u_{n_m} \doteq u'_{n_m}$ and $G \doteq G'$, thus $\widehat{instr}_b(Q_1, \Gamma_1) \doteq \widehat{instr}_b(Q_2, \Gamma_2)$.

Case: b = α**return**

Let $Q_1 = (\rho, u :: s, G)$ and $Q_2 = (\rho', u' :: s', G')$. From the semantics of $\alpha\widehat{\texttt{return}}$,

$$\widehat{instr}_b(Q_1, \Gamma_1) = (\zeta, u, G)$$

$$\widehat{instr}_b(Q_2, \Gamma_2) = (\zeta, u', G').$$

As $Q_1 \doteq Q_2$, we have $u \doteq u'$ and $G \doteq G'$, thus $\widehat{instr}_b(Q_1, \Gamma_1) \doteq \widehat{instr}_b(Q_2, \Gamma_2)$.

A.4.4 Addition to proof of Proposition 6.18 (State variation correctness)

Case: $b = \texttt{invoke m}$

From the operational semantics of `invoke` (Figure 6.4), we have

$$\overline{Q} = ((i, \overline{\rho}, \overline{p_{n_m}} :: \cdots :: \overline{p_1} :: \overline{p_0} :: \overline{s}) :: fr, \overline{G})$$

$$\overline{instr_b}(\overline{Q}) = ((0, \{0 \mapsto \overline{p_0}, 1 \mapsto \overline{p_1} \dots n_m \mapsto \overline{p_{n_m}}\}, \epsilon) :: (i, \overline{\rho}, \overline{s}) :: fr, \overline{G}).$$

From the abstract rule of $\widehat{\texttt{invoke}}$,

$$Q = (\rho, u_{n_m} :: \cdots :: u_0 : s, G)$$

$$\widehat{instr}_b(Q, \Gamma) = (\{0 \mapsto u_0, \dots, n_m \mapsto u_{n_m}\}, \epsilon, G).$$

Let $\overline{Q'}$ be the state variation and

$$\overline{Q'} \overset{Q,\beta}{\longleftrightarrow} \overline{Q} = ((i, \overline{\rho'}, \overline{p'_{n_m}} :: \cdots :: \overline{p'_1} :: \overline{p'_0} :: \overline{s'}) :: fr, \overline{G'})$$

$$\overline{instr_b}(\overline{Q'}) = ((0, \{0 \mapsto \overline{p'_0}, 1 \mapsto \overline{p'_1} \dots n_m \mapsto \overline{p'_{n_m}}\}, \epsilon) :: (i, \overline{\rho'}, \overline{s'}) :: fr, \overline{G'}).$$

From hypothesis, we have $\overline{G'} \overset{G,\beta}{\longleftrightarrow} \overline{G}$. The stacks are empty, thus the state variation relation holds. We need to prove the state variation for the local variables array. In $\overline{Q'}$, $\overline{p'_0}$ varies according to stack variation rule of Definition 6.16 in function of β and u_0 (the corresponding value on the abstract stack). In $\overline{instr_b}(\overline{Q'})$, the value of local variable 0 (which is $\overline{p'_0}$) is a variation of the local variable 0 in the abstract state $\widehat{instr}_b(Q, \Gamma)$ (which is u_0) and of β, thus it respects the local variables variation rule of Definition 6.16. We apply the same reasonment on the rest of the local variables, and we obtain the state variation for the local variables array.

Thus $\overline{instr_b}(\overline{Q'}) \overset{\widehat{instr}_b(Q,\Gamma),\beta}{\longleftrightarrow} \overline{instr_b}(\overline{Q})$.

Case: $b = \alpha\texttt{return}$

From the operational semantics of α`return` (Figure 6.4), we have

$$\overline{Q} = ((i, \overline{\rho}, \overline{v} :: \overline{s}) :: (i_0, \overline{\rho_0}, \overline{s_0}) :: fr, \overline{G})$$

$$\overline{instr_b}(\overline{Q}) = ((i_0 + 1, \overline{\rho_0}, \overline{v} :: \overline{s_0}) :: fr, \overline{G}).$$

From the abstract rule of $\alpha\widehat{\texttt{return}}$:

$$Q = (\rho, u :: s, G)$$

$$\widehat{instr}_b(Q, \Gamma) = (\zeta, u, G).$$

Let $\overline{Q'}$ be the state variation and

$$\overline{Q'} \overset{Q,\beta}{\longleftrightarrow} \overline{Q} = ((i, \overline{\rho'}, \overline{v'} :: \overline{s'}) :: (i_0, \overline{\rho_0}, \overline{s_0}) :: fr, \overline{G'})$$

$$\overline{instr_b}(\overline{Q'}) = ((i_0 + 1, \overline{\rho_0}, \overline{v'} :: \overline{s_0}) :: fr, \overline{G'}).$$

Since the local variables array in $\widehat{instr}_b(Q, \Gamma)$ equals to ζ, $\overline{instr_b}(\overline{Q}) \overset{\widehat{instr}_b(Q,\Gamma),\beta}{\longleftrightarrow} \overline{instr_b}(\overline{Q'})$ if $\overline{v'} \overset{u,\beta}{\longleftrightarrow} \overline{v}$ and $\overline{G} \overset{G,\beta}{\longleftrightarrow} \overline{G'}$. By hypothesis, $\overline{Q'} \overset{Q,\beta}{\longleftrightarrow} \overline{Q}$ and thus the two required relations hold.

A.5 Soundness of inter-procedural analysis: abstract layer

A.5.1 Proof of Lemma 6.30 (Monotonicity of the abstract transformation rules w.r.t. the $\preceq$ relation)

Proof. By case analysis, on each instruction type.

Let

$$\begin{array}{rcl} Q & = & (\rho, s, G = (V, E \cup {E^c}_1)) \\ Q' & = & (\rho', s', G' = (V', E' \cup {E^{c}}'_1)) \\ Q_1 & = & instr_b(Q, \Gamma) = (\rho_1, s_1, G_1 = (V, E \cup {E^c}_1)) \\ Q'_1 & = & instr_b(Q', \Gamma') = (\rho'_1, s'_1, G'_1 = (V', E' \cup {E^{c}}'_1)) \end{array}$$

From Definition 6.26, if $Q \preceq^{\sim_\iota}_{Q_0} Q'$ then $s \preceq^{\sim_\iota{}^{G,G'}} s'$, $\rho \preceq^{\sim_\iota{}^{G,G'}} \rho'$ and $G \preceq^{\sim_\iota}_{G_0} G'$.

