Information flow and declassification analysis for legacy and untrusted programs

Information Flow and Declassification Analysis for Legacy and Untrusted Programs

Bruno Pontes Soares Rocha

Printed by Printservice Technische Universiteit Eindhoven.

Cover design by Verspaget & Bruinink.

CIP-DATA LIBRARY TECHNISCHE UNIVERSITEIT EINDHOVEN

Pontes Soares Rocha, Bruno

Information Flow and Declassification Analysis for Legacy and Untrusted Programs / by Bruno Pontes Soares Rocha. – Eindhoven: Technische Universiteit Eindhoven, 2012.
Proefschrift. – ISBN 978-90-386-3050-2
NUR 980
Subject heading: Computer security
A catalogue record is available from the Eindhoven University of Technology Library.

Information Flow and Declassification Analysis for Legacy and Untrusted Programs

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op woensdag 15 februari 2012 om 16.00 uur

door

Bruno Pontes Soares Rocha

geboren te Belo Horizonte, Brazilië

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. S. Etalle

Copromotor:
dr. J. den Hartog

Para minha amada Mari, sempre tão perto, mesmo de tão longe.

And in the loving memory of my good friend Will, whose kindness, friendship and support won't ever be forgotten.

Acknowledgements

I would like to thank many people that supported me during my PhD in Eindhoven. First of all, I am very grateful for my supervisors and co-workers, Sandro Etalle, Jerry den Hartog and William H. Winsborough.

I thank my promotor Sandro and my daily supervisor Jerry, for their support, comprehension and friendship. Thanks a lot for putting your trust on me and letting me try the most ambitious ideas, even when they sounded pure nonsense. Trying to pursue research goals outside of our group's specialty, and to develop a complete new research approach was a big challenge. One that I'm really proud of. It was exactly the scope of the challenge that made me so excited about this work. And the fact that we could truly argue about it, on an equal level, made this exchange of ideas so fruitful. Thanks Sandro for helping me better organize and present my ideas. Thanks Jerry for helping me with all the mathematical background. And thank you both for being such good friends during this time.

Unfortunately Will will not be able to read these acknowledgements. In any case, I would like to thank him, for his support, his advices (both professional and personal) and his friendship. I believe that his influence is made present in lots of parts of this work. Will was a "keeper", and it is a great honor for me that we have shared so much together. Godspeed, my friend.

During this time I also had some great co-workers, which I would like to thank. Thank you Sruthi Bandhakavi, for the great help and the very nice discussions we had during several months. It was during these talks that we managed to develop some of the most challenging parts of this work. I also thank Mauro Conti and Bruno Crispo for the joint work we have made throughout this last year, trying to combine a bunch of different things in some nice research work. Thanks also to Mark van den Brand and Alexander Pretschner for the reviewing and helpful comments on this thesis. I would also like to thank Gilles Barthe for having recommended me to make my PhD in Eindhoven, and for some helpful discussions on my research path.

Thanks to the other members of the S-Mobile project for the support and exchange of ideas during our project meetings. Thanks to the people at the Chrome OS Security team at Google, in particular to my host Gaurav Shah, for the nice experience and learning period I had there, where I had the pleasure of being an intern.

I am very grateful for the friends I made. Thanks a lot, in particular, Daniel, Elisa, Mayla and Antonino for the friendship, the jokes, the fun, the beer, the dinners, the annoying seminars together, the coffee breaks, the breaks that didn't really need motives,

and many other things. I will miss you guys. Also Jolande for always being so cheerful and helpful, besides keeping up with my peculiar morning mood [1]. Thanks also to others, outside of my working group, with whom I shared nice moments in Eindhoven, Juan Carlos, Bea, Alberto, Antonio and Daniela. And also to the other members of the SEC, CC and DAM groups which were constant and pleasant presences during breaks, talks, seminars and others: Shona, Irene, Jing, Relinde, Gaetan, Christiane, Nicola, Fred, Kostas, the three Peters, Sebastiaan, Henk, Jan Jaap, Meilof, Dion, Boris, Milan, Berry, Benne, Tanja, Dan and Anita. Also, I thank Nespresso and Jack Daniels for their contribution in keeping me alive, awake and sane during these four years.

I would like to thank my parents, Guilherme and Jacqueline, for the unconditional support and pride, since always, even from afar. Also to my sisters, Tatiana and Camila, and to my grandmothers Darcy and Jane for all the support and for always wishing the best. To my friend Miguel for all the friendship, and for all the good moments we managed to share, both during good and bad times. Also to all my friends and family in Brazil, who at distance followed my progress in these past years, and that by one way or another always showed me their support.

Gostaria de agradecer aos meus pais, Guilherme e Jacqueline, pelo incondicional apoio e orgulho, desde sempre, mesmo de longe. Também às minhas irmãs, Tatiana e Camila, e às minhas avós Darcy e Jane por todo o apoio e por estarem sempre torcendo. Ao meu amigo Miguel por toda a amizade, e por todos os bons momentos que conseguimos compartilhar, em tempos bons e ruins. Também a todos os amigos e familiares no Brasil, que acompanharam à distância meu progresso nesses últimos anos, e que de uma forma ou outra sempre me passaram seu apoio.

I would like to especially thank my dear wife Mariana. For always being so close, feeling every bit of the struggle, the joys and the sorrows. If we could not be together physically, we were always together in our thoughts, hearts and souls. For all we have been through together in these four years, depriving ourselves of so much, and waiting for so much time, all for the sake of a certainty we never dismissed. Because if some teachings are more important and difficult than any academic knowledge, some rewards are bigger than all money in the world. I am very grateful that we have learned together that life is not about what to do or where to go, but who to be with.

Gostaria de agradecer especialmente à minha querida esposa Mariana. Por estar sempre tão perto, sentindo cada pedaço da luta, das alegrias e das tristezas. Se não pudermos estar juntos de corpo, estivemos sempre em pensamentos, coração e alma. Por tudo que passamos juntos nesses quatro anos, nos abdicando de tantas coisas, e esperando por tanto tempo, tudo em nome de uma certeza que nunca deixamos de ter. Porque se alguns aprendizados são mais importantes e difíceis que qualquer conhecimento acadêmico, algumas recompensas são maiores que todo o dinheiro do mundo. Sou grato por termos aprendidos juntos que a vida não é sobre o que fazer ou onde ir, mas sim com quem estar.

[1] The rest of you probably deserved it.

Contents

1 Introduction **1**
- 1.1 Information Flow and Declassification Analysis . . . 3
- 1.2 Current Approaches . . . 4
- 1.3 Research Question . . . 7
- 1.4 Contributions . . . 8
- 1.5 Related Work . . . 11
- 1.6 Plan of the Thesis . . . 16

2 Expression-Matching Framework **19**
- 2.1 Motivating Examples . . . 19
- 2.2 Language: Syntax and Semantics . . . 24
- 2.3 Program Validity . . . 28
- 2.4 Policy Controlled Release . . . 30
- 2.5 Soundness of the Framework . . . 31

3 Graph-Based Implementation **35**
- 3.1 Revisiting the Examples . . . 35
- 3.2 Expression Graphs . . . 40
  - 3.2.1 Program Graphs . . . 41
  - 3.2.2 Policy Graphs . . . 45
- 3.3 Graph Matching . . . 46
- 3.4 Soundness of the Implementation . . . 52
- 3.5 Algorithms and Tractability . . . 54
- 3.6 User-Defined Functions . . . 65

4 Hybrid Static-Runtime Enforcer **69**
- 4.1 Motivating Examples . . . 69
- 4.2 Approach . . . 72
  - 4.2.1 Preliminary assumptions . . . 73
- 4.3 Static Analyzer . . . 73
- 4.4 Pre-load Checker . . . 78
- 4.5 Runtime Enforcer . . . 83
  - 4.5.1 Overhead . . . 85

5 Open Problems **91**
5.1 Information Path Filtering . . . 91
5.2 Loop Iteration Counting . . . 96
5.3 Algebraic Equivalence of Expressions . . . 97
5.4 Graph-Based PCR on a Real Programming Language . . . 98
5.5 Policy Controlled Release on Assembly Code . . . 100

6 Concluding Remarks **103**
6.1 Contributions . . . 103
6.2 Limitations . . . 104
6.3 Discussion . . . 105
6.4 Future Work . . . 106

Bibliography **107**

A Proofs **117**
A.1 Proof of Lemma 2.19 . . . 117
A.2 Proof of Theorem 3.6 . . . 120
A.3 Proof of Theorem 3.13 . . . 123
A.4 Proof of Theorem 3.15 . . . 125
A.5 Proof of Theorem 3.21 . . . 125

B Source Code **127**

Index **141**

List of symbols **141**

List of figures **149**

List of algorithms **151**

Summary **153**

Curriculum Vitae **155**

CHAPTER 1

Introduction

Computer systems are constantly handling sensitive information. As most of such systems are networked and often connected to the Internet, sensitive data is also regularly transmitted between different devices. Smartphones and tablet computers carry private data of users in the form of contact lists, photos, messages and others. Social network websites not only store such information, but also regulate who has access to it. Banking systems are responsible for securing and regulating access to very sensitive financial information of their customers. Wrong handling of sensitive information can cause it to be disclosed to unauthorized parties, corrupted or lost. This can cause major loss for both companies and individuals.

The field of *computer security* can be divided into 3 main aspects: *confidentiality*, *integrity* and *availability*. Confidentiality is related to ensuring that data is only accessible by entities authorized to do so. Integrity is about preventing that data gets corrupted or modified in unauthorized ways. Finally, availability is about guaranteeing that computer systems and services are available at all times. This thesis tackles the first aspect, confidentiality of sensitive data. Here, we aim at the problem of ensuring that programs do not leak sensitive information to unauthorized entities.

Software is the fundamental decision-making component of a computer system. Every action done in data, including modification, copy, deletion and transmission is done by programs. Thus, in order to regulate actions over data, one needs to regulate how computer software operates over such data. To make matters more complicated, it is common for a computer system to have a multitude of different programs, which are in turn also regularly updated. Thus, in order to enforce how sensitive information is handled by a computer system, one needs to regulate what programs do with this information.

Information can have different degrees of confidentiality. In a company, some information might be of public domain, e.g. the company's line of products and services, its address, some general numbers about profits. However, an employee should not be able to access another employee's salary information, while a manager should be able to access this information related to all his/her subordinates. Some information might be even more sensitive: details of unannounced research projects or the company's financial situation should be accessible only by some key personnel. With this, there is a need for computer systems to regulate "who" (i.e. programs working on behalf of users) can access which kind of information.

Current computer systems have mechanisms that regulate which resources a program is allowed to access. For instance, a program may have access to a number of files, but

not to those under the `system` folder. In a typical multi-user system, a program running on behalf of user Alice may not be allowed to access files owned by user Bob. In mobile devices such as smartphones each program is typically accompanied by a manifest of all resources it can potentially access, such as the phone's camera, wi-fi, text messages, etc. The user can then deny a program's installation if he/she does not agree with that manifest. These mechanisms that regulate the resources programs can access are collectively known as *access control* mechanisms. In our company example an access control mechanism would, e.g. not allow a program executed by Alice to access a file owned by Bob, unless Bob marked that file as open for public access.

Access control is however not sufficient to guarantee that information is not leaked by a program: once access to a resource is granted, a program can do anything with it. For instance, consider a company policy in which access to salary information of an employee is only granted to the employee him/herself and his/her manager. Now, consider a program running on behalf of one of the managers of a company. As per access control rules, this program has access to the salary information of that manager's subordinates, and also to open network connections, as the program also accesses information from the Internet. In this case, the program can e.g., read the salary information of some employees and transmit it, via the network connection, to a computer outside of the company. This computer can then belong to entities which should not be authorize to access salary information. Thus, although this program satisfies the constraints of access control, it can potentially disclose the salary information to unknown (and unsafe) entities. Here, we want to control not only which resources a program can access, but what it does with it. Thus, a more elaborate mechanism is necessary.

There are different ways for a program to leak sensitive information. The aforementioned example is of an explicit flow of information: the sensitive data (salary information) is transmitted to a potentially unsafe entity (over an arbitrary network connection). Leaking data derived from sensitive data should also be avoided: it is still unsafe to disclose, e.g. the difference between the salaries of Alice and Bob, as some knowledge of both salaries can be inferred by this information. Finally, there are also implicit ways of disclosing information. Consider a program that writes some data to a file, and that this data is completely unrelated to any employee's salary information. However, this program follows some logic in which it only needs to write to the file if Bob's salary is above a certain threshold. Even though this program does not directly leak Bob's salary, it does it implicitly. By observing whether the program wrote to the file or not, one can infer if Bob's salary is above the threshold. If the file is accessible to entities that should not be able to access Bob's salary, then we have a potential leak of sensitive information.

Guaranteeing that programs do not leak sensitive information is a highly desired goal, but restricting the disclosure of all derived data is often overly strict. Real programs often need to leak information on purpose, under controlled circumstances. Let us go back one last time to the company example: a company policy may determine that individual salaries are to be kept secret, but the average salary of the company (or a given department) is allowed to be publicly disclosed. A program that makes such calculation would be leaking sensitive information, as the average salary is a derivation of such secret data, which is then disclosed to public channels. Such exceptions are very common, and most

computer systems need some type of it. Thus, in order for a mechanism to be deployed on real-world systems, one also needs a way to specify and regulate exceptions to information confidentiality.

This thesis treats the problem of ensuring that programs guarantee information confidentiality, with support to well-defined and controlled exceptions, in order to make the mechanism applicable to real-world programs. We also aim at solutions which can be potentially applied to *any* computer program, not only those designed specifically to be safe. In the next section we name and define technically the concepts discussed here. In the following sections we detail the research problem tackled by this thesis, as well as our contributions to solve it.

1.1 Information Flow and Declassification Analysis

The problem we have seen in the previous section can be formalized in terms of *information flow and declassification analysis*. In this section we technically define and detail such concepts.

Programs dealing with sensitive data must prevent confidential information from flowing to unauthorized entities [SM03b]. In order to enforce how programs use data, information flow control has became increasingly popular within the scientific community. Information flow control revolves around a classical security property called *non-interference* [GM82], which states that the publicly observable behaviour of a program is entirely independent of any secret input values it has received. Several techniques have been proposed to check whether programs satisfy this property, within both static analysis and runtime enforcement.

We illustrate the property of non-interference with an example. Consider a program with 2 inputs: one which is publicly observable, labeled *low*, and therefore not confidential, and another which is secret, labeled *high*, and whose contents should be disclosed to unauthorized entities. This program has one public output, labeled *low*. It could also have secret (*high*) outputs, but these are unnecessary for the sake of this example. Consider that $P(l, h)$ returns the program's public output for when it is executed with the input values l and h, for the *low* and *high* inputs, respectively. We say that this program satisfies non-interference if, for any two executions differing only in the value of the high input, the value of the low output does not change. In other words, for any l, h and h', we have that $P(l, h) = P(l, h')$. In this case, we say that the secret input does not interfere with the value of the public output, and thus this program does not perform any unauthorized information flow. If the secret input is e.g., a personal contact list, and the public output a network connection, we could state that this program does not disclose data from the former via the latter.

In general, non-interference is excessively restrictive: many programs that meet their security objectives fail to satisfy it. The problem is that real programs often need to, under specific circumstances, support exceptions to standard information flow control, allowing secret data to flow to public outputs. Consider the following examples:

1. In a system where a given block of data is considered secret, transmitting this data

over a network connection might violate non-interference. However, if a program encrypts the data using an algorithm known to be safe, it might be desired to allow the network transmission to happen. Here, the secret block of data should acquire a lower security level upon being subject to a specific data operation – in this case, the encryption algorithm. Information flow control would, however, consider the encrypted block of data to be a variant of the original block, and thus also being labeled as secret.

2. Consider a password checking mechanism. In a typical information flow scenario the password provided by the user should be labeled secret, and not allowed to flow into insecure outputs. The monitor screen which the user uses to interface with the system is one of such insecure outputs – we do not want the password to be shown there. However, should the password verification fail, the user must be informed about it, via an error message. But this error message will only be displayed when a function on the password (e.g. $verify(passwd)$) returns a failure result. With this, the showing of the error message reveals some information on the password (i.e. it is not the correct one). We call this an *implicit information flow*, since the error message does not actually reveal the password. This means that this program does not satisfy non-interference. Note that the exception to non-interference is not only desired, but *necessary*, in order for this program to work properly while still being secure from the information flow point of view. The needed exception would specify that, for a secret $passwd$, the boolean value returned by $verify(passwd)$ can be disclosed to a lower security level.

3. A more elaborate example is the one discussed in the previous section: a company policy that requires individual employee salaries be kept secret, but allows the average salary to be disclosed. Since non-interference prohibits any direct or indirect flow of secret information to a public output channel, *any* program that publishes the average salary violates it. This example shows not only the need for such exceptions, but also for a way to specify them in detail.

Here we can see examples where there is a need to release data that depends on secret information but that: does not actually reveal anything (1), not anything important (2), or only information that was specifically intended to be released (3). An exception to non-interference that allows secret data to be released to a public channel is called a *declassification*. To be able to make the distinction between intended release and unintended leakage of secret information we need a specification of allowed exceptions – such specifications are called *declassification policies* (see e.g., [SS05]).

1.2 Current Approaches

Most of the approaches to guarantee information flow can be divided into two main categories: static analysis and runtime enforcement. Static analysis consists of analyzing the program's code in order to predict its behaviour, while runtime enforcement revolves

around checking, during runtime, every instruction executed by the program, and taking action should an unauthorized action takes place. These two approaches are complementary: some aspects of information flow can only be tackled by static analysis (e.g. implicit flows) while others only by runtime enforcement (e.g. resources with labels only known at runtime). These techniques also have their own limitations and shortcomings, though. We now explain them in more detail.

Static Analysis. There are some existing static analysis approaches that tackle the problem of information flow and declassification analysis. Type-based [Mye99, PS03, VIS96] and dataflow-based [AB04, ABB06, BBM94, Den76] approaches have been proposed to statically analyze whether a given program enforces non-interference. In both approaches, each program variable is *labeled* with a security level (e.g., *high* for secret or *low* for public, though any lattice of labels can be supported). In type-based approaches, a special programming language is used to annotate program variables with security types. Typing rules are defined such that if the program type-checks (i.e. all assignments between variables satisfy the security types), then it is non-interferent. In dataflow-based approaches, an analysis calculates dependence relationships between program variables; non-interference is ensured if low variables are independent from high variables.

In type-based approaches exceptions to the standard flow are usually associated with specific points in the code. The programmer can specify the declassification policy by using a special *declassify* command, which releases the information conditionally, depending on the value of a given expression over program variables. If at run-time this conditional expression is true, declassification is allowed. In frameworks of this kind, declassification policies are specified in a manner that is intimately tied to the program itself.

Figure 1.1 shows an example of a type-based program using declassification. This example program calculates the average value of several records, all labeled secret, and then sends it to some public output. Note that in type-based approaches each variable has both a data type (`int`) and a security label (`secret` and `public`). Here, *numRecords()* returns the (public) number of records in the storage, while *getRecord(i)* returns the secret record on the i-th position. For the expression sum/i to be assigned to the `public` labeled variable avg, a declassification must be made. This is done by the *declassify* command, which takes an expression and downgrades it to a new security label.

A drawback of this approach is that only someone with a deep understanding of the program can reliably write declassification policies for it. Everyone else is forced to trust blindly that the policies meet the required security objectives. When code is written by trusted programmers, this assumption may be acceptable, though even then it would be preferable to separate concerns and make the specification, maintenance, and review of declassification policies independent from the program. In the case of untrusted code, or code without security annotations, relying on the programmer to identify declassification points is clearly unacceptable. Operators of systems that rely on such a program obtain little assurance that the declassification policies defined in it are appropriate. As pointed out by Zdancewic [Zda04], one of the reasons why language-based techniques have not yet been widely adopted is that the enforcement approaches require the programmer to

```
int{secret} sum := 0;
int{public} i := 0;
int{public} avg := 0;

while (i < numRecords()) do
    sum := sum + getRecord(i);
    i := i + 1;

avg := declassify(sum/i, secret to public);
doOutput(avg);
```

Figure 1.1: Type-based approach for declassification

worry not only about the correctness of the program logic, but also about how to annotate the program so that it can be deemed secure.

This makes analysis of *legacy and untrusted* programs impossible, and thus it is not very practical. In this context, legacy and untrusted programs are those which satisfy two points. First, they were developed before (or unaware of) the technology used to analyze them. This way, these programs were not developed using any specific security-oriented technology in order to make their analysis any easier. Second, acquired from an unknown or untrusted source, so that there are no guarantees if the program actually does exactly what is stated in its specification, and nothing else (or if there is any specification at all). Note that many programs downloaded from the Internet are often both legacy and untrusted, and that the aforementioned limitation of many static analyzers would make analysis of such programs impossible.

To stress this point further, work from Hicks et al. [HKMH06] conclude that although Jif [CMVZ06] is the most advanced security typed programming language, it is not ready for mainstream use because it requires considerably more programming effort to write a working program than in a conventional language. In light of this observation, we believe there is need for an information flow analysis framework that does not require programming annotations and which considers programs and policies as independent entities. This would result in greatly reducing the effort required to program an application, decoupling the program from the policy, and avoiding the need to trust the code itself.

Dataflow approaches, on the other hand, calculate dependencies between different structures of a program, thus being able to analyze code without security annotations. However, since they do not differentiate between the different operations applied to data, they do not manage to handle declassification in an automatic way.

This state of affairs implies that declassification policies cannot readily be applied to legacy code. Unless the legacy program satisfies strict non-interference (which is uncommon) the only way to determine whether such programs satisfy information-flow objectives is through the laborious process of understanding the program well enough to design a program-specific declassification policy.

Runtime Enforcement. In the classical definition of runtime enforcement [Sch00], an enforcer must track *all* program instructions in order to detect security violations. Other techniques (e.g. edit automata [LBW05]) revolve around re-writing program instructions on the presence of violations. Dataflow-based approaches can also be implemented as runtime enforcers, and one such example is taint analysis [LL05, TPF+09], which keeps track of program modules/structures which are dependent of (i.e. "tainted by") sensitive information.

Runtime enforcers, however, often cause a non-negligible processing overhead on the monitored programs, since the enforcer itself needs processing cycles and memory. Usually, the more policies the enforcer can support, the more overhead it causes: program re-writing, for instance, causes the enforcer to potentially add more instructions to the running program. Performing declassification in runtime, when possible, can be even more computationally demanding: in order to check if policies are satisfied, the enforced needs not only to track the *current* executing instruction, but also to keep track of previous operations done over data by the program. This is needed as programs may use several instructions in order to calculate a derivation allowed by a declassification policy (e.g. an arithmetic average).

Some approaches have been recently proposed to combine static analysis and runtime enforcement, thus reducing runtime overhead. These, however, rely much on the expressive power of the static analyzer in order to make the runtime component lightweight. In current approaches, either (1) the static analyzer is type-based, thus reducing runtime overhead at the cost of introducing a security-annotated language, or (2) the system does not support declassification policies, as these need either annotations or cause a heavy runtime overhead.

Thus, runtime enforcers have a limited domain of target systems/programs they can be executed on. For instance, a *mobile* program is executed on mobile devices, such as smartphones and tablets, and usually aims to keep a low execution overhead (processing, memory and battery). Here, the processing overhead caused by runtime enforcers is highly undesirable. And yet, supporting this kind of programs is very important, as mobile devices are increasingly present.

In conclusion, declassification mechanisms in state-of-the-art information flow approaches are not sufficient to meet the needs of practical analysis of real programs (i.e. to be able to analyze legacy, untrusted and mobile code). The declassification mechanism needed to deal with practical exceptions to non-interference is not available in an appropriate way for most use cases.

1.3 Research Question

The research question to be answered by this thesis arises from the limitations of current information flow and declassification mechanisms, noted in the previous section. Our main goal is to make information flow analysis possible on real-world applications, and this includes untrusted and legacy programs, possibly executed on mobile devices. Achieving this goal will take not only a large step to bring this kind of analysis to real and

deployed systems, but as well to fill the gap left by current research in the field.

Thus, in order to accomplish this, we formulate the research question as follows:

How to check information flow on legacy, untrusted and mobile code?

In order to answer this question, we need an information flow analysis mechanism which satisfies the following objectives:

1. Analyzes unannotated code, i.e. code written by a possibly untrusted, unknown programmer.

2. Supports declassification, as real applications often need to declassify information.

3. Allowed declassifications should be specified by declassification policies which are independent from the code, i.e. code and policy are written by separated entities, independently.

4. Can be implemented with decidable algorithms.

5. Must be adaptable in order to work in multiple systems, with little to no runtime overhead.

The mechanism we aim at must be able to decouple declassification analysis from the source code, thus being able to analyze untrusted and legacy programs. Also, low runtime overhead is necessary in order to allow the mechanism to work on mobile devices.

1.4 Contributions

In this thesis we answer the research question by introducing a set of mechanisms, from theory to practice, that allow the specification, verification and enforcement of declassification policies that are independent from the code to which they are applied. We introduce a novel approach for information flow and declassification analysis, laying the groundwork for bringing this kind of analysis to deployed systems. We present our approach as 3 mechanisms, each building upon its predecessor, from theory to practice:

1. A theoretical *framework* that defines a policy model and the notion of program validity with respect to a policy.

2. A high level *implementation* that defines a concrete policy language and a tractable validation procedure for checking program validity against such policies.

3. A practical *extension* of the implementation, that defines a framework which combines the previous mechanism with a runtime component implemented on a established technology, supporting more expressive policies across multiple systems, but keeping runtime overhead very low.

All of the mechanisms presented on this thesis are novel by offering the combination of the following features: (a) support to user-defined declassification policies, (b) code and policy being separated and independent from each other, (c) analysis and application of declassification policies to unannotated and untrusted code. Recalling the points from the previous section, our theoretical framework tackles points 1 (analyzes unannotated code), 2 (supports declassification) and 3 (independent declassification policies). The high level implementation adds point 4 (implemented by decidable algorithms), at the cost of slightly reducing the precision of the analysis. The practical extension then enhances the expressiveness of policies (3) and brings the approach to mobile systems, with little to no runtime overhead, tackling point 5. With this, we end with a practical mechanism that satisfies all points of the previous section. Below we treat each of these mechanisms and their benefits over existing approaches in more detail. We then provide an overview of related work in Section 1.5 before outlining the thesis plan in Section 1.6, which describes how we organize our contributions.

The theoretical *framework* defines program validity in terms of the expressions on inputs that the program calculates. A program is deemed safe according to how expressions it calculates are checked against a given set of expressions which are allowed to be released, i.e. a set of *declassifiable expressions*. Our approach to program analysis deems a program to be safe if it is able to determine that public output values depend on secret inputs only via such expressions. Programs can be written without awareness of the formal declassification policies or of how the analyzer works, as no special command is used to specify declassification or security labels. Technically, a fundamental contribution of the theoretical framework is the introduction of a property called Policy Controlled Release (PCR) — a more flexible security property that replaces non-interference — and a result that shows this property is satisfied by programs deemed valid by our analysis.

The second mechanism, the high level *implementation* of the framework, consists of a tractable analysis for determining whether a specific graph-based form of a declassification policy is enforced by the input program. It represents one possible way of implementing the framework in a tractable way, providing a basis for further work on even more expressive representations.

On our implementation, declassification policies use *graphs* to represent sets of expressions over values obtained from input channels. This allows us to express and to deal efficiently with declassification policies that refer to iterative constructs such as loops (as in the example in which the average salary may be disclosed and the individual wages must remain secret). The policies represent values that are permitted to be made public. Expressions that may be computed by the program under analysis are also represented by a form of an expression graph that incorporates representations of variables and I/O channels, and captures the dependencies of output expressions on values obtained from input channels. We augment the power of our expression graphs to allow them to express the (non-regular) property that values obtained from input channels are given by distinct read operations, thus enabling our policies to require, for instance, that an expression representing the average of input values must refer to multiple distinct values read from the input channel, and not multiple references to the value returned by a single read operation. A graph matching mechanism is used to ensure that the expressions are declassifiable per

the policy.[1] This notion of *policy simulation* is reminiscent of the standard concept of bisimulation in automata [Mil89].

In present approaches, to declassify the result of a looping program using standard flow-based techniques, one is required to manually introduce simplifications, which often consist of determining the fix-points of loops. On the other hand, type-based techniques usually rely on the programmer to identify in the code iterative declassification expressions.

Finally, the third mechanism, a practical *extension* of the graph-based approach, achieves the goal of being suitable for working on currently deployed mobile technologies, also adding a runtime component that enhances the expressiveness of policies while incurring little to no runtime overhead. It does so by extending the graph-based approach and combining it with other components. Although a purely static mechanism is a highly desirable research goal, there are certain aspects of software analysis which require runtime information to be enforced. This is why we extend our static solution and combine it with a runtime enforcer. We show how this hybrid solution be deployed on real systems, and also make explicit which kinds of policy aspects can be enforced by each approach (static and runtime).

The presented hybrid static-runtime enforcement approach has 3 stages: (1) our slightly modified static analyzer that takes a program source and a set of declassification policies and detects all flows of information between input and output channels in the program, as well as detecting points where declassification can happen (generating constraints that have to be checked at runtime); (2) a pre-load checker which, before loading the program for execution, checks the security labels of I/O operations specific to the target system against the information obtained in the previous step; and (3) a runtime enforcer that checks labels which are only known at runtime, as well as runtime constraints for the declassification policies. Calls to the enforcer are injected in the application's code, prior to its execution, on the specific points where checks are needed, thus further reducing the overhead of the enforcer. We present three motivating examples, all within the context of a mobile device, and show that our hybrid static-runtime enforcement allows to:

- support more realistic policies than present approaches—as policies may need both static (implicit flows, declassification) and runtime (dynamic labels, execution constraints) knowledge;

- achieve an often negligible runtime overhead—as most of the analysis computation is done statically, and the static analyzer is system independent.

The presentation of the hybrid enforcer is guided by the three examples and the approach is presented in an implementation-oriented fashion. That is, we do not present an in-depth formalization of the domain of problems solvable by that mechanism. However, we define it in such a way that a full implementation is straightforward, and demonstrate

[1] While PCR is termination-sensitive, our analysis and theorem are termination-insensitive in the sense that our analysis may deem valid a program that leaks secret information by failing to terminate during a while loop that is controlled by a nondeclassifiable expression.

its applicability directly over the examples, which consist of real-world scenarios, not treated by existing practical approaches.

We implement our final step, the runtime enforcer, and run it against benchmark programs on an Android device, in order to determine its overhead. We show that for most practical scenarios, the overhead is almost imperceptible. Our pre-load checker is simple and straightforward to implement directly from its definition, as so it is the injection of the runtime enforcer checks in the application's code. Finally, we do not provide a full implementation of the static analyzer, but present definitions on how to extend PCR analysis so that it can be integrated with the other steps of our approach.

For an overview of how the major contributions of this thesis are organized, refer to Table 1.1 on Section 1.6.

1.5 Related Work

Many of the initial papers on language based security [SM03b] enforced the non-interference property [GM82] statically using type-based [PS03, VIS96, BN02] or dataflow-analysis based [AB04, ABB06, BBM94, Den76] approaches. Banâtre, et al. [BBM94] were the first to propose using *accessibility graphs* to specify data and control flow dependencies between different variables in the program and thereby automatically inferring the security properties of the program. Bergeretti, et al. [BC85] represent information flows as relations between different variables in the program and Clark, et al. [CHH02] represent flows as relations between the variables and the control flow points represented by the program counter. Although the above approaches require dependency calculation similar to our expression graphs, we can additionally represent declassification policies, while they can only check for pure non-interference. More recently, Hammer, et al. [HKS06, HS09] propose an information flow control algorithm for Java. The variable dependencies are specified in the form of dependency graphs. The declassification policies are specified using path conditions, which are a conjunction of all the conditional expressions that are encountered before reaching the output program point. Although the path conditions are certainly useful to specify some kind of declassification policies, they do not compute what expressions are being declassified. Here, we attempt to capture this information using our expression graphs. Swamy, et al. [SH08] propose a formal language, AIR (Automata for Information Release), for describing stateful information release policies separately from the program that is to be secured. Although the policies are specified in the form of an automaton separate from the program, the approach requires that the programs be written in λAIR, a core formalism for a functional programming language, so that the AIR policies can be provably enforced.

Non-interference was found to be too restrictive to specify certain security properties and the use of declassification policies was proposed. One of the first papers to specify declassification policies was the paper by Myers[Mye99], where the declassification is based on principal authorization which falls under the *who* dimension. Several new declassification policies were proposed as discussed by Sabelfeld, et al. [SS05], each of which differed either in the type of declassification policies being handled, how the de-

classification policies are specified or by the enforcement mechanism.

In type-based approaches the declassification condition is tagged to the security lattice [LZ05, CM04, TZ05, CM08] or to an expression inside the program [SM03a]. Since declassification typically involves downgrading the security level from high to low, this is the right place to specify the policies. To specify which policy to use at the declassification points, new syntactic constructs are introduced into the programming language, making the policy and the program to be inter-dependent on each other. In most cases, a new *declass* command is introduced into the program. The enforcement is usually a hybrid of static analysis and dynamic execution. In some approaches [SHTZ06, BWW08], a particular section of code is encapsulated in a conditional statement. The condition specifies the declassification policy. This section of code is executed only if the condition is true, thereby dynamically enforcing declassification. More recently, some approaches advocate specifying a special security API [HKMH06, SH08, HKM05]. If the program is written using this API, declassification policies can be provably enforced. In [SPB09] authors present a λ-calculus based language for dynamic information flow tracking, that accepts more programs than type-based systems, at the cost of greater overhead. Their approach tracks information flow in multiple dimensions (i.e. it reasons over, e.g. the confidentiality of an integrity label), a goal out of the scope of this thesis. Even though variables have no static security labels, declassification is done explicitly in the code, by the programmer.

Li and Zdancewic [LZ05] use declassification policies that take the form of lambda terms over inputs, akin to our approach. Expressing the policies in lambda calculus gives them the flexibility to compare different policy terms for equivalence. This is a strength of the prior work in relation to our own. The main strength of our work in relation to theirs lies in our enforcement mechanism. For this, they use a type system that labels each variable in the program with a security policy. The security lattice is given over the lambda terms in the policy. As they also point out, their enforcement mechanism cannot handle policies such as $\lambda x : int.\lambda p : int.(x + p) * p$. On the other hand, our work handles this kind of situation, since our program expression graphs implicitly keep track of all the expressions that can flow to an output channel, enabling our approach to analyze expressions resulting from global computations. Thus, using program graphs allows us to enforce more expressive policies. The paper also hints that that their approach can be applied to untrusted code if enforced differently, but does not explain how to do so.

The type-based enforcement mechanism of delimited release [SM03a] and localized delimited release [AS07b] policies keep track of the variables involved the in the declassified expressions and ensure that they are not updated before declassification. This is required to prevent laundering of information. Our flow based enforcement automatically keeps track of the changes in the variables, thereby precluding the need to have an explicit declassification construct in the program.

Jif [CMVZ06] is one of the most advanced programming languages designed to enforce fine-grained declassification policies in the program. However, if the programs and policy are not carefully designed, as stated in [HKMH06], there is a risk of burying the policy deep inside the code and therefore requiring a change in the program with every change in the policy. In light of this observation, several researchers studied how large

programs can be written in a security typed language so that their behaviour is provably secure. Askarov, et al. [AS05] show how security typed languages can be used to implement cryptographic protocols and propose several design patterns to help the programmers to write their applications in Jif. They program a large poker application to demonstrate their approach. Hicks, et al. [HKMH06] propose FJifP, which includes all the security features of Jif and also an option to use certain methods as declassifiers. They also highlight the need for effective programming tools in which to write Jif programs.

Askarov, et al. [AS07a] provided the foundation for CGR with their definition of the Gradual Release (GR) property. Their paper quantifies the knowledge obtained by the observer as the set of possible secret inputs that could be generated by observing the public outputs, i.e., the notion of *observed knowledge*. The GR property states that the observer's knowledge increases only at declassification points. Our aim of supporting policies that are as program-independent as possible prevents our considering attacker models that involve program variables other than output channels. Thus the observed knowledge in our framework is the knowledge obtained from the outputs and does not depend on any other program events. The CGR property of [BNR08] requires the GR property. Additionally, it requires that the low-security observer of program behavior is able to detect no difference between runs that are generated from initial states that yield the same values for expressions identified in the declassification policies. Our formulations of revealed and observed knowledge follow a similar approach.

Banerjee, et al. [BNR08] achieve separation of code and declassification policies. However, their approach does not achieve a complete separation. Their declassification policies, named *flowspecs*, are a combination of a formula over program variables (P), special predicates called the agreement predicates (φ) over the program variables and a modifiable variable (x) whose type is being changed. The flowspecs are quite expressive and can be used to specify policies in *when*, *where* and *what* dimensions. However the technique only works for trusted code, which is written according to the policy specification. In their paper, if P and φ only have global variables, then they say that x can be a schematic variable instantiated with different local variables. Although this allows them to have more flexibility in terms of applying the same policy to different parts of a large code base, it does not allow them to use the policies for entirely different programs. The policies cannot be reused for any other code in which the data structures and global variable names differ. Our policy specifications are more general and can be applied to multiple, unrelated programs. Since their analysis uses the flow-insensitive, type-based approach, they require that programs disallow assigning new values to high variables prior to their use in expressions to be declassified. This means that programs need to be written in a policy-specific manner for them to be deemed valid, which is at odds with the application of their approach to legacy code. Our Policy Controlled Release (PCR) property is a variant of their Conditioned Gradual Release (CGR). Compared to the prior definition of CGR, ours is much simpler and more intuitive because it can be expressed purely on the observable behaviour of programs rather than needing details on program executions.

The notion of indistinguishability used in [AS09] is closely related to our D-equivalence relation (Section 2.4), as it is based on the attackers' knowledge of the initial values of high variables in their escape hatches, which resemble the declassifiable expressions

identified by policy in our framework. However, their expressions are identified individually, which prevents them declassifying expressions of unbounded size, such as result from iterative computations. They also do not share our objective of completely separating policy from program. This enables them to consider where declassification occurs within the program, and to handle attacker models in which non-output events are observable, which we inherently cannot do.

Giambiagi and Dam [GD04] provide a framework for analyzing a security protocol's implementation against its specification. A dependency specification defines an information flow property by characterizing the direct flow along a path in the form of allowed sequence of API and primitive function calls. However, as the authors mention in the paper, dependency specifications are very low-level objects, which can be used as intermediate representations of flow requirements. In general, their dependency specifications should accurately capture the exact number of times a method is called during a particular flow and it can only characterize a single flow. By contrast, our expression graph representation can represent several flow patterns, including loops. Also, as stated by the authors, their verification techniques are not yet fully automated, as opposed to ours.

Taint analysis [LL05, TPF+09] considers direct data flows, but, unlike information flow analysis, ignores control flows. In this sense, it is much less demanding than declassification policy enforcement. The input/output channels are labeled with a security level, such as tainted and un-tainted. A separate code analysis mechanism is required to check whether the sanitization routines (cf. declassification policies in our context) are present in the code. We, on the other hand, are associating the declassifiable expression with input and output channels, without considering how a program is written. This is certainly more expressive than a simple label. Also, a taint propagation through the program can be easily inferred by static analysis of code. Checking whether the sanitization routine is present in the code is a much more complex issue. In our work, we automatically infer whether a particular declassification policy can be applied to all possible program expressions that can be output by the program. This would correspond to checking whether all possible sanitization routines applied to inputs make them secure enough to be output. Therefore, our approach is much stronger than simple labelling of inputs and outputs in taint analysis.

Giacobazzi and Mastroeni [GM04] provide a powerful framework in which to specify the weakened variant of non-interference that is enforced under a declassification policy. We think its likely that our Policy Controlled Release property could be precisely stated in their framework, modulo the fact that our approach is communication channel-oriented, while theirs focuses on state transformation. We view our contribution as bringing the field closer to being able to implement a large class of practical analyses that can be specified in their framework. This prior work is highly abstract, and provides little guidance with respect to the construction of usable analysis tools. To achieve in their framework what our graph-based analysis achieves would require devising a representation of an abstract domain for each declassification policy. Each element of this domain would denote a set of valuations for high-security variables such that the declassifiable expressions each yield the same value, but the valuations are otherwise unconstrained. In other words, abstract domain elements differ only with respect to the values assumed by declassifiable

expressions. While it might be practical to represent these sets by using (infinite sets of) constraints, it is not at all clear how, for program C, one would compute the best correct approximation of $[\![C]\!]$ over them, as required by their approach.

Runtime enforcement mechanisms [LBW09a, CNC10, KKL+01] monitor accesses a program does during execution, enforcing access control policies. However, limited visibility of the program's code, coupled with a necessity to incur low overhead limit the types of policy that can be enforced. These mechanisms are often useful for enforcing access control, but not information flow, since the latter requires knowledge of non-executed code, in order to detect implicit flows. In [LR10] authors propose a theory for runtime enforcement, modelling runtime mechanisms that can transform results, and also an analysis of the policies that such model can enforce. Their abstract model is simple and expressive, and our runtime enforcement step can be fit in the model in a straightforward manner. The model, however, makes explicit one of the limitations of runtime enforcement: as it only considers actions performed by the application at runtime, it is unaware of implicit flows of information caused by actions that were *not* performed. Authors also do not study in detail the overhead caused by the monitor, since that varies for each implementation of their model, but point that this overhead may not be negligible. Finally, their model also supports result-sanitization policies (e.g. mask secret files from a directory listing), which are out of the scope of our approach. A recent study on policies enforceable by runtime monitoring is presented in [LBW09b] and the same authors present a framework for composing expressive runtime policies in [BLW09], but again policies are based on specific security-sensitive actions performed by the program. In [AF09] the authors propose a purely dynamic information flow analysis approach that handles implicit flows. However, this is achieved by disallowing, on the language semantics, dynamic label updates within high conditionals, an unnecessary limitation in our approach.

A hybrid approach had been proposed in [SMH01], although authors proposed the combination of inline reference monitors with static type systems. Our approach precludes the need of a type system. Concrete proposals of hybrid mechanisms are scarce, although have become increasingly popular [LG07, LGBJS07, SST07, NJK+07]. The authors of [YZLL11] integrate static analysis and runtime tracking to establish an approach to generate a sensitive data propagation graph aimed at incurring minimum time overhead on systems. All the cited approaches, however, do not support neither runtime security labels nor declassification policies. In [QDXW04] the authors use static analysis to detect which parts of the program satisfy the policy, and use a runtime enforcer to guarantee that unsafe parts are not executed. Thus, they do not enforce policies that need runtime information: the runtime enforcer serves only to select the parts of the code that may be executed.

In [SR10] the authors show that, by blocking execution of unsafe instructions, a dynamic monitor can guarantee termination-insensitive noninterference, for a flow-insensitive analysis. Then, in [RS10], the same authors prove impossibility of a sound purely dynamic information-flow monitor that accepts programs certified by a classical flow-sensitive static analysis. The authors demonstrate the need for hybrid mechanisms in flow-sensitive analysis, and present a general framework for such mechanisms. In both papers, however, authors do not consider neither declassifications nor dynamic labels.

Hybrid mechanisms that support declassification have been proposed in [AS09, CM08], however these approaches do not share the expressiveness of PCR analysis, being unable to declassify expressions of unbounded size (such as the average salary), do not share our goal of separating policy from program, and work on security typed languages, requiring the programmer to identify points where declassification occur.

A recent implementation-oriented approach is Resin [YWZK09], a language runtime that implements data-flow assertions. It is a fully runtime approach, incurring a non-negligible overhead (33% CPU overhead for their measured application). Besides, it does not share a number of our goals: it allows the programmer to specify application-level data flow assertions, as opposed to our goal of analyzing untrusted programs, and it does not aim for information flow control neither declassification policies.

1.6 Plan of the Thesis

In the next chapters we answer the research question by defining our proposed static analysis mechanism. In **Chapter 2** we define our expression-matching framework, which deems programs safe according to a set of declassifiable expressions. This framework is of theoretical nature, with many non-computable definitions. We first present some motivating examples in Section 2.1. Then, we define our programming language in Section 2.2. The core definitions of the framework are the ones that define program validity, presented in Section 2.3. We proceed to define our security property, called Policy Controlled Release, in Section 2.4. Finally, we demonstrate the soundness of the framework in Section 2.5.

In **Chapter 3** we define our graph-based implementation of the framework. This implementation is a tractable, safe approximation of the expression-matching framework. We start by revisiting the motivating examples in Section 3.1, showing how the implementation treats them. Then, we introduce our form of graphs, both for representing programs and policies, in Section 3.2. The core mechanism of the implementation is the matching between program and policy graphs, presented in Section 3.3. Then, in Section 3.4 we show the soundness of the implementation, demonstrating that it implies the program validity of the framework. In Section 3.5 we present and analyze algorithms for the implementation, in order to show its tractability. Finally, in Section 3.6 we extend both the language and the graphs in order to support user-defined functions, and show how both the framework and implementation can handle them, in order to demonstrate the suitability of our mechanism to more elaborate programming language constructs. Figure 1.2 below shows a roadmap of the core sections of chapters 2 and 3, presenting the elements of our mechanism and how they interact.

Our extension of the graph-based approach, a practical hybrid static-runtime enforcer, is presented in **Chapter 4**. We present three motivating examples in Section 4.1, all based on real mobile applications, and then outline the approach in Section 4.2, including some assumptions about the target execution system. Then we discuss a small modification on our graph-based PCR analysis in Section 4.3, and proceed to define the next steps of the hybrid enforcer: Section 4.4 presents the pre-load checker, and Section 4.5 the runtime

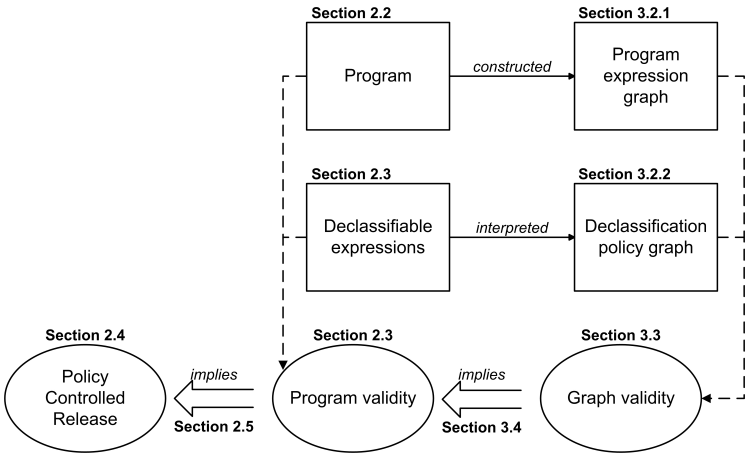


Figure 1.2: Roadmap of the core sections of chapters 2 and 3

enforcer, including an analysis on the overhead it causes on the target system.