Some semantics rules do not modify the AMG, hence $G = G_1$, $G' = G'_1$ and $\sim_\iota{}^{G,G'} = \sim_\iota{}^{G_1,G'_1}$.

Case: b = **prim op**

Let $Q = (\rho, v_1 :: v_2 :: s, G)$ and $Q' = (\rho', v'_1 :: v'_2 :: s', G')$. From the abstract semantics rule of `prim` *op* in Figure 5.5, $instr_b(Q, \Gamma) = (\rho, v_1 \cup v_2 \cup TV_{\Gamma'} :: s, G)$ and $instr_b(Q', \Gamma') = (\rho', v'_1 \cup v'_2 \cup TV_{\Gamma'} :: s', G')$, with $TV_A = \{(e, \mathtt{i}) \mid e \in A\}$.

As $G = G_1$, $G' = G'_1$ hence $\sim_\iota{}^{G,G'} = \sim_\iota{}^{G_1,G'_1}$. By hypothesis, $\rho \preceq^{\sim_\iota{}^{G,G'}} \rho'$, $s \preceq^{\sim_\iota{}^{G,G'}} s'$ and $G \preceq^{\sim_\iota}_{G_0} G'$. As $\sim_\iota{}^{G,G'} = \sim_\iota{}^{G_1,G'_1}$, we obtain $\rho \preceq^{\sim_\iota{}^{G_1,G'_1}} \rho'$, $s \preceq^{\sim_\iota{}^{G_1,G'_1}} s'$ and $G \preceq^{\sim_\iota}_{G_0} G'_1$.

We still need to prove that $(v_1 \cup v_2 \cup TV_\Gamma) \preceq^{\sim_\iota{}^{G_1,G'_1}} (v'_1 \cup v'_2 \cup TV_{\Gamma'})$.

As $v_1 \preceq^{\sim_\iota{}^{G,G'}} v'_1$ and $v_2 \preceq^{\sim_\iota{}^{G,G'}} v'_2$, according to Lemma 6.27, $(v_1 \cup v_2) \preceq^{\sim_\iota{}^{G,G'}} (v'_1 \cup v'_2)$.

The relation $\Gamma \preceq_{\mathtt{i}}^{\sim_\iota{}^{G,G'}} \Gamma'$ leads to $TV_\Gamma \preceq^{\sim_\iota{}^{G,G'}} TV_{\Gamma'}$. By applying again Lemma 6.27, we obtain $(v_1 \cup v_2 \cup TV_\Gamma) \preceq^{\sim_\iota{}^{G,G'}} (v'_1 \cup v'_2 \cup TV_{\Gamma'})$ (we recall that $\sim_\iota{}^{G,G'} = \sim_\iota{}^{G_1,G'_1}$). □

Case: b = **pop** or b = **ifeq a**

Let $Q = (\rho, v :: s, G)$ and $Q' = (\rho', v' :: s', G')$. The abstract semantics rule of `pop` pops the top of the stack, hence $instr_b(Q, \Gamma) = (\rho, s, G)$ and $instr_b(Q', \Gamma') = (\rho', s', G')$. The AMG is not modified, hence $\sim_\iota{}^{G,G'} = \sim_\iota{}^{G_1,G'_1}$ and $instr_b(Q, \Gamma) \preceq^{\sim_\iota}_{Q_0} instr_b(Q', \Gamma')$.

Case: b = **goto** This instruction does not modify the abstract state, hence the $\preceq$ relation is preserved.

Case: b = **bipush** iv or b = **aconst_null** or b = **new_iC**

Similar to case b = **prim op**.

Case: b = α**load x**

From hypothesis, $(\rho, s, G) \preceq^{\sim_\iota}_{Q_0} (\rho', s', G')$. Thus, $\rho(x) \preceq^{\sim_\iota{}^{G,G'}} \rho'(x)$; moreover $TV_\Gamma \preceq^{\sim_\iota{}^{G,G'}} TV_{\Gamma'}$, hence $(\rho, \rho(x) \cup TV_\Gamma :: s, G) \preceq^{\sim_\iota}_{Q_0} (\rho', \rho'(x) \cup TV_{\Gamma'} :: s', G')$.

Case: b = α**store x**

From hypothesis, $(\rho, u :: s, G) \preceq^{\sim_\iota}_{Q_0} (\rho', u' :: s', G')$. Thus, $u \preceq^{\sim_\iota{}^{G,G'}} u'$; moreover $TV_\Gamma \preceq^{\sim_\iota{}^{G,G'}} TV_{\Gamma'}$, hence $(\rho, \rho[x \mapsto u \cup TV_\Gamma], s, G) \preceq^{\sim_\iota}_{Q_0} (\rho'[x \mapsto u' \cup TV_\Gamma], s', G')$.

Case: b = **getfield** $f_{C'}$

Let $Q = (\rho, u :: s, G)$ and $Q' = (\rho', u' :: s', G')$. From the abstract semantics rule of `getfield` in Figure 5.5, $instr_b(Q, \Gamma) = (\rho, u_1 :: s, G)$ and $instr_b(Q', \Gamma') = (\rho', u'_1 :: s', G')$, with

$$u_1 = \{(e,t) \mid e \in adj_G(e_1, \langle f_{C'}, t\rangle) \wedge e_1 \in V^u_{\mathtt{d}}\} \cup \{\langle e, \mathtt{i}\rangle \mid e \in V^u_{\mathtt{i}}\} \cup TV_\Gamma$$

and $u'_1 = \{(e,t) \mid e \in adj_{G'}(e_2, \langle f_{C'}, t\rangle) \wedge e_2 \in V^{u'}_{\mathtt{d}}\} \cup \{\langle e, \mathtt{i}\rangle \mid e \in V^{u'}_{\mathtt{i}}\} \cup TV_{\Gamma'}$.
where $V^u_t = \{e \wedge (e,t) \in u\}$, $t \in \mathcal{F}$, $TV_A = \{(e,i) \wedge e \in A\}$.
Let us denote

$$v = \{(e,t) \mid e \in adj_G(e_1, \langle f_{C'}, t\rangle) \wedge e_1 \in V^u_{\mathtt{d}}\}$$
$$w = \{(e, \mathtt{i}) \mid e \in V^u_{\mathtt{i}}\}.$$

Hence, $u = v \cup w \cup TV_\Gamma$.
We define similary v' and w', and $u' = v' \cup w' \cup TV_{\Gamma'}$.

The bytecode does not modify the AMG, hence $\sim_\iota{}^{G,G'} = \sim_\iota{}^{G_1,G'_1}$.

By hypothesis: $\rho \preceq^{\sim_\iota{}^{G,G'}} \rho'$, $s \preceq^{\sim_\iota{}^{G,G'}} s'$, $G \preceq^{\sim_\iota}_{G_0} G'$, $\Gamma \preceq^{\sim_\iota{}^{G,G'}}_{\mathtt{i}} \Gamma'$ and $u \preceq^{\sim_\iota{}^{G,G'}} u'$.

Since $\Gamma \preceq^{\sim_\iota{}^{G,G'}}_{\mathtt{i}} \Gamma'$, then

$$TV_\Gamma \preceq^{\sim_\iota{}^{G,G'}} TV_{\Gamma'} \tag{A.51}$$

Let us now show that $v \preceq^{\sim_\iota{}^{G,G'}} v'$.
Let $e \in adj_G(e_1, \langle f_{C'}, \mathtt{d}\rangle)$ and $e_{11} \in adj_G(e_1, \langle f_{C'}, \mathtt{i}\rangle)$ (if it exists). Of course, $(e_1, e, \langle f_{C'}, \mathtt{d}\rangle) \in G$ and $(e_1, e_{11}, \langle f_{C'}, \mathtt{i}\rangle) \in G$.

As $u \preceq^{\sim_\iota{}^{G,G'}} u'$, then for all $(e_1, \mathtt{d}) \in u$ there exists $(e_2, \mathtt{d}) \in u'$ such that $e_1 \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e_2$. In other words, for all $e_1 \in V^u_{\mathtt{d}}$ there exists $e_2 \in V^{u'}_{\mathtt{d}}$ such that $e_1 \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e_2$.
Moreover, let $e' \in adj_{G'}(e_2, \langle f_{C'}, \mathtt{d}\rangle)$, and $(e_2, e'\langle f_{C'}, \mathtt{d}\rangle) \in G'$. In other words, $(e', \mathtt{d}) \in v'$.

From $e_1 \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e_2$, $(e_1, e, \langle f_{C'}, \mathtt{d}\rangle) \in G$ $(e_2, e'\langle f_{C'}, \mathtt{d}\rangle) \in G'$ and Constraint 2 of Definition 5.8, we can deduce $e \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e'$.

From $e_1 \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e_2$, $(e_1, e, \langle f_{C'}, \mathtt{d}\rangle) \in G$, $(e_1, e_{11}, \langle f_{C'}, \mathtt{i}\rangle) \in G$ and Constraint 4 of Definition 5.8, we can deduce $e_{11} \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e'$.
Hence,

- for all $(e, \mathtt{d}) \in v$, there exists $(e', \mathtt{d}) \in v'$ such that $e \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e'$,
- for all $(e_{11}, \mathtt{i}) \in v$, there exist $(e', \mathtt{d}) \in v'$, $(e, \mathtt{d}) \in v$ such that $e \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e'$ and $e_{11} \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e'$.

We can conclude with

$$v \preceq^{\sim_\iota{}^{G,G'}} v' \tag{A.52}$$

Let us now show that $w \preceq^{\sim_\iota{}^{G,G'}} v' \cup w'$.
We remind that $w = \{(e, \mathtt{i}) \mid e \in V^u_{\mathtt{i}}\}$, where $V^u_{\mathtt{i}} = \{e \mid (e, \mathtt{i}) \in u\}$.

By hypothesis, $u \preceq^{\sim_\iota{}^{G,G'}} u'$; hence, for all $(e, \mathtt{i}) \in u$ (or for all $(e, \mathtt{i}) \in w$)

- either there exists $(e', \mathtt{i}) \in u'$ such that $e \overset{\mathtt{i}}{\sim}_\iota^{G,G'} e'$; in other words, there exists $(e', \mathtt{i}) \in w'$ such that $e \overset{\mathtt{i}}{\sim}_\iota^{G,G'} e'$,
- or there exist $(e', \mathtt{d}) \in u'$, $(e_1, \mathtt{d}) \in u$ such that $e \overset{\mathtt{i}}{\sim}_\iota^{G,G'} e'$, $e_1 \overset{\mathtt{d}}{\sim}_\iota^{G,G'} e'$. For $(e_1, \mathtt{d}) \in u$, $e_1 \overset{\mathtt{d}}{\sim}_\iota^{G,G'} e'$, let $(e_1, e'_1, \langle f_{C'}, \mathtt{d}\rangle) \in G$, $(e', e'_2, \langle f_{C'}, \mathtt{d}\rangle) \in G'$. By applying Constraint 2 of Definition 5.8, we obtain $e'_1 \overset{\mathtt{d}}{\sim}_\iota^{G,G'} e'_2$. Moreover, from $e \overset{\mathtt{i}}{\sim}_\iota^{G,G'} e'$ and from Constraint 5, we obtain $e \overset{\mathtt{i}}{\sim}_\iota^{G,G'} e'_2$.