Chapters 2, 3 and 4 form the core technical contributions of this thesis. Our contributions are both theoretical (i.e. definitions of program validity according to information flow and declassification policies and theorems deeming such definitions correct) and practical (i.e. implementable definitions of the enforcement mechanisms, algorithms and experimental results). Table 1.1 presents how the main contributions of this thesis are organized through these 3 chapters. Contents of chapters 2 and 3 are presented in papers [RBdH$^+$10, RBdH$^+$11], while contents of chapter 4 are presented in [RCEC11].

Contribution	**Theoretical**	**Practical**
Chapter 2 (Framework)		
Section 2.3 (Program Validity)	Definition 2.9	
Section 2.4 (PCR)	Definition 2.14	
Section 2.5 (Soundness)	Theorem 2.15	
Chapter 3 (Implementation)		
Section 3.2 (Expression Graphs)	Theorem 3.6	Definition 3.1
Section 3.3 (Graph Matching)	Theorems 3.13 and 3.15	Definitions 3.9 and 3.12
Section 3.4 (Soundness)	Definitions 3.16 and 3.17, Theorem 3.21	
Section 3.5 (Algorithms)		Algorithms 3.1, 3.2 and 3.3
Chapter 4 (Extension)		
Section 4.3 (Static Analyzer)		Definition 4.1
Section 4.4 (Pre-load Checker)		Definition 4.2 and Algorithm 4.4
Section 4.5 (Runtime Enforcer)		Figure 4.4 and Table 4.8

Table 1.1: Main theoretical and practical contributions of this thesis

Since in this thesis we present a new approach for information flow and declassification analysis, a number of new research directions arise. In **Chapter 5** we present a

number of open problems. Section 5.1 discusses how precisely the graph-based implementation approximates the expression-matching framework, and how it can be made more precise. Two open problems on the implementation are then presented: the loop-counting problem is presented in Section 5.2 and the algebraic equivalence one in Section 5.3. These two problems are presented along with a discussion on possible research paths to solve them. We then discuss how to achieve an implementation of graph-based PCR on a real programming language (e.g. Java, C++) in Section 5.4. Finally, we discuss in Section 5.5 what we consider to be the greatest long-term open problem left by this research field, and possibly a new research question still to be answered: can this kind of analysis be done in compiled code, such as assembly-level code?

We present some concluding remarks in **Chapter 6**. Finally, we present proofs for theorems and lemmas in **Appendix A**.

CHAPTER 2

Expression-Matching Framework

In this chapter we formalize the notion of a program satisfying a declassification policy. For this, we introduce our expression-matching framework, which defines program validity according to a set of "declassifiable expressions". Elements of such set represent expressions on secret input channels which are allowed to be declassified. For instance, values from an input channel α might be considered to be secret, unless they satisfy the declassifiable expression $\alpha \bmod 2$, meaning that the parity of these values can be declassified, i.e. assume a lower security level.

The theoretical framework defines a mechanism to determine every possible expression on inputs that a program can possibly reveal. Also, expressions which are described by a declassification policy are recognized and identified. Variables that hold only such expressions are then marked as "safe" variables. To simplify the discussion, we consider that every input channel in the program has a high security level (i.e., is a private channel), whereas every output channel has a low one (i.e., is a public channel). Thus, all flows of information must be authorized by a declassification policy.

In the following sections we first present some motivating examples for the framework (2.1), then we introduce the considered language syntax and semantics in Section 2.2. After that, in Section 2.3 we present the core definitions of the framework, stating program validity according to a policy. We then proceed to introduce the security property we wish to enforce, called Policy Controlled Release (PCR), in Section 2.4. Finally, in Section 2.5, we demonstrate the soundness of the framework, i.e. that a program deemed secure by it satisfies PCR.

The contents of this chapter are presented in papers [RBdH+10, RBdH+11].

2.1 Motivating Examples

In this section we illustrate by means of three examples the mechanism of our framework. The first example refers to one of the classical situations requiring declassification: authentication and password matching. The basic security requirement is that user information should not flow to the output channel, with one exception (captured by the declassification policy): boolean queries on the user's record may be declassified. Now, in order to authenticate the user, 3 methods are possible. If the user's record is "complete" and the user has a given credential, a function named *validate* can check this credential. This is the preferred method for authentication. If, however, the user does not have the required credential, but his record is complete, then the same *validate* function can be

applied over the user's last name, validating the user's name against a list. Finally, if the user's record is not complete, then the system prompts for a password, from another input channel, and uses the function *verify* to check it along with the user login name. In the end, the result of the authentication is sent to the output channel. We use this elaborate mechanism to outline different flows of execution that a program can take. The example program is given below. The language it uses is a standard imperative programming language, with no special security constructs, which will be used throughout this thesis. The inputs and outputs to the program are specified using input and output channels and represented with Greek letters, as further explained in Section 2.2. Channel α returns the record with the user information, channel β is used to retrieve a password from the user, if necessary, and channel γ is the output channel to where authorization information is sent.

Example 2.1. *Authentication program:*

```
struct x := α;
string f;
bool v;

if iscomplete(x) then
    if hascred(x) then
        f := credential(x);
    else
        f := lastname(x);
    v := validate(f);
else
    f := login(x);
    string y := β;
    v := verify(f, y);
γ := v;
```

Pre-processing and conversion to SSA. Our analysis works on code that has already been pre-processed in the following way: (1) operators are translated into functions (e.g., $a + b$ becomes $add(a, b)$), (2) only one function is allowed per assignment, i.e., assignments of complex expressions are broken into several assignments, (3) conditions on control-flow commands (`if` and `while`) refer to a single boolean variable.

We also convert a program into the Static Single Assignment (SSA) format using standard methods [CFR+91]. SSA is a known intermediate representation form for programs, in which every variable is assigned exactly once. Variables being assigned more than once are renamed (with a different name for each assignment: typically the original name with a subscript). For variables that are modified in the body of branching statements (e.g. conditionals and loops), the translation algorithm generates a new variable name at the *join points* (at the end of the conditional or the loop). Moreover, a new function ϕ is introduced, which takes as input the variable values from all the branches, and outputs the value from the branch that was taken. During the translation, we additionally annotate the ϕ function with the conditional variable of the branch to which the ϕ function is

associated. The technique for computing SSA form of a program has been proved to be tractable. For more information on it refer to [CFR+91, BP03].

Example 2.2. *Authentication program in SSA format:*

```
struct x_1 := α;
bool c_1 := iscomplete(x_1);
string f_0;
bool v_0;

depends(β, c_1);
if c_1 then
    c_2 := hascred(x_1);
    if c_2 then
        f_1 := credential(x_1);
    else
        f_2 := lastname(x_1);
    f_3 := φ_{c_2}(f_1, f_2);
    v_1 := validate(f_3);
else
    f_4 := login(x_1);
    string y_1 := β;
    v_2 := verify(f_4, y_1);
v_3 := φ_{c_1}(v_1, v_2);
f_5 := φ_{c_1}(f_3, f_4);
γ := v_3;
```

Note that the conditions are syntactically associated with the ϕ-functions. Also, the $depends$ command is generated during the pre-processing and serves the purpose of making the control dependence between channel β and variable c_1 explicit, since the input occurs inside the conditional. This will be further explained in the next section.

As mentioned, the declassification policy allows the release of boolean queries over the user's records. This policy is represented by a set D of *declassifiable expressions*, containing expressions on inputs that are allowed to be made public. For this example, we have $D = \{hascred(\alpha), iscomplete(\alpha), validate(credential(\alpha)), validate(lastname(\alpha)), verify(login(\alpha), \beta)\}$. All the notation used in the example will be made precise later on this chapter.

Policy Matching. Now that we have both the program and the policy, we can check if the program is *safe*. In our program the (low) output γ is assigned the value of variable v_3, so what we now have to check is whether every expression possibly held by v_3 is *public*, i.e. it is either in set D or a function over one or more expressions in D (note that we don't consider a declassifiable expression to be invertible—this is discussed further ahead). This analysis needs to consider both data and control dependencies of variables. Note that v_3 can assume either the value of v_1 or v_2. In turn, v_2 holds the expression $verify(login(\alpha), \beta)$ and v_1 holds $validate(f_3)$, where f_3 can hold either $credential(\alpha)$ or

$lastname(\alpha)$. Thus, the union of all these expressions form the set of *data dependencies* of variable v_3. In other words, it is the set of every possible expression that variable v_3 can hold. Also, the choice of which of these expressions will be held by v_3 depends on the branches taken by both `if` commands, which in turn depend on the values of variables c_1 and c_2. Thus, the expressions possibly held by these variables ($iscomplete(\alpha)$ and $hascred(\alpha)$, respectively) form the set of *control dependencies* of v_3. That is, these are expressions whose values might be revealed by observing the value of v_3. For v_3 to be marked as *safe*, all expressions within both its data and control dependencies must be public. However, it is clear that each of the aforementioned expressions is in fact an element of D. Thus, we say that variable v_3 is safe, and its value can be output to channel γ. Since this is the only output operation in the code, the program is deemed valid.

Statistic Calculation. We now provide a second example, that involves a policy which allows the declassification of expressions in a given recursive pattern, represented in the code by a looping structure. This example is inspired by another classical need for declassification: statistical calculations on secure data (where high data should not be released but statistics on it may be declassified). The program, given below (already preprocessed), calculates the average of the entries in a given data structure. Channel α returns the next element of a sequence of salaries of an organization. The code below fetches all the salaries from the structure, calculates their average, and then sends the result to output channel γ.

Example 2.3. *Average calculation program in SSA format:*

```
int a1 := 0;
int i1 := 0;
int l1 := length(α);
bool c1 := leq(i1, l1);

while (c3 := φc3(c1, c2);
       a3 := φc3(a1, a2);
       i3 := φc3(i1, i2);
       c3) do
    int t1 := α;
    a2 := add(a3, t1);
    i2 := add(i3, 1);
    c2 := leq(i2, l1);

a4 := div(a3, l1);
γ := a4;
```

Note that the ϕ-functions are placed along with the loop condition and the program semantics would require that the ϕ assignment be executed even if the loop is not taken, but also once after each iteration [BM94]. With this, the ϕ functions in the `while` header work as the following: in the first iteration variable i_3 is assigned the value of i_1, since at this point i_2 has not yet been defined. On subsequent iterations i_3 is then assigned the

value of i_2, which is then the most recently defined of the arguments of the ϕ function. The same reasoning applies to variables a and c.

We first have to determine every expression on inputs that can be output by this program. Since variable a_4 is the one sent to output, it is the one that needs to be checked. Here, we use $E(a_4)$ to denote every possible expression held by variable a_4, in every possible execution of this program. By observing the program code, we have that $E(a_4) = \{0, \alpha_1, \frac{\alpha_1+\alpha_2}{2}, \frac{\alpha_1+\alpha_2+\alpha_3}{3}, \ldots\}$, where α_i represents the value obtained from the i-th access on input channel α. Also, we know that a_4 has a control dependency with the loop conditional. Being $PC(a_4)$ the set of expressions which variable a_4 has control dependencies with, in every possible execution, we have from the code that $PC(a_4) = \{0 \leq length(\alpha), 1 \leq length(\alpha), \ldots\}$. Note that these are all the boolean expressions that might be checked on the loop conditional. With this, the framework deems the program secure if all of these expressions are authorized by a declassification policy. Here, the set of declassifiable expressions is defined as: $D = \{length(\alpha), \alpha_1, \alpha_1 + \alpha_2, \alpha_1 + \alpha_2 + \alpha_3, \ldots\}$. Thus, this policy allows the declassification of $length(\alpha)$ plus any sum of distinct values from α (note that there are no bounds for the minimum number of values in the sum, this is discussed further in the rest of this thesis). Now, for the program to be validated, every member of both $E(a_4)$ and $PC(a_4)$ must be a public expression. We can see that every member of $E(a_4)$ (e.g., $\frac{\alpha_1+\alpha_2}{2}$) is a fraction of two public expressions: an element of D ($\alpha_1 + \alpha_2$) and a constant value (2). Conversely, every member of $PC(a_4)$ is a comparison between a numerical constant and $length(\alpha)$, which is in D. Thus, the framework deems the program secure.

Encryption. We now present another example, showing the use of a policy that allows release of an expression over *any* input channel. It is another of the classical examples of declassification, this time in presence of encryption: we have data that is sensitive if unencrypted, but its encrypted version can be declassified. The code below is already pre-processed: the input channel α provides a sensitive plain text file, β represents a cryptographic key. Output channel γ represents a low output.

Example 2.4. *Encryption program:*

`text` $x_1 := \alpha$;
`int` $k_1 := \beta$;
$x_2 := enc(x_1, k_1)$;
$\gamma := x_2$;

The unique aspect of this example is that here we want a policy that allows the encryption of any input value to be declassifiable. Thus we have that $D = \{enc(i, \beta) \mid i \in \mathbf{In}\}$, where **In** is the domain of all input channels. Thus, expression $enc(\alpha, \beta)$ held by variable x_2 is public, and can be declassified. Note that this classic example of the need for declassification is handled trivially by our theoretical framework.

As mentioned in our first example, we don't consider a declassification to be invertible. For this example, one may think that, after x_2 has been marked as safe, a decryption function could be used to retrieve the original α value and assign it to a new x_3 variable. However, since the decryption function would need the decryption key, the inheritance

from x_2 would not be sufficient for x_3 to be marked as safe. A dependence from input β to the decryption function would also need to be validated and this validation would not happen, as there is no policy that allows it, therefore making x_3 insecure.

2.2 Language: Syntax and Semantics

In this section we introduce the syntax and semantics of our language. **Var** is a set of variables, x, y, z, a, b, c, range over **Var** and may have subscripts; c is usually a boolean variable. Additional **IO** variables (**IO** $=$ **In** $\cup$ **Out**) represent input/output channels. Note that **Var** and **IO** are disjoint sets. We use α, β to denote input channels, γ, δ to denote output channels, and θ to range over all of **IO**. We additionally use ρ to range over **Var** $\cup$ **IO**. Input channels are regarded as streams of values and are indexed to indicate specific input values; e.g., α_n denotes the n-th input value of input channel α.

Functions are defined the usual way. Constants are functions of arity 0, and we use N to denote them. Expressions are obtained by combining functions, variables (also **IO**) and constants in the usual way.

We use a simple imperative language with assignment, conditionals and loops, already translated to SSA form. To simplify the presentation, we assume that all operators are applied using prefix notation (e.g., writing $add(a, b)$ instead of $a + b$), with at most one function per assignment (no nesting); also, expressions on conditionals refer to a single boolean variable. Any program can be translated to this format in a straightforward manner. Regarding the SSA translation, ϕ-functions always have the form $x := \phi_c(a, b)$, where c is the conditional variable that generated that ϕ-function. In `while` expressions, $\overline{C}$ represents the ϕ-functions added by the SSA translation, which are evaluated once if the loop is not taken, and at every iteration otherwise.

Definition 2.1 (Program)**.** *A program $C \in Prog$ is defined by the following syntax:*

$$C ::= \texttt{skip} \mid x := \alpha \mid \gamma := x \mid x := f(y_1, \ldots, y_k) \mid x := \phi_c(a, b) \mid C_1 \,;\, C_2 \\ \mid depends(\theta, c) \mid \texttt{if}\ c\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{while}\ \overline{C}\ ;\ c\ \texttt{do}\ C$$

The command $depends(\theta, c)$ is a special command that helps our non-standard semantics keep track of control dependence on I/O channels. It is added to the program during pre-processing: $depends(\theta, c)$ is inserted every time an input or output operation occurs inside a conditional, and relates the channel with the conditional under which it occurs. The command is added just before the conditional block in which the operation takes place.

We say that C is a *compositional statement* if C is of the form $C_1; C_2$, otherwise C is *non-compositional*. Note that any program can be written in the form $C_1; \ldots; C_n$ ($n \geq 1$) with C_i non-compositional statements. Here we call the first non-compositional statement C_1 the active command of C, denoted $head(C)$.

Next, we define the program semantics, starting from the notion of state. Note that we present an instrumented semantics, in the sense that the state of the process keeps track of certain information useful for proving the compliance of our validation mechanism.

Definition 2.2 (State)**.** *A* state $\sigma \in \Sigma$ *is a 4-tuple* $\langle E, I, O, PC \rangle$*, where:*

$$\begin{array}{rcrcl} E & \in & \mathcal{E} & = & \mathbf{Var} \to \mathbf{Exp}\langle \mathbf{In} \times \mathbb{N} \rangle \\ I & \in & \mathcal{I} & = & \mathbf{In} \to \mathbb{N} \\ O & \in & \mathcal{O} & = & \mathbf{Out} \to \wp(\mathbf{Exp}\langle \mathbf{In} \times \mathbb{N} \rangle) \\ PC & \in & \mathcal{PC} & = & (\mathbf{Var} \cup \mathbf{IO}) \to \wp(\mathbf{Exp}\langle \mathbf{In} \times \mathbb{N} \rangle) \end{array}$$

E is a mapping from variables to expressions on indexed input channels, keeping track of the expression over the input that a variable holds. I is a mapping from input channels to numeric indexes, keeping track of the index of the next value to be read (so $I(\alpha)$ denotes index of the next value to be read from channel α); initially, $I(\alpha) = 1$ for every input channel α. O maps each output channel to the set of expressions (on indexed inputs) that could be sent over that channel. Finally, PC maps variables and channels (both input and output) to sets of expressions on indexed inputs, which record the implicit information flows, i.e. the expressions on which the variables and channels are conditionally dependent. Given a state σ, we write E_σ to indicate its first component, I_σ for the second, etc. We omit σ if it is clear from the context, thus e.g. $E(x)$ denotes the expression held by x in the "current" state.

Next, we define *environments* which provide the input to the program through the channels. We have a straightforward channel model where the channels are independent of each other. In Chapter 6 we discuss how to extend this to more elaborate channel models.

Definition 2.3 (Environment)**.** *An environment* $\pi \in \Pi$ *is a mapping* $\mathbf{In} \times \mathbb{N} \to \mathbf{Val}$ *from input channels and indexes to values. Value* $\pi(\alpha, i)$ *represents the value returned from the* i*-th access on input channel* α*.*

Finally, we define a configuration over which the semantics are defined.

Definition 2.4 (Configuration)**.** *A configuration* $\omega \in \Omega$ *is a triple* $\langle C, \sigma, \pi \rangle$*, where* C *is a program,* σ *a state and* π *an environment.*

Note that the environment determines the inputs that have been or will be provided to the program and (due to our channel model) does not change during the execution of the program. The operational semantics is presented in Figure 2.1. The transitions between configurations have a label ($\in$ **Obs**) representing what can be observed externally when that transition occurs; a τ label represents a non-observable transition. In our case, the only observable action is the output, showing the channel and the value being sent over the channel. I.e. $o \in \mathbf{Obs}$ is τ or $out(\gamma, v)$ for some output channel γ and value v.

We write $f[x \overset{\odot}{\leftarrow} n]$ for a variant of f where the value assigned to x is $f(x) \odot n$. Here $\odot$ can be any operator of the right type. For instance, consider a function f and a variable x such that $f(x) = 1$. If $f' = f[x \overset{+}{\leftarrow} 2]$, then we have that $f'(x) = f(x) + 2 = 3$, and $f'(y) = f(y)$ for every $y \neq x$. We omit the operator if it is projection on the second argument, i.e. $f[x \leftarrow n](x) = n$. For changes in the state, we only indicate the components for which σ' differs from σ. Our semantics treats ϕ-functions in a special way. Unlike the standard functions, ϕ-functions are evaluated as soon as they appear.

$$\langle x := \alpha, \sigma, \pi\rangle \xrightarrow{\tau} \langle \texttt{skip}, \sigma', \pi\rangle \qquad \textit{(Input)}$$
$$\text{where} \quad E_{\sigma'} = E_\sigma[x \leftarrow \alpha_{I_\sigma(\alpha)}]$$
$$I_{\sigma'} = I_\sigma[\alpha \xleftarrow{+} 1]$$
$$PC_{\sigma'} = PC_\sigma[x \leftarrow PC_\sigma(\alpha)]$$

$$\langle \gamma := x, \sigma, \pi\rangle \xrightarrow{o} \langle \texttt{skip}, \sigma', \pi\rangle \qquad \textit{(Output)}$$
$$\text{where} \quad o = out(\gamma, V(E_\sigma(x), \pi))$$
$$O_{\sigma'} = O_\sigma[\gamma \xleftarrow{\cup} E_\sigma(x)]$$
$$PC_{\sigma'} = PC_\sigma[\gamma \xleftarrow{\cup} PC_\sigma(x)]$$

$$\langle x := f(y_1, \ldots, y_k), \sigma, \pi\rangle \xrightarrow{\tau} \langle \texttt{skip}, \sigma', \pi\rangle \qquad \textit{(Assign)}$$
$$\text{where} \quad E_{\sigma'} = E_\sigma[x \leftarrow f(E_\sigma(y_1), \ldots, E_\sigma(y_k))]$$
$$PC_{\sigma'} = PC_\sigma[x \leftarrow PC_\sigma(y_1) \cup \ldots \cup PC_\sigma(y_k)]$$

$$\langle x := \phi_c(a, b), \sigma, \pi\rangle \xrightarrow{\tau} \langle \texttt{skip}, \sigma', \pi\rangle \qquad \textit{(Phi)}$$
$$\text{where} \quad E_{\sigma'} = E_\sigma[x \leftarrow EV(\phi_c(a, b), \sigma)]$$
$$PC_{\sigma'} = PC_\sigma[x \leftarrow E_\sigma(c) \cup PC_\sigma(c) \cup PC_\sigma(a) \cup PC_\sigma(b)]$$

$$\langle \textit{depends}(\theta, c), \sigma, \pi\rangle \xrightarrow{\tau} \langle \texttt{skip}, \sigma', \pi\rangle \qquad \textit{(Depends)}$$
$$\text{where} \quad PC_{\sigma'} = PC_\sigma[\theta \xleftarrow{\cup} E_\sigma(c) \cup PC_\sigma(c)]$$

$$\langle \texttt{if } c \texttt{ then } C_1 \texttt{ else } C_2, \sigma, \pi\rangle \xrightarrow{\tau} \langle C_1, \sigma, \pi\rangle \quad \text{if } V(E(c), \pi) = \texttt{true} \qquad \textit{(If 1)}$$
$$\xrightarrow{\tau} \langle C_2, \sigma, \pi\rangle \quad \text{if } V(E(c), \pi) = \texttt{false} \qquad \textit{(If 2)}$$

$$\langle \texttt{while } \overline{C} \texttt{ ; } c \texttt{ do } C, \sigma, \pi\rangle \xrightarrow{\tau} \langle \overline{C}, \sigma, \pi\rangle \quad \text{if } V(E(c), \pi) = \texttt{false} \qquad \textit{(While 1)}$$
$$\xrightarrow{\tau} \langle \overline{C} \texttt{ ; } C \texttt{ ; while } \overline{C} \texttt{ ; } c \texttt{ do } C, \sigma, \pi\rangle$$
$$\text{if } V(E(c), \pi) = \texttt{true} \qquad \textit{(While 2)}$$

$$\langle \texttt{skip ; } C, \sigma, \pi\rangle \xrightarrow{\tau} \langle C, \sigma, \pi\rangle \qquad \textit{(Skip)}$$

$$\frac{\langle C_1, \sigma, \pi\rangle \xrightarrow{o} \langle C_1', \sigma', \pi\rangle}{\langle C_1 \texttt{ ; } C_2, \sigma, \pi\rangle \xrightarrow{o} \langle C_1' \texttt{ ; } C_2, \sigma', \pi\rangle} \qquad \textit{(Seq)}$$

$$EV : \mathbf{Exp}{<}\mathbf{Var}{>} \times \Sigma \to \mathbf{Exp}{<}\mathbf{In} \times \mathbb{N}{>}$$
$$EV(\phi_c(a, b), \sigma) = \begin{cases} E(a) & \text{if } a \text{ has been most recently defined;} \\ E(b) & \text{if } b \text{ has been most recently defined.} \end{cases}$$
$$V : \mathbf{Exp}{<}\mathbf{In} \times \mathbb{N}{>} \times \Pi \to \mathbf{Val}$$
$$V(e, \pi) = \begin{cases} \pi(\alpha, n) & \text{if } e = \alpha_n; \\ \mathbf{f}(V(e_1, \pi), \ldots, V(e_n, \pi)) & \text{if } e = f(e_1, \ldots, e_n). \end{cases}$$

Figure 2.1: Program semantics

Function EV makes this evaluation. According to standard definition of the ϕ-functions in SSA form, the function returns the variable that has been defined most recently based on the branch taken. While f is a syntactic object, the boldface **f** used in function V indicates the semantic function that is actually evaluated to a value.