We can hence conclude with

$$w \preceq^{\sim_\iota{}^{G,G'}} v' \cup w' \tag{A.53}$$

From (A.51), (A.52), (A.53) and by applying Lemma 6.27 we obtain

$$v \cup w \cup TV_\Gamma \preceq^{\sim_\iota{}^{G,G'}} v' \cup w' \cup TV_{\Gamma'}$$

$\square$

Case: $b =$ **putfield** $f_{C'}$

Let $Q = (\rho, v :: u :: s, G = (V, E \cup E^c))$ and $Q' = (\rho', v' :: u' :: s', G' = (V', E' \cup E^{c'}))$. From the abstract semantics rule of `putfield` in Figure 5.5,

$$instr_b(Q, \Gamma) = (\rho, s, G_1 = (V, E \cup (E^c \cup E_1)))$$

$$instr_b(Q', \Gamma') = (\rho', s', G' = (V', E' \cup (E^{c'} \cup E'_1)))$$

with

$$\begin{array}{rl} E_1 = & \{(e, e', \langle f_{C'}, t\rangle) \mid (e, \mathtt{d}) \in u, (e, t) \in v, e \neq n_\perp^{\mathtt{null}}\} \cup \\ & \{(e, e', \langle f_{C'}, \mathtt{i}\rangle) \mid (e, \mathtt{d}) \in u, e \neq n_\perp^{\mathtt{null}} e' \in \Gamma \vee (e', \mathtt{i}) \in u\}. \end{array}$$

E'_1 is defined similary for u', v' and Γ'.

As $G \subseteq G_1$, $G' \subseteq G'_1$, by applying Lemma 6.28 we obtain $\sim_\iota{}^{G,G'} \subseteq \sim_\iota{}^{G_1,G'_1}$.

By hypothesis, $Q \preceq_{Q_0}^{\sim_\iota{}^{G,G'}} Q'$, hence $\rho \preceq^{\sim_\iota{}^{G,G'}} \rho'$, $s \preceq^{\sim_\iota{}^{G,G'}} s'$, $v \preceq^{\sim_\iota{}^{G,G'}} v'$, $u \preceq^{\sim_\iota{}^{G,G'}} u'$ and $G \preceq_{G_0}^{\sim_\iota} G'$. As $\sim_\iota{}^{G,G'} \subseteq \sim_\iota{}^{G_1,G'_1}$, then $\rho \preceq^{\sim_\iota{}^{G_1,G'_1}} \rho'$, $s \preceq^{\sim_\iota{}^{G_1,G'_1}} s'$.

We still need to prove that $G_1 \preceq_{G_0}^{\sim_\iota} G'_1$, which corresponds to $G_1 \subseteq G_0 \oplus_{\sim_\iota} G'_1$, or

$$\begin{array}{rcl} (V, E \cup (E^c \cup E_1)) & \subseteq & (V, E \cup E^c{}_0) \oplus_{\sim_\iota} (V', E' \cup (E^{c'} \cup E'_1)) \\ & = & (V, E \cup (E^c{}_0 \cup E^{c'}{}_1)). \end{array}$$

As $G \preceq_{G_0}^{\sim_\iota} G'$, then $G \subseteq G_0 \oplus_{\sim_\iota} G'$ or

$$\begin{array}{rcl} (V, E \cup E^c) & \subseteq & (V, E \cup E^c{}_0) \oplus_{\sim_\iota} (V', E' \cup E^{c'}) \\ & = & (V, E \cup (E^c{}_0 \cup E^{c''})). \end{array}$$

The idea is to show that $E^c \cup E_1 \subseteq E^c{}_0 \cup E^{c'}{}_1$, or for all $(x, y, \langle f, t\rangle) \in E^c \cup E_1$, we also have $(x, y, \langle f, t\rangle) \in E^c{}_0 \cup E^{c'}{}_1$.

There are two cases: (1) either $(x, y, \langle f, t\rangle) \in E^c$ or (2) $(x, y, \langle f, t\rangle) \in E_1$. In the first case, as $G \subseteq G_0 \oplus_{\sim_\iota} G'$, then $E_c \subseteq E^c{}_0 \cup E^{c''}$; hence $(x, y, \langle f, t\rangle) \in E^c{}_0 \cup E^{c''}$. From $G' \subseteq G'_1$ and from Lemma 6.29 we obtain $G_0 \oplus_{\sim_\iota} G' \subseteq G_0 \oplus_{\sim_\iota} G'_1$, hence $E^c{}_0 \cup E^{c''} \subseteq E^c{}_0 \cup E^{c'}{}_1$.

We now consider the second case, $(x, y, \langle f, t\rangle) \in E_1$. We have three cases:

1. $(x, y, \langle f, t\rangle) \in \{(e, e'\langle f_{C'}, t\rangle) \mid (e, \mathtt{d}) \in u, (e', t) \in v, e \neq n_{\perp}^{\mathtt{null}}\}$
2. $(x, y, \langle f, \mathtt{i}\rangle) \in \{(e, e'\langle f_{C'}, \mathtt{i}\rangle) \mid (e, \mathtt{d}) \in u, e' \in \Gamma, e \neq n_{\perp}^{\mathtt{null}}\}$
3. $(x, y, \langle f, \mathtt{i}\rangle) \in \{(e, e'\langle f_{C'}, \mathtt{i}\rangle) \mid (e, \mathtt{d}) \in u, (e', \mathtt{i}) \in u, e \neq n_{\perp}^{\mathtt{null}}\}$

As $u \preceq^{\sim_\iota{}^{G,G'}} u'$, then:

- forall $(e_u, \mathtt{d}) \in u$ there exists $(e_{u'}, \mathtt{d}) \in u'$ such that $e_u \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e_{u'}$,
- forall $(e_v, \mathtt{d}) \in v$ there exists $(e_{v'}, \mathtt{d}) \in v'$ such that $e_v \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e_{v'}$,
- forall $(e'_u, \mathtt{i}) \in u$ then
  - either there exists $(e'_{u'}, \mathtt{i}) \in u'$ such that $e'_u \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e'_{u'}$
  - or there exist $(e''_{u'}, \mathtt{d}) \in u'$, $(e''_u, \mathtt{d}) \in u$ such that $e'_u \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e''_{u'}$ and $e''_u \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e''_{u'}$
- forall $(e'_v, \mathtt{i}) \in v$ then
  - either there exists $(e'_{v'}, \mathtt{i}) \in v'$ such that $e'_v \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e'_{v'}$
  - or there exist $(e''_{v'}, \mathtt{d}) \in v'$, $(e''_v, \mathtt{d}) \in v$ such that $e'_v \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e''_{v'}$ and $e''_v \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e''_{v'}$

We now consider each of the three cases listed above:

1. $(x, y, \langle f, t\rangle) \in \{(e, e'\langle f_{C'}, t\rangle) \mid (e, \mathtt{d}) \in u, (e', t) \in v, e \neq n_{\perp}^{\mathtt{null}}\}$
   For $(e_u, \mathtt{d}) \in u$, there are three posibilities:
   - $t = \mathtt{d}$, hence $(e_v, \mathtt{d}) \in v$ and $(e_u, e_v, \langle f_{C'}, \mathtt{d}\rangle) \in E_1$. From the `putfield` abstract semantics rule, $(e_{u'}, e_{v'}, \langle f_{C'}, \mathtt{d}\rangle) \in E'_1$. From the Rule 2a of Definition 5.9, the edge $(e_u, e_v, \langle f_{C'}, \mathtt{d}\rangle)$ is added by the $\oplus_{\sim_\iota}$ operation, hence $(e_u, e_v, \langle f_{C'}, \mathtt{d}\rangle) \in E^{c}{}'_1$,
   - $t = \mathtt{i}$, $(e'_v, \mathtt{i}) \in v$, and there exists $(e'_{v'}, \mathtt{i}) \in v'$ such that $e'_v \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e'_{v'}$. From the `putfield` abstract semantics rule, $(e_{u'}, e'_{v'}, \langle f_{C'}, \mathtt{i}\rangle) \in E'_1$. From the Rule 2a of Definition 5.9, the edge $(e_u, e'_v, \langle f_{C'}, \mathtt{i})$ is added by the $\oplus_{\sim_\iota}$ operation, hence $(e_u, e'_v, \langle f_{C'}, \mathtt{i}\rangle) \in E^{c}{}'_1$,
   - $t = \mathtt{i}$, $(e'_v, \mathtt{i}) \in v$, and there exist $(e''_{v'}, \mathtt{d}) \in v'$, $(e''_v, \mathtt{d}) \in v$ such that $e'_v \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e''_{v'}$ and $e''_v \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e''_{v'}$ From the `putfield` abstract semantics rule, $(e_{u'}, e''_{v'}, \langle f_{C'}, \mathtt{d}\rangle) \in E'_1$. From the Rule 2a of Definition 5.9, the edge $(e_u, e''_v, \langle f_{C'}, \mathtt{d}\rangle)$ is added by the $\oplus_{\sim_\iota}$ operation, hence, from Rule 2b the edge $(e_u, e'_v, \langle f_{C'}, \mathtt{i}\rangle)$ is also added.
2. $(x, y, \langle f, \mathtt{i}\rangle) \in \{(e, e'\langle f_{C'}, \mathtt{i}\rangle) \mid (e, \mathtt{d}) \in u, e' \in \Gamma, e \neq n_{\perp}^{\mathtt{null}}\}$
   As $\Gamma \preceq_{\mathtt{i}}^{\sim_\iota{}^{G,G'}} \Gamma'$, then for all $e \in \Gamma$ there exists $e' \in \Gamma'$ such that $e \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e'$. From `putfield` semantics, $(e_{u'}, e', \langle f_{C'}, \mathtt{i}\rangle) \in E'_1$. By applying Rule 2a, we add the edge $(e_u, e, \langle f_{C'}, \mathtt{i}, \rangle)$ to $E^{c}{}'_1$,
3. $(x, y, \langle f, \mathtt{i}\rangle) \in \{(e, e'\langle f_{C'}, \mathtt{i}\rangle) \mid (e, \mathtt{d}) \in u, (e', \mathtt{i}) \in u, e \neq n_{\perp}^{\mathtt{null}}\}$
   For $(e'_u, \mathtt{i}) \in u$ there are two cases:
   - either there exists $(e'_{u'}, \mathtt{i}) \in u'$ such that $e'_u \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e'_{u'}$. From `putfield` semantics, the edge $(e_{u'}, e'_{u'}, \langle f_{C'}, \mathtt{i}\rangle) \in E'_1$. By applying Rule 2a of the composition, the edge $(e_u, e'_u, \langle f_{C'}, \mathtt{i}\rangle)$ is added to $E^{c}{}'_1$,
   - or there exist $(e''_{u'}, \mathtt{d}) \in u'$, $(e''_u, \mathtt{d}) \in u$ such that $e'_u \overset{\mathtt{i}}{\sim}_\iota{}^{G,G'} e''_{u'}$ and $e''_u \overset{\mathtt{d}}{\sim}_\iota{}^{G,G'} e''_{u'}$. From `putfield` semantics, $(e_{u'}, e''_{u'}, \langle f_{C'}, \mathtt{d}\rangle) \in E'_1$; from Rule 2a, the edge $(e_u, e'_u, \langle f_{C'}, \mathtt{i}\rangle \in E^{c}{}'_1$.

□

Personal bibliography

▷▷ International conferences

[1] D. Ghindici, G. Grimaud et I. Simplot-Ryl. Embedding Verifiable Information Flow Analysis. In Proc. of the 4th Annual International Conference on Privacy, Security and Trust (PST 2006), McGraw-Hill, pages 343-352, Toronto, Canada, November 2006.

[2] D. Ghindici, G. Grimaud, I. Simplot-Ryl, I. Traore et Y. Liu. Integrated Security Verification and Validation: Case Study. In Proc. of the Second IEEE LCN Workshop on Network Security (WNS 2006), held in conjunction with the 31st Annual IEEE Conference on Local Computer Networks (LCN 2006), Tampa, Florida, November 2006.

[3] D. Ghindici, G. Grimaud et I. Simplot-Ryl. An information flow verifier for small embedded systems. In Proc. of Workshop in Information Security Theory and Practices (WISTP 2007) - Smart Cards, Mobile and Ubiquitous Computing Systems, LNCS 4462, pages 189-201, Heraklion, Crete, Greece, May 2007.