The semantics are standard small-step semantics, instrumented to work with the expression tracking components of the program state. This way, variables hold expressions on inputs rather than values, and these expressions are evaluated to their actual values only when needed. This is done by function V, which uses the environment to evaluate inputs to their values.

- Rule *Input* updates the expression held by a variable x (i.e. $E(x)$) to α_i, α being the accessed input channel and i its current access sequence number (i.e. $I(\alpha)$). Thus, α_i represents the i-th value from α. The current sequence number for the

input channel is also incremented and x inherits the set of conditional expressions related to α (i.e. $PC(\alpha)$ is assigned to $PC(x)$).

- Rule *Output* is the only that produces an observable transition: $out(\gamma, V(E_\sigma(x), \pi))$ represents an observable output to channel γ with the value evaluated from the expression currently held by variable x. This expression is also added to the set of expressions output by channel γ (i.e. $O(\gamma)$). Finally, the set of conditional expressions on x is included on the set of conditional expressions on γ: the set $PC(x)$ is included in $PC(\gamma)$.

- Rule *Assign* updates the expression held by x to the expression on the right-hand side of the assignment, translating any variables (y_k) to the corresponding expressions held by them ($E(y_k)$). Additionally, the set of conditional expressions on x is updated with the union of such sets relative to all variables in the right-hand side.

- Rule *Phi* handles assignments with ϕ functions. The expression held by x is updated with the expression held by either a or b, depending on which of them has been most recently defined. This logic is defined by function EV. The set of conditional expressions on x is updated with the union of the expression held by c and the sets of conditional expressions of all variables on the right-hand side: c, a and b. $E(c)$ is added because c is the conditional variable of the `if` or `while` command related to the current ϕ assignment. Also, both $PC(a)$ and $PC(b)$ are included since in loops both a and b are eventually assigned to x. In non executed loops and `if` commands only one of a or b is assigned to x, but the other is never defined, so its PC is empty anyway.

- Rule *Depends* handles the special $depends$ command. As the purpose of the command is to state the dependency of an I/O channel to a conditional variable, both the expression held by c and its related set of conditional expressions are included in the set of conditional expressions of I/O channel θ.

- Rules *If* and *While* handle their homonymous commands in a standard way: evaluating the boolean expression on the conditional variable and executing the branch corresponding to its value. These rules don't need to deal with dependencies created by the `if` and `while` structures, since this is done by the ϕ and $depends$ commands related to them.

- Finally, rules *Skip* and *Seq* trivially handle the `skip` command and a sequence of commands, respectively.

The *initial state* σ_{init} is the state in which no channels have been read yet ($I(\alpha) = 1$), all variables are undefined ($E(x) = \perp$) and no output has been written to any channel ($O(\gamma) = \emptyset$). A *run* of program C in environment π is a sequence of configurations, starting from the initial configuration and linked by transitions, i.e., $t \in (\mathbf{Obs} \times \Omega)^\infty$ in which for $t = \langle o_0, \omega_0 \rangle . \langle o_1, \omega_1 \rangle \ldots \langle o_n, \omega_n \rangle$, $o_0 = \tau$, $\omega_0 = \langle C, \sigma_{init}, \pi \rangle$, and for each i, such that $0 \leq i < n$, $\omega_i \xrightarrow{o_{i+1}} \omega_{i+1}$ is a transition given by the semantics (Figure 2.1).

We say the run is a *full run* if no steps are possible from end state ω_n otherwise the run is called a *prerun*. We write $o(t)$ for the sequence of (visible) output actions $o_0.o_1.o_2\ldots$ taken in t and $t \equiv_{\text{out}} t'$ if $o(t) = o(t')$. Similarly, we write $\omega(t)$ for the sequence of configurations $\omega_0.\omega_1.\omega_2\ldots$ taken in t. For $i \in \mathbb{N}$, $t_i = \langle o_i, \omega_i\rangle$, $o(t_i) = o_i$, and $\omega(t_i) = \omega_i$. For sets of traces T, T' we write $T \equiv_{\text{out}} T'$ if $\forall t \in T : \exists t' \in T' : t \equiv_{\text{out}} t'$ and vice versa. Finally, we also write $Run(C, \pi)$ for all the runs of C in environment π (note that for each prerun in t there is exactly one run t' which extends t with one step).

2.3 Program Validity

In this section we define program validity with regards to a set of declassifiable expressions. The definitions presented here form our *framework* for analysis of programs by matching the possible expressions held by variables with expressions allowed to be declassified. Note that these definitions include intractable computations. In the next chapter we present our specific *implementation* of the framework, which safely approximates it, providing a tractable analysis mechanism.

First, we assume that we are given a set of expressions that are declassifiable. These are expressions over indexed inputs which are allowed to be declassified.

Definition 2.5 (Declassifiable Expressions)**.** *Declassifiable expressions are the set of expressions, $D \subseteq \mathbf{Exp}\langle \mathbf{In} \times \mathbb{N}\rangle$, that are specified or defined to be declassifiable. These are expressions on the indexed input channels and constants.*

In the rest of this thesis we refer to the fixed yet unspecified set of declassifiable expressions D. Given these expressions, we would like to inductively define the set of public expressions, i.e., expressions whose values are safe to be output.

Definition 2.6 (Public Expressions)**.** *Let D be a set of declassifiable expressions, we say that the expression e is public according to D if the following relation holds:*
$public(e, D) \equiv (e \in D) \vee (e = f(e_1, \ldots, e_n) \wedge public(e_1, D) \wedge \ldots \wedge public(e_n, D))$

With the format of the declassifiable expressions defined, we now need definitions regarding program validity. First, we define every possible state a program C can reach: $states(C) = \{\sigma \mid \langle C', \sigma, \pi'\rangle \in \omega(t), t \in Run(C, \pi), \pi \in \Pi\}$. In other words, $states(C)$ represents every state that C can achieve for every possible execution of it in every possible environment.

We are now going to define "safety" of variables and I/O channels of a given program C, according to a set of declassifiable expressions D.

Definition 2.7 (Variable Validity)**.** *Let C be a program, ρ be a variable or I/O channel used in C and D be a set of declassifiable expressions. About ρ, we say that:*

1. *ρ is data dependency safe (DDS) if every expression possibly held by it is public (declassifiable).*

$$dds(\rho, C, D) \equiv \forall \sigma \in states(C) : \begin{cases} public(E_\sigma(\rho), D) & \text{if } \rho \in \mathbf{Var} \\ \forall e \in O_\sigma(\rho) : public(e, D) & \text{if } \rho \in \mathbf{Out} \\ public(\rho, D) & \text{if } \rho \in \mathbf{In} \end{cases}$$

2. *ρ is control dependency safe (CDS) if every expression that its value might depend on is allowed to be declassified.*

$$cds(\rho, C, D) \equiv \forall \sigma \in states(C), e \in PC_\sigma(\rho) : public(e, D)$$

3. *ρ is a safe variable (or I/O channel) if it is both DDS and CDS.*

$$safe(\rho, C, D) \equiv dds(\rho, C, D) \wedge cds(\rho, C, D)$$

Note that dds returns false for every ρ which is an input channel, unless there is a declassifiable expression that states that the whole channel can be made public.

Proposition 2.8. *Let x be a variable in program C and part of an assignment of the form $x := e$, with e of the form α, $f(y_1, \ldots, y_k)$ or $\phi_c(a, b)$, and D be a set of declassifiable expressions. We have that if $cds(x, C, D)$ then $\forall \rho \in e : cds(\rho, C, D)$. In other words, if x is CDS with respect to D, then every variable or I/O channel in e is also CDS.*

Finally, we can define program validity with respect to a set of declassifiable expressions.

Definition 2.9 (Program Validity)**.** *We say that a terminating program C is valid with respect to a set of declassifiable expressions D if every output channel γ accessed in C is safe. That is:*

$$valid(C, D) \equiv \forall \gamma \in \textbf{Out} : safe(\gamma, C, D)$$

We illustrate the concepts presented here by referring to Example 2.3. From the example, we recall that $D = \{length(\alpha), \alpha_1, \alpha_1 + \alpha_2, \alpha_1 + \alpha_2 + \alpha_3, \ldots\}$. Also, for every possible state σ that the program can reach, the expressions in $O_\sigma(\gamma)$ can be any from the set $\{0, \alpha_1, \frac{\alpha_1+\alpha_2}{2}, \frac{\alpha_1+\alpha_2+\alpha_3}{3}, \ldots\}$. With this, by Definitions 2.6 and 2.7 and by the content of D we can see that $public(e, D)$ holds for every $e \in O_\sigma(\gamma)$ and for every σ the program can reach, and thus $dds(\gamma, C, D)$ holds. Also, we know that $PC_\sigma(\gamma)$ can contain the expressions $\{0 \leq length(\alpha), 1 \leq length(\alpha), \ldots\}$. Thus, from the same definitions and D we have that $public(e, D)$ holds also for every $e \in PC_\sigma(\gamma)$ and thus $cds(\gamma, C, D)$ holds. Consequently, $safe(\gamma, C, D)$ holds, and since γ is the only output channel in the example, we also have $valid(C, D)$.

With Definition 2.9 we conclude the presentation of our expression-matching framework. Further in this thesis we demonstrate that the framework implies the property called Policy Controlled Release. The property is presented in the following section (2.4) and we show that the framework implies it in Section 2.5. In Chapter 3 we present a computable implementation of the framework presented so far, realized through the use of graphs. Refer to Figure 1.2 for a roadmap of the structure of the thesis.

2.4 Policy Controlled Release

In this section we define our reference security property called *Policy Controlled Release*. It is an "end-to-end" property in the sense that it bounds the knowledge that an attacker can gain by observing information released on output channels during any collection of runs. Our property closely follows the Conditional Gradual Release (CGR) given by Banerjee et al. [BNR08], though our variant differs from the original definition in several important respects, being simpler and independent of characteristics of the program's execution. CGR itself is a variant of the Gradual Release [AS07a] property. To simplify the discussion, we assume that information obtained from all the input channels is confidential and can be modified only by the target machine (on which the program runs). Reading from an input channel is not visible to an outsider. On the other hand, any information placed on the output channels is regarded as public. Releasing information from the secret input channels to the public output channels is permitted only according to declassification policies. Recall that we have also assumed that the input channels are non-interactive in the sense that reading data from one input channel, has no effect on the values obtained from other input channels. We discuss the relaxation of these assumptions in Chapter 6. The security property is defined in terms of the program only.

Two environments are said to be D-Equivalent if the values of the declassifiable expressions are the same in both the environments. Evaluating the expressions in the D set (see V in Section 2.2, Figure 2.1) gives the actual values that can be declassified.

Definition 2.10 (D-Equivalent Environments ($\approx_D$))**.** *Given a set of declassifiable expressions D, two environments π_1 and π_2 are said to be* D-equivalent, *$\pi_1 \approx_D \pi_2$, if $\forall e \in D : V(e, \pi_1) = V(e, \pi_2)$.*

Note that the fact that the expressions have the same values does not necessarily mean that the input values are the same in both environments. For example, the boolean expression $\alpha_1 > 0$ will have the same value for all the environments in which the first value obtained from channel α is larger than zero.

Lemma 2.11. *Let D be a set of declassifiable expressions, π_1 and π_2 be two environments such that $\pi_1 \approx_D \pi_2$, and e be an expression in* **Exp**$\langle$**In** $\times \mathbb{N}\rangle$*. If $public(e, D)$, then $V(e, \pi_1) = V(e, \pi_2)$.*

By observing the value of declassifiable expressions, one can learn something about the actual environment. In particular one learns that it must belong to a given class of D-equivalent environments. The expressions in D are correctly enforced if no further information can be learned.

Definition 2.12 (Revealed Knowledge ($\mathcal{R}$))**.** *Given a set of declassifiable expressions D and an environment π we define the knowledge of π revealed by D as:*

$$\mathcal{R}(\pi, D) = \{\pi' \mid \pi \approx_D \pi'\}$$

Note that the smaller the set $\mathcal{R}(\pi, D)$ is, the more information about π is revealed. The revealed knowledge represents a bound on the amount of information that may be revealed by a program that complies with set D. The next step is to define the amount of information a program reveals.

The behaviour of a program that an observer can see is the sequence of outputs it generates. Thus an observer cannot distinguish two environments if their runs produce the same sequence of visible output actions.

Definition 2.13 (Observed Knowledge ($\mathcal{K}$))**.** *Given a program C and an environment π, we define the knowledge of π that can be observed in C as:*

$$\mathcal{K}(\pi, C) = \{\pi' \mid Run(C, \pi) \equiv_{out} Run(C, \pi')\}$$

Our security property, Policy Controlled Release (PCR), states that the knowledge obtained from observing the program is bounded by the information released by the declassification policies.

Definition 2.14 (Policy Controlled Release (PCR))**.** *We say that a program C satisfies policy controlled release for the set of declassifiable expressions D if for all environments $\pi : \mathcal{K}(\pi, C) \supseteq \mathcal{R}(\pi, D)$.*

To illustrate the concept with an example, consider a set D with only the previously mentioned boolean expression, i.e. $D = \{\alpha_1 > 0\}$. Also, consider environments π and π', in which $\pi(\alpha, 1) = 1$ and $\pi'(\alpha, 1) = 5$. Thus, $\pi' \in \mathcal{R}(\pi, D)$ and vice-versa, since in both environments the expression in D has the same value. Now consider a program C, to be executed in both π and π'. For C to satisfy the PCR property, it has to produce the same outputs for the both environments. This means that, if the value of the declassifiable expression does not change, no new information is revealed by C. However, if the output sequence changes from π to π', that means that C is revealing something *more* than what is specified by D.

2.5 Soundness of the Framework

Let us anticipate the result we aim at, which states that if a program is secure according to our analysis, then the program satisfies the PCR property. The proof of this Theorem is given at the end of this section.

Theorem 2.15. *For any terminating program C and a set of declassifiable expressions D, if $valid(C, D)$ then the program C satisfies PCR with respect to D.*

The proof of the Theorem relies on determining a linking between correspondent runs of a same program, the existence of which is stated by Lemma 2.19. First we need to define the properties of this linking and the intuition behind how the linking works and why it must exist. The linking is inspired by the proof of soundness in Banerjee et al. [BNR08].

However, our proof is simpler because we do not need to consider the exact path taken by the program to reach a particular state – our D-equivalence property together with our flow-sensitive approach to check validity ensures that all runs leading to the same output actions take the same branches. Additionally, the proof is termination-insensitive. This means that for the proofs to go through, we assume that the loops, in which the conditional expression is non-declassifiable, terminate.

The core idea behind the linking is that a program can be in one of two distinct confidentiality levels: a level L (low, public) in which it may output data or a level H (high, secret) where it may behave differently depending on non-declassifiable information. For ease of notation we assume, without loss of expressiveness, that in a program any statement other than `skip` can only occur once. This allows us to assign a unique type, denoted $\Gamma_C(C')$, to each statement C' of program C, so we have a mapping: $\Gamma_{\cdot}(\cdot) : Prog \times Prog \rightarrow \{H, L\}$. When the program is clear from the context we omit it and simply talk about the type $\Gamma(C')$ of statement C'. Given a set of declassifiable expressions D, a program C in the form $C = C_1 \ ; \ C_2$, we type all non-compositional statements C' contained in C as follows:

$$\Gamma_{C_1 \ ; \ C_2}(C') \ = \ \begin{cases} \Gamma_{C_1}(C') & \text{if } C' \in C_1 \\ \Gamma_{C_2}(C') & \text{if } C' \in C_2 \end{cases}$$

And then, for a non-compositional C_i, we define $\Gamma_{C_i}(C')$ as:

C_i	C'	condition	$\Gamma_{C_i}(C')$
`if` c `then` C_1 `else` C_2	any	$\neg safe(c, C, D)$	H
`if` c `then` C_1 `else` C_2	$= C_i$	$safe(c, C, D)$	L
`if` c `then` C_1 `else` C_2	$\neq C_i$	$safe(c, C, D)$	$\Gamma_{C_1 \ ; \ C_2}(C')$
`while` $\overline{C}$; c `do` C_w	any	$\neg safe(c, C, D)$	H
`while` $\overline{C}$; c `do` C_w	$= C_i$	$safe(c, C, D)$	L
`while` $\overline{C}$; c `do` C_w	$\neq C_i$	$safe(c, C, D)$	$\Gamma_{\overline{C} \ ; \ C_w}(C')$
otherwise	`skip`	-	H
otherwise	$\neq$ `skip`	-	L

In the table above, the rightmost column represents the value of $\Gamma_{C_i}(C')$ when all the conditions on the columns to its left are met. For example, the first row of the table reads: when $C_i =$ `if` c `then` C_1 `else` C_2, C' is any statement and the condition $\neg safe(c, C, D)$ holds, then $\Gamma_{C_i}(C') = H$. The intuition behind the typing is as follows:

1. Every `skip` command is typed H. [1]

2. If C' is a conditional statement (`if` or `while`) whose condition c is not marked as declassifiable, i.e. $\neg safe(c, C, D)$ then $\Gamma(C') = H$ and also all statements nested inside C', directly or indirectly, are typed H.

[1] Although it is intuitive to type `skip` as L, we type it H for convenience, as doing so helps us to simplify the notion of *low continuation*, explained further ahead.

3. If C' is a conditional statement whose condition is declassifiable, then $\Gamma(C') = L$ (unless nested within a statement of the previous case) and we repeat the procedure for the statement(s) in the body in the same way.

4. Each non-compositional statement not typed H according to the above rules is typed L.

The type of a compositional statement C' is the type of its active command $head(C')$. In the L level the program will behave 'the same' in two D-equivalent environments. The next definitions capture this notion of 'the same'. We first consider the states that a program could reach.

Definition 2.16 (Compatible States ($\asymp$))**.** *Let C be a program and D be a set of declassifiable expressions. We say that two states σ_1 and σ_2 are compatible for C and D, denoted $\sigma_1 \asymp_{(C,D)} \sigma_2$, if the following conditions hold:*

1. $\forall \alpha \in$ **In** : *if* $cds(\alpha, C, D)$ *then* $(I_{\sigma_1}(\alpha) = I_{\sigma_2}(\alpha)$ *and* $PC_{\sigma_1}(\alpha) = PC_{\sigma_2}(\alpha))$.
2. $\forall x \in$ **Var** : *if* $cds(x, C, D)$ *then* $(E_{\sigma_1}(x) = E_{\sigma_2}(x)$ *and* $PC_{\sigma_1}(x) = PC_{\sigma_2}(x))$.

Intuitively, this definition reflects the fact that if the control dependencies of a variable or channel are declassifiable then they cannot be altered/read from by the program in a H level and as L behaviour has to be the same, they cannot distinguish between two D-equivalent environments.

Lemma 2.17. *The relation $\asymp_{(C,D)}$ is transitive.*

We define the *low continuation of* $C = C_1; \ldots; C_n$, denoted *L-cont*(C) as the statement $C_i; \ldots; C_n$ where i is the first index for which C_i is not typed high. Notice that if $\Gamma(C_1) = L$, then *L-cont*$(C) = C$. Now we define a correspondence relation over two runs of a program.

Definition 2.18 (Correspondence between two runs (Q))**.** *Let C be a program, π and π' be environments, and D be a set of declassifiable expressions. Let t be a prerun of $Run(C, \pi)$ and t' be a prerun of $Run(C, \pi')$ with $\|t\| = n$ and $\|t'\| = m$. A correspondence between t and t' is a relation $Q \subseteq \{0, 1, \ldots, n\} \times \{0, 1, \ldots, m\}$ satisfying the following conditions:*

1. *(zero-element)* $0\ Q\ 0$
2. *(completeness)* $\forall i \in \{1, \ldots, n\} : \exists j \in \{1, \ldots, m\} : i\ Q\ j$ *and vice versa*
3. *(trace-equivalence) For all i, j, with $t_i = \langle o_i, \langle C_i, \sigma_i, \pi \rangle\rangle$ and $t'_j = \langle o'_j, \langle C'_j, \sigma'_j, \pi' \rangle\rangle$, such that $i\ Q\ j$, the following conditions hold:*
   (a) *(output-equivalence)* $o(t_1) \ldots o(t_i) = o(t'_1) \ldots o(t'_j)$
   (b) *(state-compatibility)* $\sigma_i \asymp_{(C,D)} \sigma'_j$
   (c) *(level-agreement)* $\Gamma(C_i) = \Gamma(C'_j)$

(d) (code-agreement) $L\text{-}cont(C_i) = L\text{-}cont(C'_j)$

We say two runs correspond if there exists a correspondence relation between them.

From the requirement 'output-equivalence', it is clear that two corresponding runs produce the same output. The other requirements allow us to inductively build the correspondence relation Q between runs of a program in D-equivalent environments. We can now state the key Lemma in proving Theorem 2.15.

Lemma 2.19. *Let C be a program and D be a set of declassifiable expressions, satisfying $valid(C, D)$. Let π and π' be two environments satisfying $\pi \approx_D \pi'$, such that C terminates under both π and π'. Let $\omega_0 = \langle C, \sigma_{init}, \pi \rangle$ and $\omega'_0 = \langle C, \sigma_{init}, \pi' \rangle$. For each prerun S starting from ω_0 there exists a prerun S' starting from ω'_0 such that S corresponds to S'.*

Proof of the Lemma can be found in the Appendix and with this we now prove Theorem 2.15.

Proof of Theorem 2.15. Lemma 2.19 implies that the executions of a D-valid program in two D-equivalent environments can be linked in a way that guarantees they will result in the same runs. This implies that for all environments π, π' if $\pi' \in \mathcal{R}(\pi, D)$ then also $\pi' \in \mathcal{K}(\pi, C)$. □

CHAPTER 3

Graph-Based Implementation

In this chapter we present our graph-based implementation of the expression-matching framework, which we refer to as *graph-based PCR*. The implementation uses a form of graphs to represent expressions that can be held by variables. The graphs work as finite structures able to represent possibly infinite sets of expressions. This graph-based approach presents a computable mechanism to validate a program according to the PCR property. The implementation is, however, a *safe approximation* of the framework: if a program is deemed safe by the implementation, it is guaranteed to be safe by the framework as well, but the opposite is not true. In other words, the graph-based approach is guaranteed to never validate an unsafe program, but it can, under some circumstances, reject a safe program. The precision of the approximation is discussed in more detail in Chapters 5 and 6.

In the following section (3.1) we revisit the examples of Section 2.1, showing how the graph-based PCR handles them. Then we introduce our *expression graphs* in Section 3.2, both for the program and policy, and in Section 3.3 we proceed to define how the policy graph is matched against the program graph. The soundness of the implementation is then shown in Section 3.4, where we show that a program deemed safe by the graph-based PCR is also safe according to the expression-matching framework. In Section 3.5 we present and analyze algorithms for this approach, in order to demonstrate its tractability. Finally, in Section 3.6 we extend both our toy language and the expression graphs in order to support user-defined functions, showing that the approach is suitable for modular programs, and thus paving the way for supporting extensions such as object-orientation.

The contents of this chapter are presented in papers [RBdH+10, RBdH+11].

3.1 Revisiting the Examples

In this section we revisit the examples of Section 2.1, and show how graph-based PCR handles them. First, let us recall the authentication program example, already pre-processed:

Example 3.1. *Authentication program in SSA format:*

`struct` $x_1 := \alpha$;
`bool` $c_1 := iscomplete(x_1)$;
`string` f_0;
`bool` v_0;

$depends(\beta, c_1)$;
`if` c_1 `then`
    $c_2 := hascred(x_1)$;
    `if` c_2 `then`
        $f_1 := credential(x_1)$;
    `else`
        $f_2 := lastname(x_1)$;
    $f_3 := \phi_{c_2}(f_1, f_2)$;
    $v_1 := validate(f_3)$;
`else`
    $f_4 := login(x_1)$;
    `string` $y_1 := \beta$;
    $v_2 := verify(f_4, y_1)$;
$v_3 := \phi_{c_1}(v_1, v_2)$;
$f_5 := \phi_{c_1}(f_3, f_4)$;
$\gamma := v_3$;

Expression Graph. An expression graph is an abstraction for representing the set of expressions that may be assigned to one or more variables, taking into consideration the input channels and the constants a program refers to. In an expression graph nodes represent variables, constants and I/O channels, whereas directed edges represent assignments. The labels on the edges denote the functions used in the assignments, while the subscripts indicate the indices of the arguments from the parent nodes. Edges of ϕ-functions are dashes as they are used to represent distinct paths that information can follow during an execution, each path separately representing a set of expressions. The `control` edge illustrates that there is a control dependency between two nodes, the parent being the variable representing the control expression. Figure 3.1 shows the expression graph g associated with the variable v_3 of our program. For clarity, a control edge between c_1 and β is omitted, since c_1 also causes a control dependency in v_3, and it will be analyzed anyway.

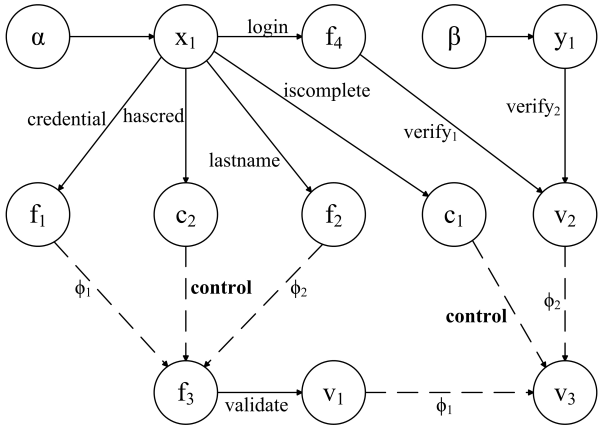


Figure 3.1: Expression graph for variable v_3 of authentication program

Policy Graph. Declassification policies are also represented using graphs. In fact, a policy graph is similar to the expression graphs associated with program variables, except for some key differences, including: (1) nodes can be labeled with "wildcards", i.e., labels in the form $*$, (2) certain nodes in the policy are marked as "final nodes" (represented by the double lined circle), representing expressions that can be declassified. A declassification policy consists of a graph which might contain several disjoint components (to allow multiple expressions to be released). The policy graph, d, for our authentication program in Example 3.1 is given in Figure 3.2. We know that information from either channels α and β cannot directly flow to the channel γ, the policy of Figure 3.2 allows such a flow under a few additional conditions. The following operations are allowed: two boolean checks on the user's record α (if it has a credential and if is a complete record), two validation operations over user's information (validation through the credential or the last name), and a verification of the user login against a supplied password from channel β. The final nodes $*_1$, $*_2$, $*_3$, $*_5$ and $*_7$ represent the expressions that can be declassified. Note how this policy graph corresponds to the set D, discussed in Example 2.2.

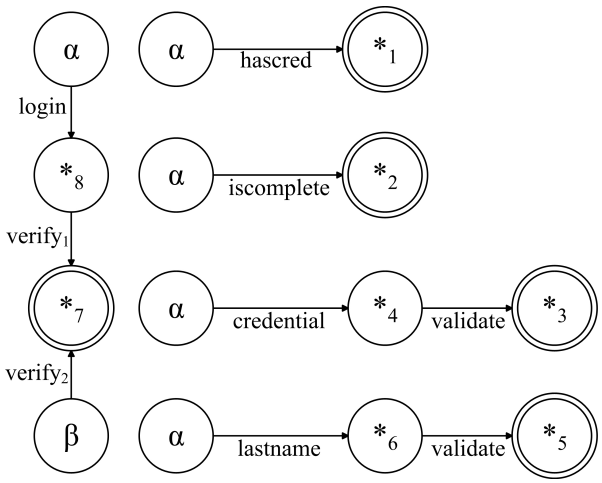


Figure 3.2: Policy graph for example of authentication program

Policy Matching. Now that we have both the program and the policy graph, we can check if the program is *safe*. In our program the (low) output γ is assigned the value of variable v_3, so what we now have to check is that the paths in the program graph indicating the flow of information from a high input to v_3 are safe, i.e., that they match at least one component of the declassification policy. This analysis is done in two stages: first all data dependencies of a node are checked, later in the second stage the control dependencies are checked. The node representing v_3 in the graph has 3 *information paths* (defined precisely in the next sections, not the standard concept of a path in a graph) reaching it: (a) one that comes from channel α passing through nodes x_1, f_1, f_3 and v_1, (b) another also coming from α, but passing through f_2 instead of f_1, and the final one (c) coming from both α and β, converging on node v_2. These paths represent the three possible outcomes of the nested `if` commands.

To determine that the node v_3 is safe we will first analyse its parents. First, node

v_2, has only information path (c) reaching it. This path matches the leftmost component of our declassification policy in Figure 3.2, and we say that node v_2 *simulates* node $*_7$, meaning that all expressions possibly held by v_2 are recognized by $*_7$. Or, in other words, $v_2 \sim_{g,d} *_7$. Because of this, this path to v_2 is marked as *data dependency safe*, and so is the node, since this is the only path reaching it (note that the graphs are not exactly the same, our definition of policy simulation handles this properly). Since v_2 has no additional control dependencies (the dependency with c_1 is processed in the analysis of v_3), we now know it is a *safe* node.

With (node) v_2 being safe, we now analyse v_1. This node has two information paths (a) and (b) reaching it. We can see that, for each path, v_1 simulates a final node of each of the 3-node components of the policy ($*_3$ and $*_5$), on the bottom of Figure 3.2. Thus, both paths are data dependency safe, and so is the node itself. There is however a control dependency that we have to consider, with c_2 that reaches v_1. But here node c_2 simulates the final node of the topmost policy component ($*_1$), thus making it safe (it has no control dependencies or other paths) and thus making v_1 *control dependency safe*. Therefore, we now know that v_1 is a safe node.

We can now go back to v_3. Since its two parent nodes are safe, we know that v_3 is a data dependency safe node, since all the paths were covered. In order to demonstrate it is also control dependency safe, we need to show that node c_1 is safe, this is done by showing that the node simulates a final node of a policy graph ($*_2$). Thus, v_3 is a safe variable and the program's expression graph is deemed valid.

Statistic Calculation. We now recall the second example, about the statistical calculation over data. Recall the average calculation program:

Example 3.2. *Average calculation program in SSA format:*

```
int a_1 := 0;
int i_1 := 0;
int l_1 := length(α);
bool c_1 := leq(i_1, l_1);

while (c_3 := φ_{c_3}(c_1, c_2);
       a_3 := φ_{c_3}(a_1, a_2);
       i_3 := φ_{c_3}(i_1, i_2);
       c_3) do
    int t_1 := α;
    a_2 := add(a_3, t_1);
    i_2 := add(i_3, 1);
    c_2 := leq(i_2, l_1);

a_4 := div(a_3, l_1);
γ := a_4;
```

Let us see how our graph-based implementation deems this program secure, in a computable way. To do so, it produces the expression graph associated to variable a_4 (Figure 3.3). For the sake of clarity, Figure 3.3 only includes data-dependencies of a_4. Since

we also have to consider the control dependencies, and the only control dependency of a_4 goes through c_3, we represent the graph associated with node c_3 separately in Figure 3.4. The numeric annotations on the edges indicate looping contexts in which assignments are performed. The assignments that happen within the loop have its corresponding edges marked with 1. The other edges are part of context 0 (not within a loop), and their annotations are omitted. These are needed for checking a relation called input uniqueness, discussed further ahead.

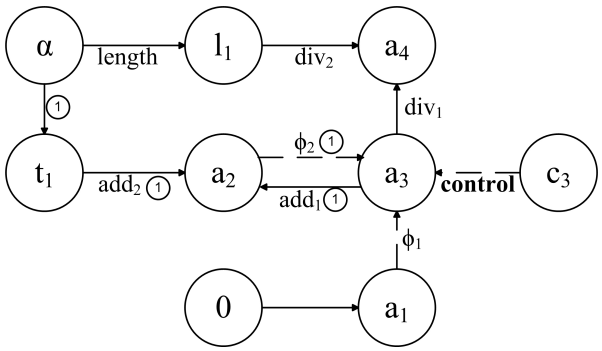


Figure 3.3: Expression graph for variable a_4

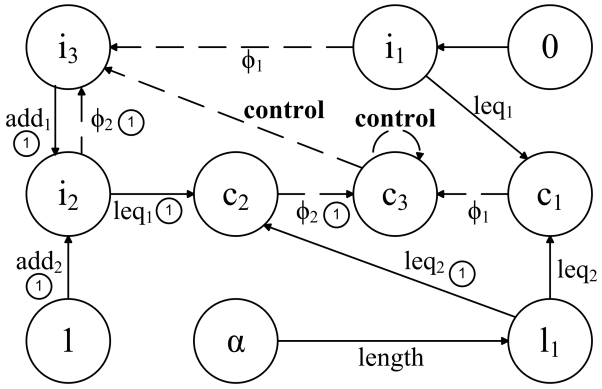


Figure 3.4: Expression graph for variable c_3

The policy graph for the average example is given in Figure 3.5. This policy allows for the release of a sequence of additions over entries from input α. The final node $*_3$ represents the sum expression.

The policy contains an additional constraint that states that no individual α-values should reach $*_3$ more than once (every access to α must be unique). Assuming that d is the policy graph, we say that $(\alpha, *_3) \in uni(d)$. This is called an *input uniqueness relation*, we discuss how to express this in Section 3.2.

We also assume that there is an omitted component of the policy graph that specifies that the expression $length(\alpha)$ can be declassified.

This example program is deemed valid by the policy. Node a_3 simulates node $*_3$ on the policy and the α-uniqueness constraint is satisfied through the use of the looping

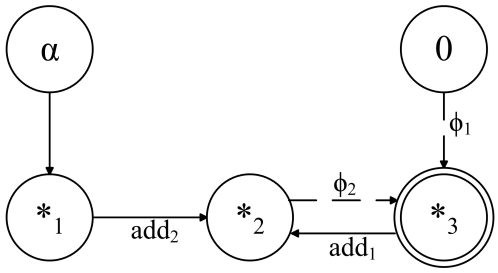


Figure 3.5: Declassification policy graph for average example

context annotations. Variable l_1 holds $length(\alpha)$, which is also authorized as previously mentioned. This allows us to mark c_3 as safe, which in turn makes a_3 control dependency safe. Therefore, a_4 is marked as safe. The mechanisms used in this process are detailed in the next sections.

It is important to note that, since our approach works as a static analyzer, it is beyond the focus of our representation mechanism (i.e. graphs) to represent the run-time behaviour of the program, including the number of times a given loop runs. This problem, however, can be treated by a combination of static analysis and runtime enforcement, as it is done in Chapter 4. Also, a discussion on how to tackle the problem only statically can be found in Chapter 5, Section 5.2.

Encryption. Finally, we show how the graph-based implementation tackles the encryption example:

Example 3.3. *Encryption program:*

`text` $x_1 := \alpha$;
`int` $k_1 := \beta$;
$x_2 := enc(x_1, k_1)$;
$\gamma := x_2$;

Here, we have a policy that allows the disclosure of *any* input channel, as long as it is encrypted with a specific key, using a specific encryption function. For this, we need to use the wildcards in the policy. Figure 3.6 shows the graphs for both the policy and the program. In this case, node $*_2^{\mathtt{in}}$ in the policy is a wildcard that matches any input node, and thus it matches node α in the graph, making it clear that the content of variable x_2 can be made public, matching the final node $*_1$.

3.2 Expression Graphs

In this section we formalize the notion of expression graphs, both for the program and the policy. Note that program graphs are automatically calculated based on the program's source, whereas the policy graph is supplied by the policy writer.

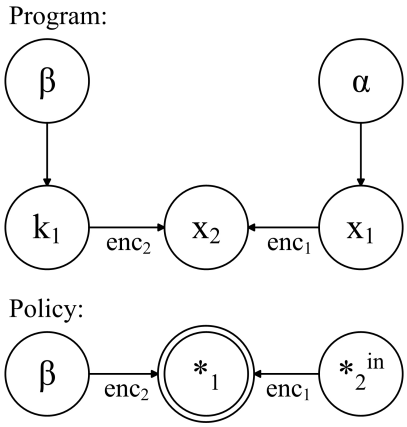


Figure 3.6: Encryption program and its matching policy

3.2.1 Program Graphs

First we define how a program expression graph (program graph, for short) is built from the program. We consider directed graphs, given by $g = (V, E) \in \mathcal{G}$ in which $V \subseteq$ **Vertex** and $E \subseteq$ **Edge** are the sets of vertices and edges, respectively. Vertices and edges are structured objects that contain labeling information. Each vertex has the form $n = (l, t)$, where l is the label[1] and t is the type, which is one of `var`, `in`, `out`, and `const`, corresponding to variables, I/O channels and constants, respectively, and which we denote by $type(l)$. We use the convention of denoting the vertex with label l by n_l, and we assume that the type or possible types of the node is clear from the label. For instance, n_x, n_α, n_γ, n_ρ and n_N are nodes of types `var`, `in`, `out`, any and `const`, respectively. Each edge has the form $e = (n, n', t, u)$, in which: n and n' are the origin and destination vertices, respectively; t is the edge type, which can be `plain` (for assignments with no function application), `control` (for control dependencies between boolean variables in conditionals and variables assigned inside the conditional block), or f_i, a function name f subscripted with an index i to the function, for edges that represent function applications[2]; $u \in \mathbb{N}$ is an index that represents the looping context in which the assignment represented by the edge takes place, i.e. it is different than zero if the assignment takes place within a loop, and different loops are marked with distinct indexes. We use metavariables u and v to denote looping context indices. We write $n \xrightarrow[u]{t} n'$ as a constructor that returns an edge (n, n', t, u).

We lift set union ($\cup$) to graphs in the standard way: $(V_1, E_1) \cup (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$. [3] We define the following pre-order on $\mathcal{G}$. For $g_1 = (V_1, E_1)$, $g_2 = (V_2, E_2) \in \mathcal{G}$, we write $g_1 \precsim g_2$ if $V_1 \subseteq V_2$ and there exists an injection $\varpi : \mathbb{N} \overset{1-1}{\to} \mathbb{N}$ such that $\{(n_1, n_1', t_1, \varpi(u_1)) \mid (n_1, n_1', t_1, u_1) \in E_1\} \subseteq E_2$. When $g_1 \precsim g_2$ and $g_2 \precsim g_1$, we say that the two graphs are equivalent denoted as $g_1 \simeq g_2$. Finally, the operator $\prec$ is defined

[1] Here, the term *label* used for nodes and edges denotes a *name* (or *id*) of that node/edge. Not to be confused with the notion of *security label*, often used in computer security literature.

[2] For unary function names we elide the argument index.

[3] Note that if $V_1 \cap V_2 = \emptyset$, then the union will consist of two disjoint graphs.

the obvious way (using $\subset$ instead of $\subseteq$).

To build the graph we use the function G, defined in Figure 3.7, which takes a command C and a looping context index u as arguments and returns the corresponding graph. We write $G(C)$ as a short to $G_0(C)$. The ϕ-functions generated by the SSA translation are used to handle control flow dependencies. Note that there are multiple definitions for assignments, according to the format of the RHS operator. Also, for the looping context index, the ϕ-functions of the loops receive a special treatment. In these cases, the function is called and returns the first argument (ϕ_1 edge) always once, regardless if the loop runs or not, whereas it is called and returns the second argument (ϕ_2 edge) as many times as the loop runs. Thus, the ϕ_1 edge is labeled with the looping context index of before entering the loop, and ϕ_2 is labeled with the one of the loop body. This is specified by function Φ. Also, we use $\mathit{fresh}()$ to return a "fresh" integer value, i.e. a previously unused value, used for looping context identifiers. We use ϵ to denote the control context index on the `control` edges. The value of ϵ is unspecified, as it is irrelevant to our other constructions.

$$
\begin{array}{rcl}
G : \mathit{Prog} \times \mathbb{N} \to \mathcal{G} & & \\
\Phi : \mathit{Prog} \times \mathbb{N} \times \mathbb{N} \to \mathcal{G} & & \\
G_u(\texttt{skip}) & = & \emptyset \\
G_u(C_1 \ ; \ C_2) & = & G_u(C_1) \cup G_u(C_2) \\
G_u(\texttt{if } c \texttt{ then } C_1 \texttt{ else } C_2) & = & G_u(C_1) \cup G_u(C_2) \\
G_u(\texttt{while } \overline{C} \ ; \ c \texttt{ do } C) & = & \Phi_{u,v}(\overline{C}) \cup G_v(C) \\
 & & \text{where } v = \mathit{fresh}() \\
G_u(x := \alpha) & = & n_\alpha \xrightarrow[u]{\texttt{plain}} n_x \\
G_u(\gamma := x) & = & n_x \xrightarrow[u]{\texttt{plain}} n_\gamma \\
G_u(x := y) & = & n_y \xrightarrow[u]{\texttt{plain}} n_x \\
G_u(x := f(y_1, \ldots, y_k)) & = & n_{y_1} \xrightarrow[u]{f_1} n_x, \ldots, n_{y_k} \xrightarrow[u]{f_k} n_x \\
G_u(x := \phi_c(a, b)) & = & n_a \xrightarrow[u]{\phi_1} n_x, n_b \xrightarrow[u]{\phi_2} n_x, n_c \xrightarrow[\epsilon]{\texttt{control}} n_x \\
G_u(\mathit{depends}(\theta, c)) & = & n_c \xrightarrow[\epsilon]{\texttt{control}} n_\theta \\
\Phi_{u,v}(C_1 \ ; \ C_2) & = & \Phi_{u,v}(C_1) \cup \Phi_{u,v}(C_2) \\
\Phi_{u,v}(x := \phi_c(a, b)) & = & n_a \xrightarrow[u]{\phi_1} n_x, n_b \xrightarrow[v]{\phi_2} n_x, n_c \xrightarrow[\epsilon]{\texttt{control}} n_x
\end{array}
$$

Figure 3.7: Graph building function. Note that $\mathit{fresh}()$ returns a previously unused value and that ϵ is used to denote an unspecified index.

Definition 3.1 (Expression Graph)**.** *The expression graph $g \in \mathcal{G}$ of a program C is given by $G(C) = G_0(C)$. $G_0(C)$ is constructed in Figure 3.7.*

We use $\mathit{nodes}(g)$ and $\mathit{edges}(g)$ to denote the sets of vertices and edges of g, respectively. We use the 5-ary predicate written $n \xrightarrow[u]{t}{}_g n'$ which is defined to hold if $(n, n', t, u) \in \mathit{edges}(g)$. When g is implicit in the context where the predicate is used, we write simply $n \xrightarrow[u]{t} n'$. Note that this notation overloads the notation for the edge constructor. Whether we use this notation to denote the constructor or the predicate will

be clear from the context, throughout this thesis. When the value of u is irrelevant, we write $n \xrightarrow{t} n'$ which is equivalent to $\exists u : n \xrightarrow[u]{t}_g n'$. And when $t = \texttt{plain}$, we write $n \rightarrow n'$ which is equivalent to $\exists u : n \xrightarrow[u]{\texttt{plain}}_g n'$. We also write t_d when t is a data dependency type, i.e. $t \neq \texttt{control}$. In a similar fashion, $n \xrightarrow{w}{}^* n'$ denotes that there is a path (excluding `control` edges) between nodes n and n', with w being the sequence of labels on this path. In other words, w denotes the sequence of labels of edges used to reach n', starting from n, and excluding `control` edges. Additionally, $n \xrightarrow[u]{w}{}^* n'$ denotes that the whole path w has the same context u and we use $n \rightarrow^* n'$ when both w and u are irrelevant. We also use $\xrightarrow{\phi}$ to denote either $\xrightarrow{\phi_1}$ or $\xrightarrow{\phi_2}$. Using a notation analogous to that of bisimulation [Mil89], we call an edge a τ-edge if its type is either `plain` or ϕ. Finally, we write $\not\rightarrow n$ to denote that the indegree of n is zero, and function $type(n)$ returns the type of a node n.

Before presenting a first theorem, we first present the definition of input-uniqueness.

Definition 3.2 (Input Uniqueness on Expressions)**.** *An expression e : $\mathbf{Exp}\langle \mathbf{In} \times \mathbb{N} \rangle$ is said to be α-unique if every occurrence of α represents a distinct access on that input channel, i.e. every α_i occurring in e has a distinct index i.*

We now define a notion of α-uniqueness for graph nodes n that will be used below to express the requirement that expressions recognized by n be α-unique. Given a graph g, this notion is represented by a set of pairs $uni(g) \subseteq \mathbf{In} \times \mathbf{Vertex}$ and $(\alpha, n) \in uni(g)$ indicates that n is intended to represent only α-unique expressions. In policy graphs (see Definition 3.7 below), this set is given explicitly. For program graphs, we derive it according to the following definition. (We believe that this definition is somewhat conservative in the sense that it may not extract all α-uniqueness pairs that could be derived in some cases, but it serves us well, is simple, and admits efficient computation.)