[4] N. Bel Hadj Aissa, D. Ghindici, G. Grimaud et I. Simplot-Ryl. Contracts as a support to static analysis of open systems. In Proc. The First Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS 2007), Oslo, Norway, October 2007.

[5] D. Ghindici et I. Simplot-Ryl. On Practical Information Flow Policies for Java-Enabled Multiapplication Smart Cards. In Proc. of the Eighth Smart Card Research and Advanced Application IFIP Conference (CARDIS 2008), LNCS 5189, pages 32-47, Egham, Surrey, UK, September 2008.

[6] D. Ghindici, I. Simplot-Ryl et J.M. Talbot. A sound analysis for secure information flow using abstract memory graphs. In Proc. 3rd International Conference on Fundamentals of Software Engineering (FSEN 2009), LNCS to appear, Kish Island, Persian Gulf, Iran, April 2009. Springer.

▷▷ Technical reports

[7] D. Ghindici, G. Grimaud et I. Simplot-Ryl. Embedding Verifiable Information Flow Analysis. Rapport technique 2006-03, LIFL UMR 8022 CNRS - USTL, April 2006.

[8] D. Ghindici, I. Simplot-Ryl et J-M. Talbot. A sound dependency analysis for secure information flow (extended version). Technical Report 0347, INRIA, November 2007.

Bibliography

[ABB06] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. *ACM SIGPLAN Notices*, 41(1):91–102, 2006.

[ABF03] M. Avvenuti, C. Bernardeschi, and N. D. Francesco. Java bytecode verification for secure information flow. *ACM SIGPLAN Notices*, 38(12):20–27, 2003.

[ACL03] J. Andronick, B. Chetali, and O. Ly. Using Coq to Verify Java Card Applet Isolation Properties. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs03)*, volume 2758 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, 2003.

[AGGSR07] N. B. H. Aissa, D. Ghindici, G. Grimaud, and I. Simplot-Ryl. Contracts as a support to static analysis of open systems. In *Proceedings The First Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS07)*, Oslo, Norway, 2007.

[And94] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[And96] R. J. Anderson. A security policy model for clinical information systems. In *Proceedings of the IEEE Symposium on Security and Privacy (SP96)*, page 30, Washington, DC, USA, 1996. IEEE Computer Society.

[And06] J. Andronick. *Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur – Plate-Forme Java Card et Système d'Exploitation*. PhD thesis, Université Paris-Sud, 2006.

[Aon] Aonix Inc. Perc products.

[Bal93] T. Ball. What's in a region?: or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems*, 2(1-4):1–16, 1993.

[BB08] F. Bavera and E. Bonelli. Type-based information flow analysis for bytecode languages with variable object field policies. In *Proceedings of the 2008 ACM symposium on Applied computing (SAC08)*, pages 347–351, New York, NY, USA, 2008. ACM.

[BBR04] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation: (extended abstract). In *Proceedings 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 2–15, Venice, Italy, 2004. Springer.

[BCG$^+$02] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets: extended version. *Journal of Computer Security*, 10(4):369–398, 2002.

[Ber01] Y. Bertot. Formalizing a jvml verifier for initialization in a theorem prover. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV01)*, pages 14–24, London, UK, 2001. Springer-Verlag.

[BF02] C. Bernardeschi and N. D. Francesco. Combining abstract interpretation and model checking for analysing security properties of java bytecode. In *Proceedings of the Third International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI02)*, pages 1–15, London, UK, 2002. Springer.

[BFLM04] C. Bernardeschi, N. D. Francesco, G. Lettieri, and L. Martini. Checking secure information flow in java bytecode by code transformation and standard bytecode verification. *Software – Practice and Experience*, 34(13):1225–1255, 2004.

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation, Bedford, MA, 1977.

[BL76] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997 Rev. 1, The MITRE Corporation, Bedford, MA, 1976.

[BN02] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the 15th IEEE Workshop on Computer Security Foundations (CSFW02)*, page 253, Washington, DC, USA, 2002. IEEE Computer Society.

[BN05] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, 2005.

[BNS04] M. Barnett, W. Naumann, and Q. Sun. 99.44% pure: useful abstractions in specifications. In *Proceedings of the ECOOP'2004 workshop: Formal techniques for Java-like programs*, 2004.

[BNSS06] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. Allowing state changes in specifications. In *Proceedings of International Conference on Emerging Trends in Information and Communication Security (ETRICS06)*, volume 3995 of *Lecture Notes in Computer Science*, pages 321–336, Freiburg, Germany, 2006. Springer.

[BPR07] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In *Proceedings 16th European Symposium on Programming (ESOP07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 125–140, Braga, Portugal, 2007. Springer.

[BR05] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *Proc. 2005 ACM SIGPLAN Intl. workshop on Types in Languages Design and Implementation (TLDI05)*, pages 103–112, New York, NY, USA, 2005. ACM.

[BRN06] G. Barthe, T. Rezk, and D. Naumann. Deriving an information flow checker and certifying compiler for java. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (SP06)*, pages 230–242, Washington, DC, USA, 2006. IEEE Computer Society.

[BSF04] M. Q. Beers, C. Stork, and M. Franz. Efficiently verifiable escape analysis. In *Proceedings of 18th European Conference Object-Oriented Programming (ECOOP 2004), Oslo, Norway, June 14-18, 2004*, volume 3086 of *Lecture Notes in Computer Science*, pages 75–95. Springer, 2004.

[CBC93] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *In Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL93)*, pages 232–245, New York, NY, USA, 1993. ACM Press.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL77)*, pages 238–252, Los Angeles, California, 1977. ACM Press.

[CC08] S. Cavadini and D. Cheda. Run-time information flow monitoring based on dynamic dependence graphs. *Third International Conference on Availability, Reliability and Security (ARES08)*, 0:586–591, 2008.

[CF07] D. Chandra and M. Franz. Fine-Grained Information Flow Analysis and Enforcement in a Java Virtual Machine. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC07)*, 2007.

[CFH$^+$98] A. Carzaniga, A. Fuggetta, R. S. Hall, A. van der Hoek, D. Heimbigner, and A. L. Wolf. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, 1998.

[CGS$^+$03] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Transactions on Programming Languages and Systems*, 25(6):876–910, 2003.

[CPS$^+$00] C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano, and H. Srinivasan. Dependence analysis for Java. In *Proceedings 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC99)*, pages 35–52, London, UK, 2000. Springer.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[Den76] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[Den82] D. E. R. Denning. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

[DGGJ03] D. Deville, A. Galland, G. Grimaud, and S. Jean. Smart card operating systems: Past, present, and future. In *Proceedings of the 5th USENIX/NordU Conference*, Västerås, Sweden, 2003.

[DPS03] S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Access control: Principles and solutions. *Software – Practice and Experience*, 33(5), 2003.

[EBM05] A. C. Eduardo Bonelli and R. Medel. Information-flow analysis for a typed assembly language with polymorphic stacks. In *Proceedings of 2nd Intl. Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS05)*, volume 3956 of *Lecture Notes in Computer Science*. Springer, 2005.

[Fen74] J. S. Fenton. Memoryless subsystems. *Computing Journal*, 17(2):143–147, 1974.

[GGSR06a] D. Ghindici, G. Grimaud, and I. Simplot-Ryl. Embedding verifiable information flow analysis. In *Proceedings Conference on Privacy, Security and Trust (PST06)*, pages 343–352, Toronto, Canada, 2006. McGraw-Hill.

[GGSR$^+$06b] D. Ghindici, G. Grimaud, I. Simplot-Ryl, I. Traore, and Y. Liu. Integrated security verification and validation: Case study. In *Proceedings of the Second IEEE LCN Workshop on Network Security (WNS06), held in conjunction with the 31st Annual IEEE Conference on Local Computer Networks (LCN06)*, page to appear, Tampa, Florida, 2006.

[GGSR07] D. Ghindici, G. Grimaud, and I. Simplot-Ryl. An information flow verifier for small embedded systems. In *Proceedings Workshop in Information Security Theory and Practices (WISTP07) Smart Cards, Mobile and Ubiquitous Computing Systems*, volume 4462 of *Lecture Notes in Computer Science*, pages 189–201, Heraklion, Crete, Greece, 2007. Springer.

[GHSR06] G. Grimaud, Y. Hodique, and I. Simplot-Ryl. Can small and open embedded systems benefit from escape analysis? In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS06)*, 2006.

[GHSR07] G. Grimaud, Y. Hodique, and I. Simplot-Ryl. A verifiable lightweight escape analysis supporting creational design patterns. In *Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW07)*, pages 440–447, Washington, DC, USA, 2007. IEEE Computer Society.

[Gir99] P. Girard. Which security policy for multiapplication smart cards? In *USENIX Workshop on Smartcard Technology*, pages 21–28, 1999.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GM96] E. F. Gary McGraw. *Java Security: Hostile Applets, Holes & Antidotes*. Wiley and Sons, 1996.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL04)*, pages 186–197. ACM Press, 2004.

[GM05] R. Giacobazzi and I. Mastroeni. Adjoining declassification and attack models by abstract interpretation. In *Proceedings 14th European Symposium on Programming (ESOP05), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 295–310, Edinburgh, UK, 2005. Springer.

[GS05] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proceedings of 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI05)*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362, Paris, France, 2005. Springer.

[GSR08] D. Ghindici and I. Simplot-Ryl. On practical information flow policies for java-enabled multiapplication smart cards. In *Proc. 8th IFIP Conf. on Smart Card Research and Advanced Applications (CARDIS08)*, volume 5189 of *Lecture Notes in Computer Science*, pages 32–47, Egham, Surrey, UK, 2008. Springer.

[GSRT07] D. Ghindici, I. Simplot-Ryl, and J.-M. Talbot. A sound dependency analysis for secure information flow (extended version). Technical Report 0347, INRIA, 2007.

[GSRT09] D. Ghindici, I. Simplot-Ryl, and J.-M. Talbot. A sound analysis for secure information flow using abstract memory graphs. In *Proc. 3rd International Conference on Fundamentals of Software Engineering (FSEN'09)*, volume to appear of *Lecture Notes in Computer Science*, page to appear, Kish Island, Persian Gulf, Iran, 2009. Springer.

[GT05] L. Georgiadis and R. E. Tarjan. Dominator tree verification and vertex-disjoint paths. In *Proceedings of the 16th annual ACM-SIAM Symposium on Discrete algorithms (SODA05)*, pages 433–442, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[HBCC99] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[HKM05] B. Hicks, D. King, and P. McDaniel. Declassification with cryptographic functions in a security-typed language. Technical Report NAS-TR-0004-2005, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, 2005.

[HKS06] C. Hammer, J. Krinke, and G. Snelting. Information flow control for java based on path conditions in dependence graphs. In *Proceedings IEEE International Symposium on Secure Software Engineering (ISSSE06)*, pages 87–96, Arlington, VA, 2006.

[Hod07] Y. Hodique. *Sûreté et optimisation par les systèmes de types en contexte ouvert et contraint*. PhD thesis, Univ. Lille 1, France, 2007.

[HP06] R. R. Hansen and C. W. Probst. Non-interference and erasure policies for java card bytecode. In *6th International Workshop on Issues in the Theory of Security (WITS06)*, 2006.

[HS06] S. Hunt and D. Sands. On flow-sensitive security types. In *Proceedings 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL06)*, pages 79–90. ACM Press, 2006.

[Jav] Java In The Small. http://www.lifl.fr/POPS/JITS/.

[JG96] H. M. James Gosling. The Java language environment: A white paper. Technical report, 1996.

[JS04] B. Jeannet and W. Serwe. Abstracting call-stacks for interprocedural verification of imperative programs. In *Proceedings 10th International Conference of Algebraic Methodology and Software Technology (AMAST04)*, volume 3116 of *Lecture Notes in Computer Science*, pages 258–273, Stirling, Scotland, UK, 2004. Springer.

[KS02] N. Kobayashi and K. Shirane. Type-based information flow analysis for low-level languages. In *Proceedings 3rd Asian Workshop on Programming Languages and Systems (APLASO2)*, pages 302–316, Shanghai, China, 2002.