Definition 3.3 (Input Uniqueness on Nodes)**.** *Let n be a node in the program expression graph g, α be an input channel and n_α be the graph node that represents α. We say that n is* α-unique, *and write $(\alpha, n) \in uni(g)$ if none of the following relations hold:*

1. *If n_α reaches some node n', which in turn reaches n via two distinct paths. That is:*
   $\exists n' \in nodes(g), w, w_1, w_2, f :$
   $n_\alpha \xrightarrow{w}{}^* n' \xrightarrow{w_1 \cdot f_i}{}^* n,$
   $n_\alpha \xrightarrow{w}{}^* n' \xrightarrow{w_2 \cdot f_j}{}^* n,$
   $f \neq \phi, i \neq j, w_1 \cap w_2 = \emptyset, n' \neq n_\alpha$

2. *If n is reached by n_α and also part of a cycle, but the looping contexts of the cycle and the path differ at some point. That is:*
   $\exists u, v : n_\alpha \rightarrow^* \xrightarrow[u]{t_d} \rightarrow^* n \rightarrow^* \xrightarrow[v]{t'_d} \rightarrow^* n, u \neq v$

3. *If a parent node of n is not α-unique. That is:*
   $\exists n' \in nodes(g) : n' \xrightarrow{t_d} n, (\alpha, n') \notin uni(g)$

The definition specifies the cases in which a node holds an expression that combines more than one occurrence of a same input value. The first relation states the case in which a same expression on the input channel converge on a single node via two different paths, through some function f. The second relation is about a node inside a loop, holding an expression on the input that grows iteratively with the loop execution. If the input access is inside the same loop of the cycle in the graph, then each iteration will grow the expression with a "fresh" value from the input. Otherwise, this freshness cannot be guaranteed, and neither input uniqueness. Finally, the third relation states that, once a node is marked as not α-unique, then so are its children. The correctness of the definition is implied by Theorem 3.6, presented further in this Section. However, we also present it as the Corollary below, for clarity.

Corollary 3.4. *Let C_0 be a program, π_0 be an environment, t be a run in $Run(C_0, \pi_0)$, and $g = G(C_0)$. For any configuration $\langle C, \sigma, \pi \rangle \in t$, σ satisfies the following:*

$$\forall x \in \textbf{Var}, \alpha \in \textbf{In}, n_x \in nodes(g) : \textit{if } (\alpha, n_x) \in uni(g) \textit{ then } E_\sigma(x) \textit{ is } \alpha\textit{-unique.}$$

We now define a function $exp : \mathcal{G} \times \textbf{Vertex} \to \wp(\textbf{Exp}\langle \textbf{In} \times \mathbb{N} \rangle)$ which makes precise the set of expressions represented by each graph node. We write $exp_g(n)$ to denote the set of expressions represented by node n in graph g, and we omit g when it is clear from context.

$$\begin{aligned} exp_g(n_N) &= \{N\} \\ exp_g(n_\alpha) &= \{\alpha_i \mid i \in \mathbb{N}\} \\ exp_g(n_\gamma) &= \bigcup_{n' \to n_\gamma} exp_g(n') \\ exp_g(n_x) &= \Psi_{n_x}(Exp) \end{aligned}$$

in which

$$Exp = \{f(e^1, \ldots, e^k) \mid \forall n^1, \ldots, n^k : n^i \xrightarrow{f_i} n_x \wedge e^i \in exp_g(n^i)\} \cup \bigcup_{n' \xrightarrow{\tau} n_x} exp_g(n')$$

and

$$\Psi_n(Exp) = \{e \in Exp \mid \forall \alpha \in \textbf{In} : \text{if } (\alpha, n) \in uni(g) \text{ then } e \text{ is } \alpha\text{-unique}\}$$

where Ψ_n is a filter used to deal with input uniqueness, removing expressions which don't satisfy α-uniqueness if the node n holds that property. Note that in the above definition exactly one of the subsets of $exp(n_x)$ will be non-empty as each node is reached by either a single plain edge, two ϕ edges or k function edges.

We also define the function $cexp : \mathcal{G} \times \textbf{Vertex} \to \wp(\textbf{Exp}\langle \textbf{In} \times \mathbb{N} \rangle)$ that computes all conditional expressions the value held by a node can depend upon.

$$cexp_g(n) = \bigcup_{n' \xrightarrow{t} n} cexp_g(n') \cup \bigcup_{n' \xrightarrow{\texttt{control}} n} exp_g(n')$$

We can finally state our first result, which is about the soundness of the graph translation. First we define correspondence between the graph and a single program state, and then proceed to the theorem that relates the graph with a program.

Definition 3.5 (Correspondence Between State and Graph)**.** *Let σ be a program state and g be a program expression graph. We say that σ and g correspond to each other if they satisfy $P(\sigma, g)$, defined as the conjunction of the following properties:*

$$\begin{aligned}
P_E(\sigma, g) &\equiv \forall x \in \textbf{Var} : \textit{if } E(x) \textit{ is defined then} \\
&\quad (x, \texttt{var}) \in nodes(g) \wedge E(x) \in exp(n_x) \\
P_I(\sigma, g) &\equiv \forall \alpha \in \textbf{In} : \textit{if } I(\alpha) > 0 \textit{ then} \\
&\quad (\alpha, \texttt{in}) \in nodes(g) \\
P_O(\sigma, g) &\equiv \forall \gamma \in \textbf{Out} : \textit{if } O(\gamma) \textit{ is defined then} \\
&\quad (\gamma, \texttt{out}) \in nodes(g) \wedge O(\gamma) \subseteq exp(n_\gamma) \\
P_{PC}(\sigma, g) &\equiv \forall \rho \in \textbf{Var} \cup \textbf{IO} : \textit{if} PC(\rho) \textit{ is defined then} \\
&\quad (\rho, type(\rho)) \in nodes(g) \wedge PC(\rho) \subseteq cexp(n_\rho)
\end{aligned}$$

Theorem 3.6 (Soundness of the Graph Translation)**.** *Let C_0 be a program, π_0 be an environment, t be a run in $Run(C_0, \pi_0)$, and $g = G(C_0)$. For any configuration $\langle C, \sigma, \pi \rangle \in t$, σ satisfies $P(\sigma, g)$.*

P_E states that each variable has a corresponding node, and that the expression of the variable is contained in the set of possible expressions held by that node; P_I states that for each input channel accessed in the process there exists a corresponding node in the graph; P_O states that for each output channel there exists a corresponding node, and that the set of expressions sent to that output in the process is a subset of the set of possible expressions held by that node; finally, P_{PC} states that for each variable (and I/O channel), the set of conditional expressions that the variable depends on is a subset or equal to that set for the corresponding node.

3.2.2 Policy Graphs

Policy graphs work in the same way as program graphs, with a few key differences: (1) one or more nodes are marked as "final nodes"; (2) nodes can have "wildcards" as label, in the form of $*_i$, meaning that they can match any other node, regardless of the label; (3) edges don't have control context labels; and (4) input uniqueness relations are provided with the policy, working as constraints over the recognized expressions. These differences are justified by the fact that the program graph is calculated, in order to represent all possible expressions that can be held by variables in the program, whereas policy graphs are supplied, recognizing the set of expressions that can be declassified. For clarity, we write $*^t$ to denote the wildcard on a node of type t, and just $*$ when $t = \texttt{var}$. The matching process between the policy and program graph is defined in Section 3.3.

Definition 3.7 (Declassification Policy)**.** *A declassification policy is a graph $d \in \mathcal{D}$, with possibly disjoint components, in the form $d = (V, E, V_f, U)$, where $V \subseteq$ **Vertex** is a set of vertices, $E \subseteq$ **Edge** is a set of edges, $V_f \subseteq V$ is a set of final vertices and $U \subseteq$ **In** $\times V$ is a set of input uniqueness relations.*

The final vertices hold the expressions allowed to be declassified. We write $fnodes(d)$ to denote the set V_f of final vertices on policy graph d. Thus, the set of expressions allowed by a policy graph is determined by $\bigcup_{n_f \in V_f} exp_d(n_f)$. Also, we use $uni(d)$ to return the set of input uniqueness relations from a policy d. Thus, for our working example of the average salary, we have that the policy of Figure 3.5 recognizes the set of expressions $\{0, add(0, \alpha_i), add(add(0, \alpha_i), \alpha_j), \ldots\}$, with all indices on α being distinct, as $(\alpha, *_3) \in uni(d)$. It is important to point that this work addresses the problem of enforcing declassification policies, rather than specifying them. However, it is fairly straightforward to derive a rule that translates the policy graph to/from some form of regular expressions (e.g. regular tree expressions [CDG+07]).

With this, we can extend the definition of exp to define the expressions held by $*$ nodes in the policy graphs:

$$\begin{aligned} exp_d(*^{\texttt{const}}) &= \text{Const} \\ exp_d(*^{\texttt{in}}) &= \mathbf{In} \times \mathbb{N} \end{aligned}$$

where Const denotes the set of (syntactical) constants. Nodes $*^{\texttt{var}}$ hold the same expressions as `var` nodes on the program graph, thus defined by $exp_d(n_x)$. A node of type `const` with a wildcard $*$ label holds any constant as its expression and a node of type `in` with wildcard label matches any indexed input α_i.

3.3 Graph Matching

Having defined the expression graphs of program and policy in the previous section we now introduce the mechanism that matches them. This will allow us to define which nodes are safe according to the policy. If all output nodes are safe, then the program represented by the graph is safe too. Note that multiple disjoint components of a declassification graph may be needed to show the safety of a program. To simplify the matching process we first extract the sub-graphs from the program's expression graph that could be validated separately (called *information paths*). Next we carry out the matching between an information path and a (single) component on a policy graph. The matching mechanism is defined via a number of predicates that serve as computable implementations of the program validity predicates presented in Section 2.3. Thus, predicates in this section are named after their counterparts of Section 2.3, like the following: $\overline{dds}$ is the graph-based implementation of dds defined for program validity. Section 3.5 presents and discusses the algorithms for the mechanism introduced here.

An information path captures one way in which expressions can flow into a node starting from input channels and constants. Multiple function edges to the same node represent that such node holds an expression equal to the application of that function with its parents as arguments, thus all edges need to be included in the path. On the other hand, ϕ-edges represent points where control flow may branch, and therefore each ϕ-edge represents a distinct information path. Note that an information path may still have multiple incoming τ-edges because loops may cause the program flow to reach the same

node multiple times. We represent an information path p by the set of edges it contains (the set of vertices for the path can be obtained by collecting the source or destination of the edges). A set of information paths is thus a set of set of edges. To streamline the notation, we define the following operators for sets of information paths S and S', and single elements e (i.e., edges):

$$\begin{array}{lcl} S \oplus e & = & \{g \cup \{e\} \mid g \in S\} \\ S \otimes S' & = & \{g \cup g' \mid g \in S, g' \in S'\} \end{array}$$

We then define function ip to collect all the information paths that reach a node n in a graph g, which is a set of sub-graphs of g. Recall from the graph construction rules that every node is either reached by a number of τ edges *or* a number of f_i edges in which function f is the same and index i is distinct in all edges.

Definition 3.8 (Information Path)**.** *Let g be a program expression graph and n be a node in this graph. We say that p is an information path for n (in g) if $p \in ip_g(n)$, defined as:*

$$ip_g(n) = \begin{cases} \{\emptyset\} & \text{if } \not\rightarrow n \\ \bigcup\limits_{n' \xrightarrow[u]{\tau} n} ip_g(n') \oplus (n', n, \tau, u) & \text{if } n \text{ is reached by one or more } \tau \text{ edges} \\ \bigotimes\limits_{n' \xrightarrow[u]{f_i} n} ip_g(n') \oplus (n', n, f_i, u) & \text{if } n \text{ is reached by one or more } f_i \text{ edges} \end{cases}$$

We say that an information path for n is maximal *if it is not a subset of some other information path. The set $mip_g(n)$ contains all maximal information paths for n.*

$$mip_g(n) = \{p \mid p \in ip_g(n), \forall p' \in ip_g(n) : p \not\prec p'\}$$

Again, we omit subscript g when it is clear from context.

If one information path is a sub-graph of another one, then validating the larger graph also validates the smaller so we only need to consider the maximal information paths.

Figure 3.8 presents examples of information paths. The first graph demonstrates how ϕ-edges create distinct information paths, as opposed to standard (function) edges. The second graph shows how the calculation of maximal information paths handles cycles: in their presence the whole structure is included, as one of the information paths is a sub-graph of the other.

The next step is to relate the set of maximal information paths to the policies. This is done by the notion of *policy simulation* which is a "bundled" version of weak simulation in state transition systems [Mil89]. First, we need some supporting notation. We call two nodes *similar* $n \simeq n'$ if they have the same type and either the labels are the same or one of them is a wildcard $*$.

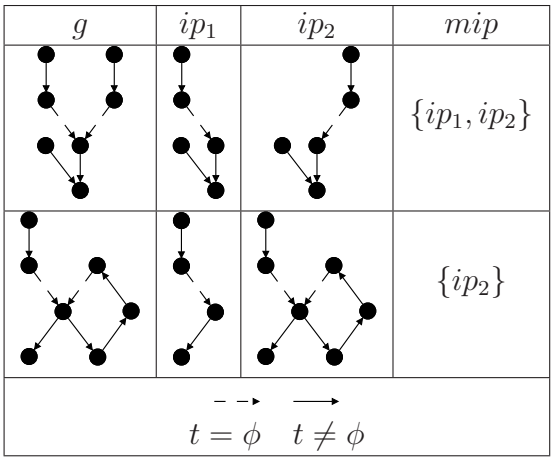


Figure 3.8: Example of information path calculation

Definition 3.9 (Policy Simulation)**.** *Let C be a program, $g = G(C)$ be its expression graph, p be an information path for some node of g and d be a policy graph. A relation $\mathcal{R} \subseteq nodes(p) \times nodes(d)$ is called a policy simulation if, for each $(n, n_d) \in \mathcal{R}$, the following holds:*

1. *Nodes n and n_d are similar, i.e. $n \simeq n_d$.*

2. *Node n weakly simulates node n_d. That is, the following holds:*

   - *If $\exists n' \in nodes(p) : n' \xrightarrow{\tau} n$ then*
     $\exists n'_d \in nodes(d) : n'_d \ (\xrightarrow{\tau})^* \ n_d$ and $(n', n'_d) \in \mathcal{R}$.
   - *If $\exists f, \exists n^1 \ldots n^k \in nodes(p) : n^i \xrightarrow{f_i} n$ then*
     $\exists n^1_d \ldots n^k_d, n'_d \in nodes(d) : n^i_d \ (\xrightarrow{\tau})^* \xrightarrow{f_i} n'_d (\xrightarrow{\tau})^* \ n_d$ and $(n^i, n^i_d) \in \mathcal{R}$.

3. *Node n satisfies, in g, the input uniqueness restrictions specified for n_d in d, i.e. for every $\alpha \in$ **In**, if $(\alpha, n_d) \in uni(d)$ then $(\alpha, n) \in uni(g)$.*

We use $\sim_{p,d}$ to denote the largest policy simulation (i.e. the union of all of them) between information path p and policy graph d.

Next, we present the definitions for validating a program's expression graph, implementing those of program validity. First we define node validity, a concept analogous to that of variable validity in Definition 2.7.

Definition 3.10 (Node Validity)**.** *Let g be a program expression graph, n be a node in g, p be an information path for n and d be a policy graph. About node n, we say that:*

1. *n is data dependency safe in p if it matches some final node of the declassification*

policy d or if all its parents in p are already data dependency safe.

$$\begin{aligned}\overline{dds_p}(n,d) \quad \equiv \quad & (\exists n_f \in fnodes(d) : n \sim_{p,d} n_f) \textit{ or} \\ & (\forall n_\alpha \in nodes(p), w : \textit{if } n_\alpha \xrightarrow{w}{}^{*} n \textit{ then} \\ & \exists n' \in nodes(p), w', w'' : n_\alpha \xrightarrow{w'}{}^{*} n' \xrightarrow{w''}{}^{*} n \wedge \\ & w = w' \cdot w'' \wedge \overline{dds_p}(n', d))\end{aligned}$$

2. *n is data dependency safe (DDS) if all the maximal information paths that reach it are data dependency safe.*

$$\overline{dds}(n,g,d) \quad \equiv \quad \forall p \in mip(n,g) : \overline{dds_p}(n,d)$$

3. *n is control dependency safe (CDS) if all nodes that reach it by a path starting with a* `control` *edge are DDS.*

$$\overline{cds}(n,g,d) \quad \equiv \quad \forall n', n'' \in nodes(g) : \textit{if } n' \xrightarrow{\texttt{control}} n'' \xrightarrow{w}{}^{*} n \textit{ then } \overline{dds}(n',g,d)$$

4. *n is a safe node if it is both DDS and CDS.*

$$\overline{safe}(n,g,d) \quad \equiv \quad \overline{dds}(n,g,d) \wedge \overline{cds}(n,g,d)$$

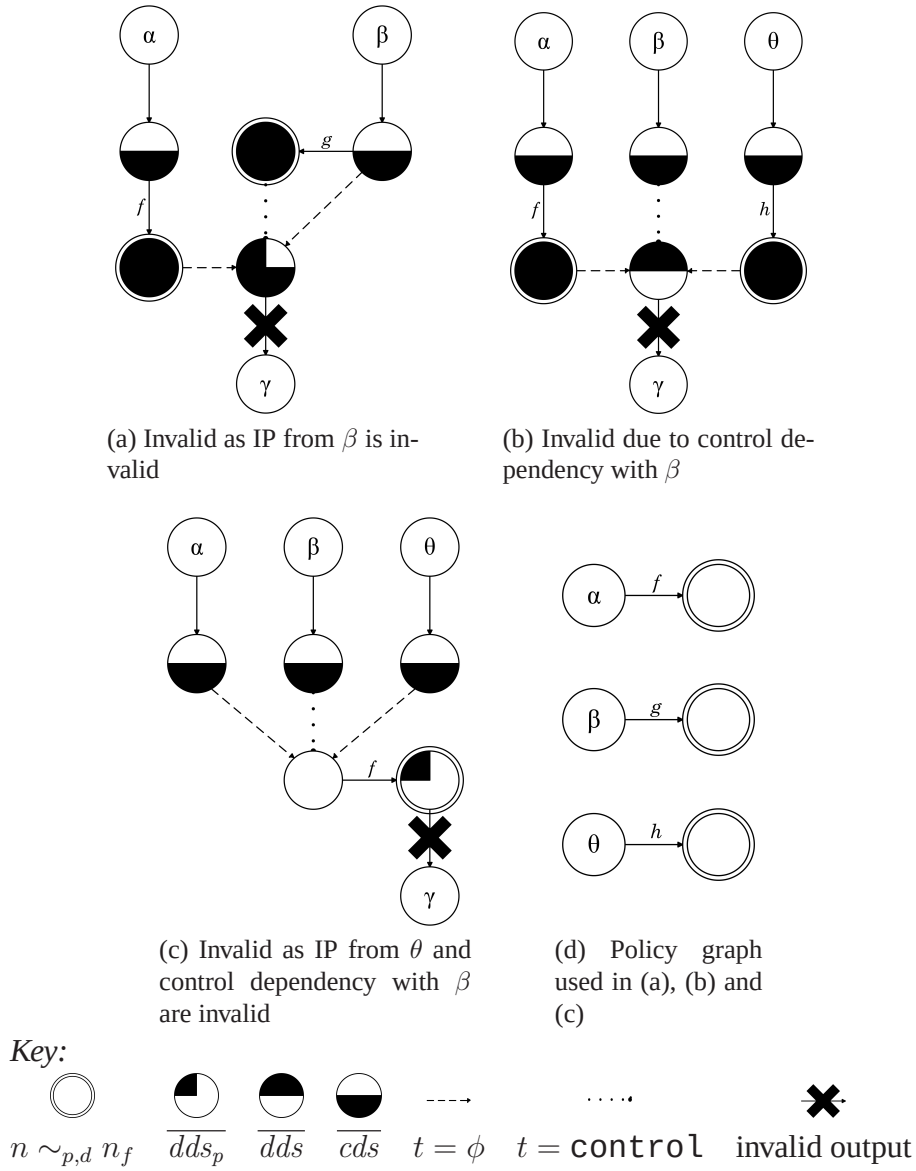
The following proposition on the CDS relation shows that a non CDS node makes all its descendants also non CDS. This will be needed for stating the soundness of the analysis mechanism.

Proposition 3.11. *Let n be a node in graph g and d be a policy graph. If n is not CDS with respect to d, then every node n' reached by n is also not CDS, i.e. $\forall n, n' \in nodes(g)$: if $n \xrightarrow{w}{}^{*} n'$ and $\neg\overline{cds}(n,g,d)$ then $\neg\overline{cds}(n',g,d)$.*

Finally, we can present the definition of a valid graph, which holds for a graph if all its outputs are safe.

Definition 3.12 (Graph Validity)**.** *We say that the expression graph g is valid with respect to a policy d if every node representing an output channel is a safe node. That is:*

$$\overline{valid}(g,d) \quad \equiv \quad \forall n_\gamma \in nodes(g) : \overline{safe}(n_\gamma, g, d)$$



(a) Invalid as IP from β is invalid

(b) Invalid due to control dependency with β

(c) Invalid as IP from θ and control dependency with β are invalid

(d) Policy graph used in (a), (b) and (c)

Figure 3.9: Examples of graph matching

We say g is d-valid if $\overline{valid}(g, d)$.

Figure 3.9 presents examples of the graph matching process. Here, unnecessary labels are not shown, for simplicity. Double-lined nodes match some final node of a policy, for some of its information paths. Note that for an output to be valid, all its information paths must be data dependency safe, as well as the output node be control dependency safe. Figure 3.9c shows how a node can simulate a policy final node, and yet not be completely safe. In this case, simulation only validates one of the two information paths of the node, and there is also an unresolved control dependency.

With the matching mechanism defined, we can present the soundness theorem. It states that if a node n in the graph simulates a node n_d in the policy graph, then the set of

expressions possibly held by n is a subset of the set held by n_d in the policy. The proof can be found in the Appendix.

Theorem 3.13 (Soundness of the matching mechanism)**.** *For a program's expression graph g and a policy d, the following relation holds:*

$$\forall n \in nodes(g), n_d \in nodes(d) : \textit{if } n \sim_{g,d} n_d \textit{ then } exp_g(n) \subseteq exp_d(n_d)$$

For the next theorem, we need to define the notion of a public expression, in terms of a declassification policy. The relation, analogous to Definition 2.6, is defined below.

Definition 3.14 (Public Expressions)**.** *Let d be a declassification policy graph, we say that the expression e is public according to d if the following relation holds:*

$$\begin{array}{rcl} \overline{public}(e,d) & \equiv & (\exists n_f \in fnodes(d) : e \in exp_d(n_f)) \textit{ or} \\ & & (e = f(e_1,\ldots,e_n) \wedge \overline{public}(e_1,d) \wedge \ldots \wedge \overline{public}(e_n,d)) \end{array}$$

With this, we can present the theorem of safety between process and policy, demonstrating that if the corresponding graph of a program satisfies a policy, then the expressions on the process will also satisfy it. This theorem is a consequence of Theorems 3.6 and 3.13 and it is important for making precise the link between the framework and the graph-based implementation.

Theorem 3.15 (Safety between process and policy)**.** *Let C_0 be a program, $g = G(C_0)$, π_0 be an environment, t be a run in $Run(C_0, \pi_0)$ and d be a policy graph. For any configuration $\langle C, \sigma, \pi \rangle \in t$, the following relations hold:*

$$\begin{array}{ll} (i) & \forall x \in \textbf{Var} : \textit{if } E_\sigma(x) \textit{ is defined and } \overline{dds}(n_x, g, d) \textit{ then } \overline{public}(E_\sigma(x), d) \\ (ii) & \forall \gamma \in \textbf{Out} : \textit{if } O_\sigma(\gamma) \textit{ is defined and } \overline{dds}(n_\gamma, g, d) \textit{ then } \forall e \in O_\sigma(\gamma) : \overline{public}(e, d) \\ (iii) & \forall \rho \in \textbf{Var} \cup \textbf{IO} : \textit{if } PC_\sigma(\rho) \textit{ is defined and } \overline{cds}(n_\rho, g, d) \\ & \qquad \textit{then } \forall e \in PC_\sigma(\rho) : \overline{public}(e, d) \end{array}$$

(i) states that if a variable in the program has its corresponding node in the graph (which is guaranteed to exist by Theorem 3.6) being data dependency safe, then the expression held by that variable in the process is public (i.e. allowed by the policy); (ii) states that if an output channel in the program has its corresponding node in the graph being data dependency safe, then all expressions sent to it in the process are public; finally, (iii) states that if a variable or I/O channel has a corresponding node in the graph being control dependency safe, then all conditional expressions of the variable (or I/O channel) in the process are public.

With this, we conclude the presentation of our graph-based implementation of the framework. In the next Section we prove that the validity of a program graph in the implementation implies the validity of the corresponding program, in the framework.

3.4 Soundness of the Implementation

General Setting. On Section 2.5 we have established the soundness of our framework. It consists of three fundamental elements: the program C, the set of declassifiable expressions D and the predicate $valid(C, D)$, and relates them to the PCR property, presented in Definition 2.14. Theorem 2.15 demonstrates that, if $valid(C, D)$ holds, then C satisfies the PCR property for the set of expressions D. We now demonstrate that the implementation presented in Sections 3.2 and 3.3 is actually sound, that is, the validation of a program graph implies the validation of the program used to create the graph. To do so we first need to give a formal definition of the framework itself.

Definition 3.16 (PCR Framework)**.** *A* PCR framework *is a triple $\langle Pol, Prog, valid\rangle$, in which:*

1. $Pol \subseteq \mathbf{Exp}\langle \mathbf{In} \times \mathbb{N}\rangle$ *is a set of declassifiable expressions.*
2. $Prog$ *is the domain of programs.*
3. $valid : Prog \times Pol \to Bool$ *is a predicate that satisfies the following:*

$$valid(C, D) \Rightarrow C \text{ satisfies the PCR property with respect to } D$$

In other words, it relates the previous two elements with the PCR property (Definition 2.14), as stated by Theorem 2.15.

Our next step is to relate our graph-based implementation to the expression-matching framework. For that, we first define what is an implementation of the framework, in general terms. An implementation needs to have computable approximations of the non-computable elements of the PCR framework: the $valid$ predicate and the set of declassifiable expressions D. Besides that, an implementation might also include an abstraction of the program C. This notion is defined below.

Definition 3.17 (Implementation of a PCR Framework)**.** *We say that $\langle(\overline{Pol}, \mathbb{P}), (\overline{Prog}, \mathbb{C}), \mathbb{V}\rangle$ is an implementation of $\langle Pol, Prog, valid\rangle$ provided that:*

1. $\overline{Pol}$ *is a domain of declassification policies and $\mathbb{P}$ is a policy interpretation function $\mathbb{P} : \overline{Pol} \to Pol$, which takes a declassification policy and returns the set of declassifiable expressions represented by it.*
2. $\overline{Prog}$ *is a domain of abstractions of programs and $\mathbb{C}$ is a function $Prog \to \overline{Prog}$ that takes a program and returns a corresponding abstraction. For $C \in Prog$ and $\overline{states}(\mathbb{C}(C))$ being the set of possible program states that the abstraction can represent, we have that $\overline{states}(\mathbb{C}(C)) \supseteq states(C)$.*
3. $\mathbb{V} : \overline{Prog} \times \overline{Pol} \to Bool$ *is a validation function which takes a program abstraction in $\overline{Prog}$ and a policy in $\overline{Pol}$ and returns a boolean. For $C \in Prog$ and $d \in \overline{Pol}$, the function $\mathbb{V}$ satisfies the following:*

$$\mathbb{V}(\mathbb{C}(C), d) \Rightarrow valid(C, \mathbb{P}(d))$$

Note that for an implementation to be tractable, then all elements on the left-hand side of the above formula should also be (i.e., $\mathbb{V}$, $\mathbb{C}$ and d).

Application of the General Setting. Now we proceed to present each of the elements of our specific graph-based implementation, and then to present the Theorem that shows that the graph-based analysis implements the PCR framework. We start by the policy interpretation function.

Definition 3.18 (Interpretation of Policy Expression Graphs)**.** *Let $d = (V, E, V_f, U)$ be a declassification policy, according to Definition 3.7. The policy interpretation function $int : \mathcal{D} \to \wp(\mathbf{Exp}\langle \mathbf{In} \times \mathbb{N} \rangle)$ is:*

$$int(d) = \bigcup_{n_f \in V_f} exp_d(n_f)$$

The interpretation function respects the notion of public expression, as shown by the following Lemma.

Lemma 3.19. *Let e be an expression, and d be a declassification policy. We have that: $\overline{public}(e, d) \Rightarrow public(e, int(d))$. In other words, if e is public according to d, then it is also public according to the set of declassifiable expressions represented by d.*

Proof. The proof is trivial, obtained by expanding Definitions 2.6 and 3.14 and combining them with Definition 3.18. □

Now we proceed to show that the program expression graphs are program abstractions.

Lemma 3.20 (Expression Graph as a Program Abstraction)**.** *Let C be a program. The program expression graph $G(C)$ is a program abstraction of C which represents the following program states:*

$$\overline{states}(G(C)) = \{\sigma \mid P(\sigma, G(C))\}$$

where $P(\sigma, G(C))$ is the one defined in Definition 3.5.

Proof. Proof is achieved by combining the program semantics with the definition of G to demonstrate that if $\sigma \in states(C)$ then $P(\sigma, G(C))$ holds. □

Finally, we state the Theorem that links the implementation to the framework:

Theorem 3.21. *The graph-based analysis defined as $\langle (\mathcal{D}, int), (\mathcal{G}, G), \overline{valid} \rangle$ implements the PCR framework $\langle Pol, Prog, valid \rangle$.*

The proof, which can be found in the Appendix, is achieved by expanding the elements of the equation $\overline{valid}(G(C), d) \Rightarrow valid(C, int(d))$. On the next section we proceed to show that this implementation is tractable.

3.5 Algorithms and Tractability

In this section we show the tractability of the graph-based implementation of the PCR framework. For this, we present simple algorithms for the graph matching mechanism and demonstrate that they have polynomial complexity. The algorithms presented here are not meant to be optimal in any sense, but rather a proof of the tractability of the analysis.

First, let us define our input size n. We wish to have a n that relates to the program's size. Thus, the natural approach is to consider n as the number of lines in the source code, which is equivalent to the number of non-compositional statements of a program C. For `if-then-else` and `while` commands, we naturally also count all the nested commands. For the analysis that follows it is also important to determine how the number of edges in the program's graph relates to n. We know that `if` and `while` commands do not generate edges, I/O operations and simple assignments generate each a single edge, and function assignments generate as many edges as function arguments. Thus, it is safe to define the number of edges as a factor of the number n of lines of code. We denote this factor c_e and state that a program has $c_e n$ edges. We can formally determine the value of c_e by the following:

$$c_e = 1 - \frac{|C_{\text{while}}| + |C_{\text{if}}| + |C_{\text{skip}}|}{n} + \sum_{C_f \in C_{\text{func}}} \frac{args(C_f) - 1}{n}$$

where $|C_{\text{while}}|$, $|C_{\text{if}}|$ and $|C_{\text{skip}}|$ are the number of `while`, `if` and `skip` commands in C, respectively. C_{func} is the set of function assignment commands in C and $args(C_f)$ returns the number of arguments in the function on the RHS of assignment command C_f.

Before discussing the matching mechanism, it is important to point the complexity of the graph-building process. From Figure 3.7 it is clear that function G has a complexity of $\mathcal{O}(n)$ for both space and time, as for every command in C, G generates a constant number of elements in the graph. We also consider that a pre-processing is done in the graph: nodes are previously checked for being inside of cycles in the graph, setting the $incycle(n)$ predicate for each node n. The algorithm for this pre-processing is omitted.

Now we proceed to build the algorithms. For simplicity, we split the analysis into 2 algorithms: one for calculating input uniqueness relations and other for validating the graph, which is also further divided into parts. It is clear that both algorithms can be merged, but the separated analysis is more clear. First, we present the algorithm for calculating input uniqueness relations. Algorithm 3.1 accepts a graph g as input and calculates, for every input α, the α-uniqueness relation on every node n. Recalling Definition 3.3 we know that, for the input uniqueness calculation, the graph edges that directly leave input channel nodes (e.g. $n_\alpha \to n$) have a special meaning: they represent the distinct accesses on that input channel. With that, each such edge represents a distinct input command on the program C. Thus, here we refer to these edges as *input edges*. Also, since some algorithms don't walk through `control` edges, we write (n, n', t_d, u) for an edge whose type t_d is any but `control`.

The algorithm has a simple approach: it initially considers that every node n is α-unique, for every input channel α. Then, it walks through the graph, starting on the input

Algorithm 3.1: Input uniqueness calculation

function: MAIN (g)
init : $\forall n \in nodes(g), e, e' \in edges(g), \alpha \in \mathbf{In} : (\alpha, n) \in uni(g), \texttt{path}(n, e) = \emptyset, \texttt{visited}(e, e') = \texttt{visitedn}(e, \alpha) = \texttt{false}$
1 **foreach** $(\alpha, \texttt{in}) \in nodes(g)$ **do**
2 **foreach** $(n_\alpha, n, t_d, u) \in edges(g)$ **do**
3 UNI ($(n_\alpha, n, t_d, u), n, n_\alpha, u, g$);

function: UNI ($e = (n_\alpha, n, t_d, u), n', n_p, u', g$)
4 $\texttt{path}(n', e) := \texttt{path}(n_p, e) \cup n_p$;
5 **if** $\exists (n^1, n', f_i, u'), (n^2, n', f_j, u') \in edges(g) : \texttt{path}(n^{\{1,2\}}, e) \neq \emptyset, n' \notin \texttt{path}(n^{\{1,2\}}, e)$ **then**
6 NOTUNI (α, n', g);
7 **else if** $incycle(n') \wedge u' \neq u$ **then** NOTUNI (α, n', g);
8 **else**
9 **foreach** $(n', n'', t'_d, u'') \in edges(g) : \neg\texttt{visited}((n', n'', t'_d, u''), e)$ **do**
10 $\texttt{visited}((n', n'', t'_d, u''), e) = \texttt{true}$;
11 UNI (e, n'', n', u'', g);

function: NOTUNI (α, n, g)
12 $uni(g) := uni(g) \setminus (\alpha, n)$;
13 **foreach** $(n, n', t_d, u) \in edges(g) : \neg\texttt{visitedn}((n, n', t_d, u), \alpha)$ **do**
14 $\texttt{visitedn}((n, n', t_d, u), \alpha) = \texttt{true}$;
15 NOTUNI (α, n', g);

channel nodes, verifying if any node does not satisfy α-uniqueness for the α input channel from which the current walk began. Function `MAIN` initializes the structures and launches each of the walks, starting on each input edge. Each walk is performed by the recursive function `UNI`. This function has 5 arguments: the input edge $e = (n_\alpha, n, t_d, u)$ for which input uniqueness will be checked, the current node n' being checked, both the parent node n_p and the control context u' from which n' was reached and the graph g. Upon entering, the function first checks if the current node does not satisfy input uniqueness for the input edge being analyzed. The two first conditions of Definition 3.3 are checked. The first condition is checked through the use of structure $\texttt{path}(n', e)$, which records, for every input edge, the path of nodes used to reach node n'. Thus, the first `if` command checks if the current node has two different parent nodes that reach it through a function (f) and both are also reached by the input edge e. This, along with n' not being on the path of any of the parents imply that condition 1 of the Definition will be found on the node that performs the join of two expressions on the same input. The second condition is checked in the following: if the current node n' is in a cycle, condition 2 is satisfied if the looping contexts of the cycle and the input edge differ. If any of these conditions is met, then node is not α-unique and function `NOTUNI` is called. That function marks an argument node n as not input unique for the argument input channel and also does so recursively for every node reached by n, thus satisfying condition 3 of the definition. If the conditions are not met, node n' is still α-unique. Thus, the analysis proceeds to the child nodes of n'. Here, the structures `visited` and `visitedn` are used to keep track of which graph edges have already been visited for that input edge analysis and for the propagation of non-α-uniqueness, respectively. This is necessary to avoid infinite computation due to the presence of cycles in the graph.

Now we proceed to analyze the complexity of this algorithm. We use $\mathcal{C}(\cdot)$ to express the worst-case time complexity of computation $\cdot$, which can be a function or operator. First, we know that function `MAIN` makes a number of comparisons equal to the number of input edges in the graph. This number is the same as the number of input commands in the code. Since this is a fraction of n, we denote it $c_i n$. Thus, we have that $\mathcal{C}(\texttt{MAIN}) = c_i n \cdot \mathcal{C}(\texttt{UNI})$. If we make the worst-case assumption that every node is reached by every input channel, we can analyze the algorithm to conclude that, for each edge in the graph, there will be a call to either `UNI` or `NOTUNI`, for each input edge walk. To be more precise, for a fraction of the edges in g, `UNI` will make a comparison on all the parent nodes of the current node and call `NOTUNI`. For another fraction, it will also make the comparison on the parent nodes, then the constant time comparison of the `else if`, and then call `NOTUNI`. For a third fraction, the algorithm makes the parent nodes comparison, then the constant one, and finally executes the `else` branch. The remaining fraction are the edges reachable by the ones that triggered the `NOTUNI` call: these perform only the computation of `NOTUNI`. Here, we call c_{in} the average indegree of nodes in the graph. So, if we name these fractions c_1, c_2, c_3 and c_4, we have that the cost of the whole computation for each input edge is, on average:

$$c_1 c_e n \cdot (c_{in} + \mathcal{C}(\texttt{NOTUNI})) + c_2 c_e n \cdot (c_{in} + 1 + \mathcal{C}(\texttt{NOTUNI})) + \\ c_3 c_e n \cdot (c_{in} + 1 + \mathcal{C}(\texttt{UNI.else})) + c_4 c_e n \cdot \mathcal{C}(\texttt{NOTUNI})$$

However, it is quite clear that $\mathcal{C}(\texttt{UNI.else}) = \mathcal{C}(\texttt{NOTUNI})$, since the loops are essentially the same. With this, we have:

$$\begin{aligned}\mathcal{C}(\texttt{UNI}) &= c_e n \cdot (c_1 c_{in} + c_1\mathcal{C}(\texttt{UNI.else}) + c_2 c_{in} + c_2 + c_2\mathcal{C}(\texttt{UNI.else}) \\ &\quad + c_3 c_{in} + c_3 + c_3\mathcal{C}(\texttt{UNI.else}) + c_4\mathcal{C}(\texttt{UNI.else})) \\ &= c_e n \cdot ((c_1 + c_2 + c_3 + c_4) \cdot \mathcal{C}(\texttt{UNI.else}) + (c_1 + c_2 + c_3) \cdot c_{in} + c_2 + c_3)\end{aligned}$$

But we know that $c_1 + c_2 + c_3 + c_4 = 1$ and we can also round its partial sums up to 1, to achieve:

$$\mathcal{C}(\texttt{UNI}) = c_e n \cdot (\mathcal{C}(\texttt{UNI.else}) + c_{in} + 1)$$

Now, the `else` branch of `UNI` (and also the main body of `NOTUNI`) makes a number of comparisons equal to the outdegree of the node being analyzed, and for each of these it makes an assignment and a recursive call. Thus, if we say that c_{out} is the average outdegree of nodes in the graph, we can say that, on the average of all executions, $\mathcal{C}(\texttt{UNI.else}) = \mathcal{C}(\texttt{NOTUNI}) = c_{out}$. Thus, we have that:

$$\begin{aligned}\mathcal{C}(\texttt{UNI}) &= c_e n \cdot (c_{out} + c_{in} + 1) \\ \mathcal{C}(\texttt{MAIN}) &= c_i n \cdot (c_e n \cdot (c_{out} + c_{in} + 1)) \\ &= n^2 c_i c_e \cdot (c_{out} + c_{in} + 1)\end{aligned}$$

We know that, by definition, c_i is a fraction that ranges from 0 to 1. Although c_e can be potentially infinite (as there are no bounds for the number of arguments a function takes), for it to have a value of order n the program would need to have, on average, each line of code with a function assignment in which the function takes n arguments. Since this is quite unrealistic, we can safely assume c_e to be of an order smaller than n. As for c_{in}, the same reasoning applies, as the average indegree of nodes in the graph is tightly related to the number of arguments functions take. Finally, a similar reasoning also applies for c_{out}: for it to have a value of order n, each variable on the program needs to be on the RHS of an assignment, on average, n times (notice that this condition happens together with the aforementioned condition for c_e). Since this is also unrealistic, we also assume c_{out} to be of an order below n. Thus, we have that our input uniqueness calculation algorithm has worst-case time complexity of $\mathcal{O}(n^2)$. As for space complexity, one can easily see that `path` can take $(n \cdot c_i n) \cdot n$ of space , `visited` takes $c_e n \cdot c_i n$ of space, `visitedn` also takes up to $c_e n \cdot c_i n$ (each channel accessed only once, having $c_i n$ channels), and $uni(g)$ can take up to $n \cdot c_i n$, thus giving the final space complexity of $n^3 c_i + n^2 c_e c_i + n^2 c_e c_i + n^2 c_i = n^3 c_i + n^2(2 c_e c_i + c_i)$, which is $\mathcal{O}(n^3)$.

We now proceed to the main graph validation algorithm. Algorithm 3.2 presents the mechanism. The main analysis loop is located within lines 1-6 of the `MAIN` function. For each output node in the graph, all the information paths from that node are calculated and the node is checked to be data dependency safe for each such information path. Finally, the node is checked to be control dependency safe. Should any of these checks fail, the function returns `false`. Here, we write `MIP` for a function call, and `mip` for a global data structure. Notice that the body of function `MIP` is presented and discussed further ahead.

Algorithm 3.2: Program expression graph validation

function: MAIN (g, d)
init : $\forall e \in edges(g), n \in nodes(g), p \in \mathcal{G}, \gamma \in$ **Out** : visitdds$(e, p) =$ visitcds$(e, \gamma) =$ false, color$(n) =$ white, cycle$(n) =$ false
1 **foreach** $(\gamma,$ out$) \in nodes(g)$ **do**
2 MIP(n_γ, g);
3 **foreach** $p \in$ mip(n_γ) **do**
4 **if** $\neg$DDS(n_γ, p, d) **then return** false;
5 **if** $\neg$CDS(n_γ, g, d) **then return** false;
6 **return** true;

function: DDS (n, p, d)
7 /* dds(n, p, d) is set with return value of this call */
8 **foreach** $n_f \in fnodes(d)$ **do**
9 **if** SIM(n, p, n_f, d) **then return** true;
10 **if** $type(n) =$ in **then return** false;
11 **foreach** $(n', n, t, u) \in edges(p) : \neg$visitdds$((n', n, t, u), p)$ **do**
12 visitdds$((n', n, t, u), p) =$ true;
13 **if** $\neg$DDS(n', p, d) **then return** false;
14 **return** true;

function: CDS (n, g, d)
15 **foreach** $(n', n, t, u) \in edges(g) : \neg$visitcds$((n', n, t, u), \gamma)$ **do**
16 visitcds$((n', n, t, u), \gamma) =$ true;
17 **if** $\neg$CDS(n', g, d) **then return** false;
18 **if** $t =$ control **then**
19 **foreach** $p \in$ mip(n') **do**
20 **if** $\neg$dds(n', p, d) **then return** false;
21 **return** true;

We will first analyse functions DDS and CDS. Both functions walk through the graph backwards, checking for dds and cds. In the case of DDS, a walk is made for each information path p. The first thing it does is to check if the current node n simulates any final node of the policy d, returning `true` if successful. For this it uses the SIM function, which is left unspecified. Recalling our definition of simulation, it has 3 clauses: the first consists of a simple node similarity check (check of types and labels), the third is a check for input uniqueness, which uses the information pre-computed by the previous algorithm, and finally the second clause consists of a slight variation of the classical definition of weak simulation in automata. Ours is sufficiently similar to the classical definition such that algorithms for the latter can be used in the former. Thus, from [Li09, AI08] we know that such algorithm exists and is P-complete. If node n does not simulate any policy final node, then the check continues by analysing its parents. If n is an input node, that means the current walk in the graph went from output to input without any match happening. That means the node is not dds and `false` is returned. Otherwise, the parent nodes are analysed recursively, and node n will be deemed dds if all its parents are dds as well. Structure `visitdds` is used to keep track of visited edges, due to the cycles on the graph. Function CDS works in a very similar way, except that it works on the graph itself, and not on information paths. It walks the graph backwards deeming a node cds if all its parent nodes are also cds and additionally if its `control` edge parents are also dds for all their information paths. Note that it uses the buffered `dds` structure computed during the calls to DDS and that nodes with no parents are always cds.

The complexity of the main function involves three parallel checks. In order to determine it, we first define the number of output nodes in the graph as $c_o n$ (with c_o being analogous to c_i in the previous analysis). Also, we need to define the number of information paths a node can have. We know that a node has a number of information paths equal to one plus the number of ϕ edges that can reach it throughout the graph. Considering a worst-case scenario in which a node is reached by every other node in the graph, this number becomes the number of ϕ assignments in the program, which we denote as $c_\phi n$. Thus, we have the complexity of the main function as:

$$\mathcal{C}(\texttt{MAIN}) = c_o n \cdot (\mathcal{C}(\texttt{MIP}) + c_\phi n \cdot \mathcal{C}(\texttt{DDS}) + \mathcal{C}(\texttt{CDS}))$$

Now, let us check the complexities of DDS and CDS. If we make the worst-case assumption that every output node is reached by every other node, we can conclude that each of these functions will be recursively called for each edge in the graph (thus, $c_e n$ times). In DDS, we know that SIM is called a number of times equals to the number of final nodes in the policy. Here, we need to determine the size of the policy. Since this is quite arbitrary (a big system can have a whole library of policies) we will consider it to be another input size. Thus, we call m the number of nodes in the policy, $c_{e'} m$ the number of edges on it, and $c_f m$ the number of final nodes. From [Li09, AI08] we can safely assume the upper bound of $\mathcal{C}(\texttt{SIM})$ to be $\mathcal{O}(T \cdot S)$, where T is the number of transitions and S the number of states of the system. For this, our system consists of both the program and policy graphs. Thus, we consider $\mathcal{C}(\texttt{SIM}) = \mathcal{O}((c_e n + c_{e'} m) \cdot (n + m)) = \mathcal{O}(n^2 + m^2)$ (recalling the discussion about the order of c_e, which is also applied here for $c_{e'}$). After that, DDS makes a single comparison and then a number of comparisons equals to the

number of parent nodes of n. Using a similar reasoning for `CDS`, we know that each call to it visits c_{in} edges, on average, and for each makes a single comparison (that leads to the recursive call) and another comparison that, if true (always true in worst-case), causes it to loop for each information path of the parent node in question ($c_\phi n$, worst-case), also making a single comparison. Thus, we have:

$$\begin{aligned} \mathcal{C}(\texttt{DDS}) &= c_e n \cdot (c_f m \cdot \mathcal{O}(n^2 + m^2) + 1 + c_{in}) \\ \mathcal{C}(\texttt{CDS}) &= c_e n \cdot (c_{in} \cdot (1 + c_\phi n)) \end{aligned}$$

Now we proceed to the information path calculation function, represented in Algorithm 3.3. Here, the treatment of cycles in the graph is a bit more complicated. We use structure `color`(n) as follows: it returns `white` if node n has not yet been visited by `MIP`, it returns `black` if n has already been visited and its information paths are all known, and finally it returns `gray` if n is currently being visited, i.e. the function call has been made but not yet returned.