[KV04] O. Kolsi and T. Virtanen. MIDP 2.0 security enhancements. In *In Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS04)*, page 90287.3, Washington, DC, USA, 2004. IEEE Computer Society.

[Lam73] B. W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[Lau01] P. Laud. Semantics and program analysis of computationally secure information flow. In *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP01)*, pages 77–91, London, UK, 2001. Springer.

[LBR06] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[Ler03] X. Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reason.*, 30(3-4):235–269, 2003.

[LR04] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *ACM SIGPLAN Notices*, 39(4):473–489, 2004.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[LT04] Y. M. Liu and I. Traore. Uml-based security measures of software products. 1st Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES04), Hamilton, Ontario, Canada, 2004, within the 4th International Conference on Application of Concurrency to System Design 2004 (ACSD04), 2004.

[LY99] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[LZ05] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. *ACM SIGPLAN Notices*, 40(1):158–170, 2005.

[ML97] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings 16th ACM symposium on Operating systems principles (SOSP97)*, pages 129–142. ACM Press, 1997.

[ML98] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 19th IEEE Computer Society Symposium on Research in Security and Privacy (RSP98)*, 1998.

[ML00] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[MM08] K. E. Mayes and K. Markantonakis. *Smart Cards, Tokens, Security and Applications*, chapter Multi Application Smart Card Platforms and Operating Systems, pages 51–83. Springer, 2008.

[MSZ04] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW04)*. IEEE Computer Society Press, 2004.

[MSZ06] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Comput. Secur.*, 14(2):157–196, 2006.

[Mye99a] A. C. Myers. Jflow: practical mostly-static information flow control. In *Proceedings 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL99)*, pages 228–241. ACM Press, 1999.

[Mye99b] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, Cambridge, 1999.

[Nau05] D. A. Naumann. Verifying a secure information flow analyzer. In *Proceedings 18th International Conference Theorem Proving in Higher Order Logics (TPHOLs05)*, volume 3603 of *Lecture Notes in Computer Science*, pages 211–226, Oxford, UK, 2005. Springer.

[Nec97] G. C. Necula. Proof-carrying code. In *Proceedings 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL97)*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[NIS] NIST. An introduction to computer security: the nist handbook.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[PS02] F. Pottier and V. Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL02)*, pages 319–330, Portland, Oregon, 2002.

[PS03] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003. ©ACM.

[RMB05] A. C. Ricardo Medel and E. Bonelli. Non-interference for a typed assembly language. In *LICS'05 Affiliated Workshop on Foundations of Computer Security (FCS05)*, 2005.

[RR98] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA98*, 1998.

[Sal01] A. D. Salcianu. Pointer analysis and its applications for java programs. Master's thesis, Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, 2001.

[Sal06] A. D. Salcianu. *Pointer analysis for java programs: novel techniques and applications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2006. Adviser-Martin C. Rinard.

[SBN04] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Proceedings of 11th International Static Analysis Symposium (SAS04)*, volume 3148 of *Lecture Notes in Computer Science*, pages 84–99. Springer, 2004.

[Sim03] V. Simonet. Flow Caml in a nutshell. In *Proceedings of the first Applied Semantics II (APPSEM-II) Workshop*, pages 152–165, Nottingham, United Kingdom, 2003.

[Sim04] V. Simonet. *Inférence de flots d'information pour ML: formalisation et implantation*. PhD thesis, Université Paris 7, 2004.

[SM03] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[Smi01] G. Smith. A new type system for secure information flow. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations (CSFW01)*, pages 115–125, Washington, DC, USA, 2001. IEEE Computer Society.

[SR01] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. *ACM SIGPLAN Notices*, 36(7):12–23, 2001.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL99)*, pages 105–118, New York, NY, USA, 1999. ACM.

[SS05] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings 18th IEEE workshop on Computer Security Foundations (CSFW05)*, pages 255–269, Washington, DC, USA, 2005. IEEE Computer Society.

[SS07] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, page to appear, 2007.

[SST07] P. Shroff, S. Smith, and M. Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings 20th IEEE Computer Security Foundations Symposium (CSF07)*, volume 00, pages 203–217, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[ST07] S. F. Smith and M. Thober. Improving usability of information flow security in java. In *Proceedings of the 2007 workshop on Programming languages and analysis for security (PLAS07)*, pages 11–20. ACM Press, 2007.

[Sun] Sun Microsystem. Connected Limited Device Configuration and K Virtual Machine, http://java.sun.com/products/cldc/.

[Sun08] Q. Sun. *Constraint-based Modular Secure Information Flow Inference For Object-Oriented Programs*. PhD thesis, Stevens Institute of Technology, 2008. Adviser-David Naumann.

[The04] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, 2004. `http://coq.inria.fr`.

[TZ05] S. Tse and S. Zdancewic. A design for a security-typed language with certificate-based declassification. In *Proceedings 14th European Symposium on Programming (ESOP05), Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 279–294, Edinburgh, UK, 2005. Springer.

[VIS96] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.

[VS97] D. M. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT97)*, pages 607–621, London, UK, 1997. Springer.

[WR99] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA99)*, volume 34 of *ACM SIGPLAN Notices*, pages 187–206, Denver, Colorado, 1999. ACM Press.

[XQZ$^+$05] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[YI06] D. Yu and N. Islam. A typed assembly language for confidentiality. In *Proceedings of 15th European Symposium on Programming Languages and Systems, (ESOP06)*, volume 3924 of *Lecture Notes in Computer Science*, pages 162–179. Springer, 2006.

[Zan06] D. Zanardini. Abstract Non-Interference in a fragment of Java bytecode. In *Proceedings of the ACM Symposium on Applied Computing (SAC06)*, Dijon, France, 2006. ACM Press.

[Zda04] S. Zdancewic. Challenges for information-flow security. The First International Workshop on Programming Language Interference and Dependence (PLID04), 2004.

[ZM01] S. Zdancewic and A. C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW01)*, pages 15–23. IEEE Computer Society Press, 2001.

[ZM02] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher Order Symbol. Comput.*, 15(2-3):209–234, 2002.

List of Figures