Upon entering the function, `color`(n) is set to `gray`, indicating that n is currently being analyzed. The function then performs a backwards walk on the graph. For each parent node of n it checks the color of that node. If it is white, then the recursive call is made. If it is black then the buffer `mip` is used, in order to avoid redundancy. Otherwise, the parent node is still being analyzed and this means a cycle has just been completed. In that case, the specific parent node on which the cycle was found is marked as a *cycle root*, via the structure `cycle`. Then no recursive call is made, to avoid a "livelock" from happening. After this, lines 7 and 8 basically add the parent edge and the parent node's information paths (except when a cycle was completed) to the current set, according to Definition 3.8 of function ip on Section 3.3.

After the first loop is done, current node n will have its information paths calculation done, unless it is inside a cycle. In that case, n will have all its information path components, with the exception of the `mip` of the cycle root node. Also, function `JOIN` is used to ensure the definition of *maximal* information paths: if a node is in a cycle, then all ϕ-edges that reach it within the control context of that cycle should be part of a same information path, ensuring that no information path is a subgraph of some other. The function is called for every $u \neq 0$ that reaches n. Then, if n is a cycle root, this means it has already accumulated all the cycle edges (of the cycle in which it is the root) in its `mip`, and all of its parents which are still marked as white can now be revisited: n is marked as black, and with this the cycle nodes will be able to add `mip`(n) to their collections. Then, the code from line 13 verifies if n itself will need to be revisited: this is true if any of its parents n' is not visited (i.e., is not colored black). If this is the case (which only happens for nodes which are within a cycle), n is marked back as being white, and function `MAKE_WHITE` is called, which recursively turns white every descendant of n which was previously marked black. The `MAKE_WHITE` function is required due to the presence of nested cycles: a node can be the cycle root of the inner cycle, but just a "regular" node of the outer cycle. In this case, the calculation of the inner cycle nodes' `mip`s will only be complete after the outer cycle has been treated. Note that in the presence of nested cycles one of the roots is treated first, and all the nodes within the other cycle return from their analysis being marked for revisit (white). Then, after one of the roots is marked as

Algorithm 3.3: Calculation of information paths

```
   function: MIP (n, g)
 1 color(n) := gray;
 2 foreach (n', n, t_d, u) ∈ edges(g) do
 3     add := ∅;
 4     if color(n') = white then add := MIP(n', g);
 5     else if color(n') = black then add := mip(n');
 6     else if color(n') = gray then cycle(n') := true;
 7     if t_d = f_i then mip(n) := mip(n) ⊗ (add ⊕ (n', n, t_d, u));
 8     else if t_d = τ then mip(n) := mip(n) ∪ (add ⊕ (n', n, t_d, u));
 9 if incycle(n) then foreach (n', n, t_d, u ≠ 0) ∈ edges(g) do JOIN(mip(n), u);
10 if cycle(n) then
11     color(n) := black;
12     foreach (n', n, t_d, u) ∈ edges(g) : color(n') = white do MIP(n', g);
13 if ∃(n', n, t_d, u) ∈ edges(g) : color(n') ≠ black then
14     color(n) := white;
15     foreach (n, n'', t_d, u) ∈ edges(g) : color(n'') = black do
16         MAKE_WHITE(n'');
17 else color(n) := black;
18 return mip(n);

   function: MAKE_WHITE (n)
19 color(n) = white;
20 foreach (n, n', t_d, u) ∈ edges(g) : color(n') = black do MAKE_WHITE(n');

   function: JOIN (mip, u)
21 foreach p, p' ∈ mip do
22     p := p ∪ {(n, n', t_d, u) | (n, n', t_d, u) ∈ p' \ p)};
23     p' := p' ∪ {(n, n', t_d, u) | (n, n', t_d, u) ∈ p \ p')};
24     if p = p' then mip := mip \ p';
```

visited, all nodes are revisited and the process is repeated for the other cycle nodes. There is clearly room for improvement on the efficiency of this algorithm, but this would result in making it more complex, which is not the aim here.

Figure 3.10 below shows in detail the node colouring steps of the `MIP` function for a graph with a simple cycle. Here, unnecessary labels are omitted, dashed lines represent ϕ-edges, and the small letter `c` in the center of a node represents that the node is a cycle root (i.e.`cycle` is `true` for that node).

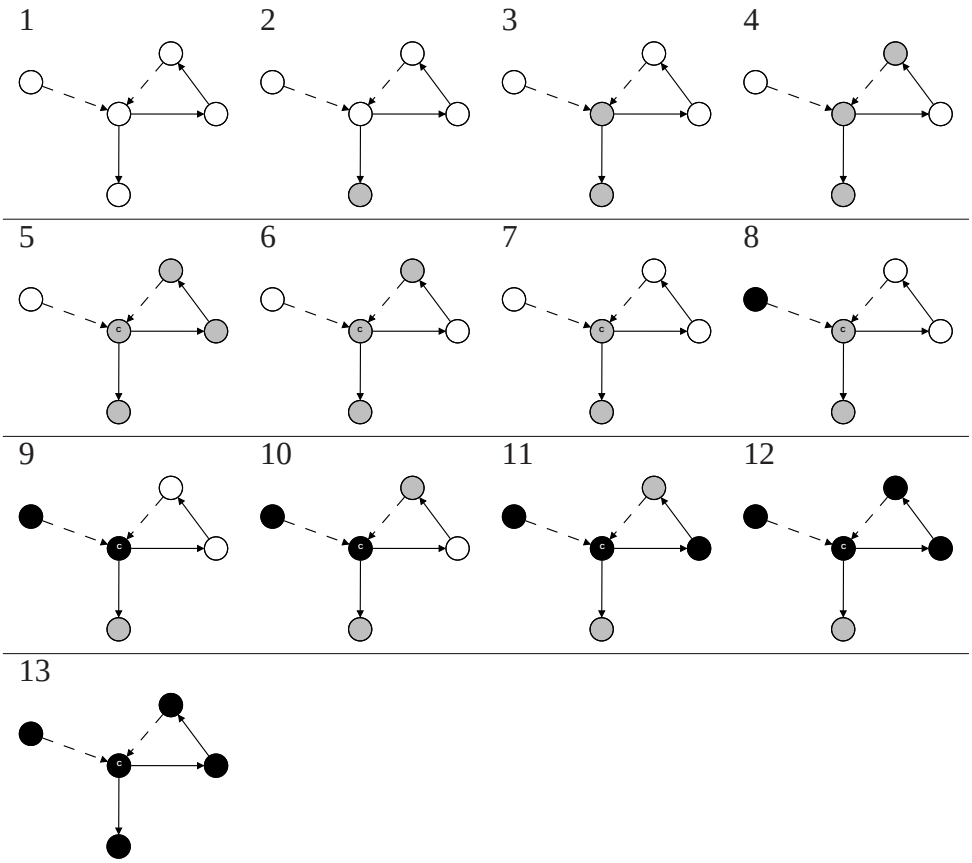


Figure 3.10: Algorithm 3.3 executing on a simple graph with a cycle

Figure 3.11 shows the node colouring steps for a more complex graph, with two nested cycles. For clarity, not all steps are shown. Note how `MAKE_WHITE` is called after step 17, making the two nodes in the nested loop white. This necessary since the `mip`s of these nodes still do not include the complete `mip` of the outermost cycle root – this is only possible after step 21.

For the complexity of `MIP` we first have to determine how many times the function will be recursively called. Assuming the worst-case scenario in which every output is reached by every other node in the graph we know that `MIP` reaches every node in the graph. However, some nodes are visited more than once: each node is visited once plus one more time for each distinct cycle it belongs to. However, due to the graph building rules plus the fact that the code is in SSA format, we know that the number of distinct cy-

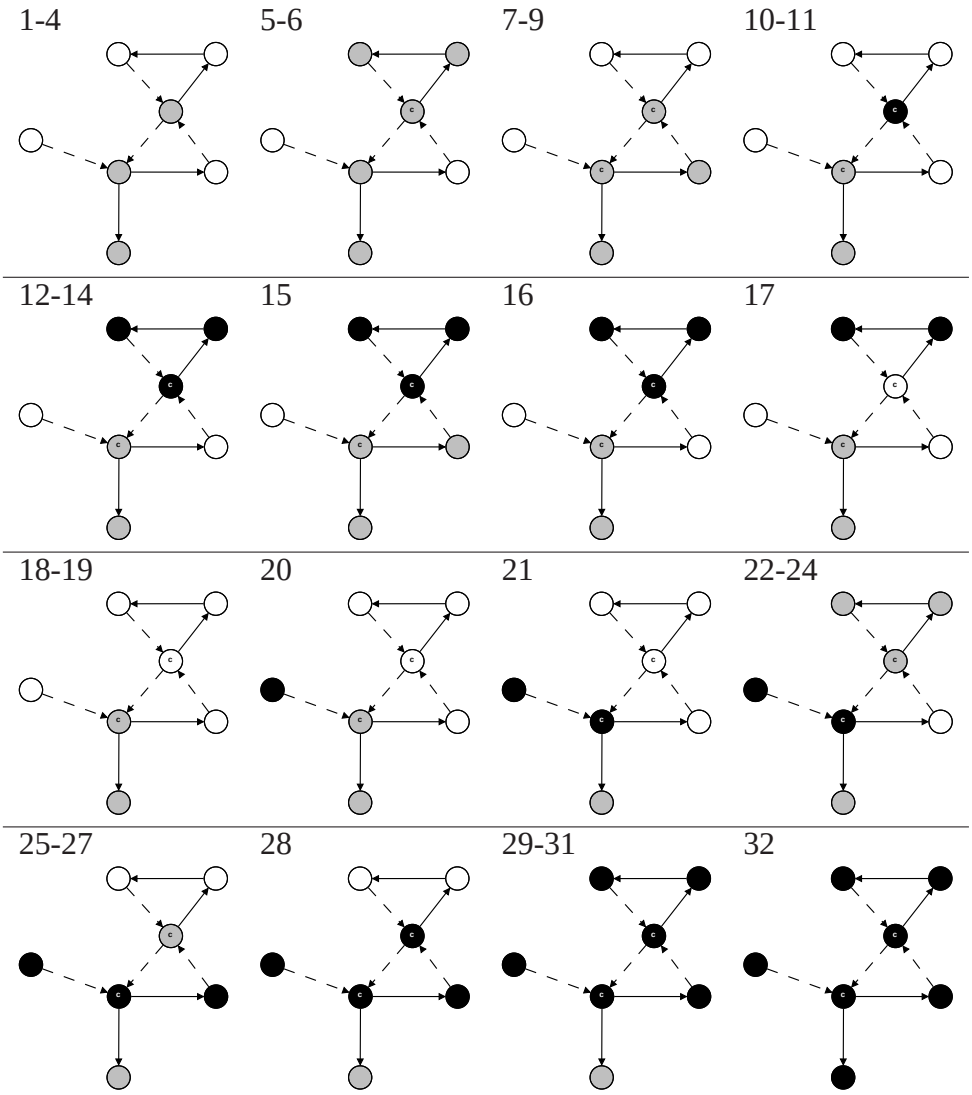


Figure 3.11: Algorithm 3.3 executing on a graph with nested cycles

cles a node can belong to is determined by the number of nested `while` commands within which the variable that represents the node is assigned in the code. Thus, simplifying for the worst-case scenario, this is proportional to the number of `while` commands in the program. Considering that every variable is assigned within all the loops in the code, we have that, in the worst-case scenario, `MIP` visits each node in the graph $c_w n$ times, with c_w being the ratio of `while` commands in n. Thus, the function is called $c_w n^2$ times.

Now we analyze the body of function `MIP`. We first have a loop that runs for c_{in} times and within which two comparisons are made. After the second comparison, a set operation is made with the information paths. Recalling that the maximum number of information paths a node can have is $c_\phi n$ and the definitions of the operators $\oplus$ and $\otimes$, we can simplify this operation to a worst-case complexity of $c_\phi^2 n^2$, which happens when the first `then` branch (line 7) is taken. Then, assuming a worst-case in which all `if` commands are

taken, we have a loop running c_{in} times and calling `JOIN`, then the comparison of line 10, resulting in a new loop running c_{in} times, then the comparison of line 13 which also runs c_{in} times, and cause a loop that runs c_{out} times, each of which making a call for `MAKE_WHITE`. Thus, we have:

$$\mathcal{C}(\texttt{MIP}) = c_w n^2 \cdot (c_{in} \cdot (1 + c_\phi^2 n^2) + c_{in} \cdot \mathcal{C}(\texttt{JOIN}) + c_{in} + c_{in} + c_{out} \cdot \mathcal{C}(\texttt{MAKE\_WHITE}))$$

Now, we check function `JOIN`. It takes every pair of information paths in argument mip and adds to each of them every edge with looping context equal to argument u that is present in one but not the other. Finally, if after this operation they are left identical, one of them is removed from the set. The loop analyzes a set of size $c_\phi n$, on average, and takes every distinct pair of it. Thus, it runs for $\frac{(c_\phi n)^2 - c_\phi n}{2}$ times. The set operations then need to inspect every element of each information path to be performed. In a worst-case scenario, information-paths are of size close to n. Thus, we consider each set operation with complexity $2n$. Finally, with the same assumptions, the last comparison runs at complexity n and results in another set operation of constant complexity. Then, we have:

$$\begin{aligned} \mathcal{C}(\texttt{JOIN}) &= \frac{c_\phi^2 n^2 - c_\phi n}{2} \cdot (2n + 2n + n) \\ &= \frac{5c_\phi^2 n^3 - 5c_\phi n^2}{2} \end{aligned}$$

Now, we calculate the average complexity of a call to function `MAKE_WHITE`. The function has a constant complexity, but it recursively turns black nodes into white ones. The worst case happens in a program where all commands are withing a same nested loop. In this case, each call to `MAKE_WHITE` will visit a number of nodes on the order of n. Then, we have that $\mathcal{C}(\texttt{MAKE\_WHITE}) = n$. And thus, making the substitutions for $\mathcal{C}(\texttt{JOIN})$ and $\mathcal{C}(\texttt{MAKE\_WHITE})$, we have:

$$\mathcal{C}(\texttt{MIP}) = c_w n^2 \cdot (c_{in} \cdot (1 + c_\phi^2 n^2) + c_{in} \cdot \frac{5c_\phi^2 n^3 - 5c_\phi n^2}{2} + 2c_{in} + c_{out} n)$$

In order to reach the final complexity of the matching mechanism, we first discuss about the constants. c_w, c_ϕ and c_f, like c_o and c_i, are ratios from 0 to 1. Also, for c_{in}, c_{out} and c_e the reasoning to consider them of an order lower than n applies. Thus, we have:

$$\begin{aligned} \mathcal{C}(\texttt{DDS}) &= c_e n \cdot (c_f m \cdot \mathcal{O}(n^2 + m^2) + 1 + c_{in}) \\ &= \mathcal{O}(n^3 m + n m^3) \\ \mathcal{C}(\texttt{CDS}) &= c_e n \cdot (c_{in} \cdot (1 + c_\phi n)) \\ &= \mathcal{O}(n^2) \\ \mathcal{C}(\texttt{MIP}) &= c_w n^2 \cdot (c_{in} \cdot (1 + c_\phi^2 n^2) + c_{in} \cdot \frac{5c_\phi^2 n^3 - 5c_\phi n^2}{2} + 2c_{in} + c_{out} n) \\ &= \mathcal{O}(n^5) \\ \mathcal{C}(\texttt{MAIN}) &= c_o n \cdot (\mathcal{C}(\texttt{MIP}) + c_\phi n \cdot \mathcal{C}(\texttt{DDS}) + \mathcal{C}(\texttt{CDS})) \\ &= c_o n \cdot \mathcal{O}(n^5) + c_o c_\phi n^2 \cdot \mathcal{O}(n^3 m + n m^3) + c_o n \cdot \mathcal{O}(n^2) \\ &= \mathcal{O}(n^6) + \mathcal{O}(n^5 m + n^3 m^3) + \mathcal{O}(n^3) \\ &= \begin{cases} \mathcal{O}(n^6) & \text{if } n \gg m \\ \mathcal{O}(n^6) & \text{if } n \approx m \\ \mathcal{O}(m^3) & \text{if } n \ll m \end{cases} \end{aligned}$$

As for the space complexity, we know that: `visitdds` takes up to $c_e n \cdot c_\phi n$ of space, `visitcds` takes $c_e n \cdot c_o n$, `color` and `cycle` both take n and `mip` takes $n \cdot c_\phi n$. Thus we have the final space complexity of $c_e c_\phi n^2 + c_e c_o n^2 + 2n + c_\phi n^2 = \mathcal{O}(n^2)$.

3.6 User-Defined Functions

In this section we present an extension to our toy language, adding user-defined functions. This extension is presented with updated versions of some definitions of chapters 2 and 3, but we omit changes in the theorems and proofs, since they are straightforward.

First, let us define how functions appear on programs. We update the language syntax to:

$$\begin{aligned} C ::= {} & \texttt{skip} \mid x := \alpha \mid \gamma := x \mid x := f(y_1, \ldots, y_k) \mid x := \phi_c(a, b) \mid C_1 \,;\, C_2 \\ & \mid \mathit{depends}(\theta, c) \mid \texttt{if}\ c\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{while}\ \overline{C}\ ;\ c\ \texttt{do}\ C \\ & \mid \texttt{def}\ F(v_1^F, \ldots, v_k^F)\ C \mid \texttt{return}\ r \mid x := F(y_1, \ldots, y_k) \end{aligned}$$

We use a capital F to distinguish user-defined functions from system functions (f). Also, we use $\mathit{body}(F)$ to return the command C which is the body of function F and, conversely, $\mathit{func}(C)$ returns the name of the function in which C is located in the code. In order to make definitions simpler, we also consider the following: (1) user-defined functions have unique names (no overloading); (2) input/output operations are not allowed inside a function; (3) functions only have access to variables declared within their bodies, plus the arguments (i.e. no global variables); and (4) for each function F there is a number of variables with fixed names, defined as follows: v_k^F represents the k-th argument of function F and r^F represents its return value. These variables are defined during function calls and returns, respectively. Note that (2) and (3) imply that user-defined functions have no side effects.

Now, we present additional semantics rules to treat user-defined functions. First, we add a new component $S \in \mathcal{S} = \mathit{Prog}^*$ to the program state σ, which represents the call stack. Operations $\mathit{push}(C, S)$ and $\mathit{pop}(S)$ return a new stack resulted from pushing a command C and popping the top element, respectively. Operation $\mathit{top}(S)$ returns the top element of S, but without removing it.

$$\langle \texttt{def}\ F(v_1^F, \ldots, v_k^F)\ C, \sigma, \pi \rangle \xrightarrow{\tau} \langle \texttt{skip}, \sigma, \pi \rangle \qquad \textit{(Def.)}$$

$$\langle x := F(y_1, \ldots, y_k)\ ;\ C_n, \sigma, \pi \rangle \xrightarrow{\tau} \langle C'\ ;\ C^F, \sigma', \pi \rangle \qquad \textit{(Call)}$$
$$\begin{aligned} \text{where} \quad C' &= v_1^F := y_1\ ;\ \ldots\ ;\ v_k^F := y_k; \\ C^F &= \mathit{body}(F) \\ S_{\sigma'} &= \mathit{push}(x := r^F\ ;\ C_n, S_\sigma) \end{aligned}$$

$$\langle \texttt{return}\ r, \sigma, \pi \rangle \xrightarrow{\tau} \langle r^F := r\ ;\ \mathit{top}(S_\sigma), \sigma', \pi \rangle \textit{(Return)}$$
$$\begin{aligned} \text{where} \quad S_{\sigma'} &= \mathit{pop}(S_\sigma) \\ F &= \mathit{func}(\texttt{return}\ r) \end{aligned}$$

Now we proceed to define how the program expression graph handles the user-defined functions. First, there are two new types of edges, $\mathtt{C}_\star$ and $\mathtt{R}_\star$, which represent function call and return, respectively. The $\star$ subscript is a unique identifier, in order to represent distinct calls of a same function. As for the looping context annotations, commands within a function but not inside any looping block will be executed in the looping context of the calling command. Thus, the edges associated to these commands are annotated with the function name, which serves to represent the situation just described. Note that, with this, the domain of looping context annotations is extended to $\mathbb{N} \cup \mathbf{Func}$, where **Func** is the domain of user-defined function names. The additional rules for the G function are presented below.

$$\begin{aligned} G_u(\mathtt{def}\ F(v_1^F, \ldots, v_k^F)\ C) &= G_F(C) \\ G_u(x := F(y_1, \ldots, y_k)) &= n_{y_1} \xrightarrow[u]{\mathtt{C}_\star} n_{v_1^F}, \ldots, n_{y_k} \xrightarrow[u]{\mathtt{C}_\star} n_{v_k^F}, n_{r^F} \xrightarrow[u]{\mathtt{R}_\star} n_x \\ &\quad \text{where } \star = \mathit{fresh}() \\ G_u(\mathtt{return}\ r) &= n_r \xrightarrow[u]{\mathtt{plain}} n_{r^F} \end{aligned}$$

With this, an adjustment on the graph notation must be made, with the definition of valid paths.

Definition 3.22 (Path validity)**.** *A path w on an expression graph is said to be valid according to the following grammar:*

$$\begin{aligned} w &::= w_s \mid w_c \mid w_r \\ t &::= \mathtt{plain} \mid \phi_i \mid f_i \\ w_s &::= t w_s \mid \mathtt{C}_\star w_s \mathtt{R}_\star w_s \mid \epsilon \\ w_c &::= \mathtt{C}_\star w_c \mid w_s w_c \mid \epsilon \\ w_r &::= \mathtt{R}_\star w_r \mid w_s w_r \mid \epsilon \end{aligned}$$

In other words, the notation $\xrightarrow[u]{w}{}^*$, $\xrightarrow{w}{}^*$ and $\rightarrow^*$ only holds for paths defined by this grammar.

In the above definition, w_s represents paths which start and end outside of functions, with call edges being eventually followed by their corresponding return edges. Paths w_c are the ones that start outside and end inside a function, while w_r are the opposite case, of paths that start inside and end outside of function calls. Note that if w_c and w_r were combined in a same path, this would result in, e.g. a path that takes edge $\mathtt{C}_1$ and then $\mathtt{R}_2$, which not valid.

Now the input uniqueness definition must account for edges labeled with function names as control context. This can be accomplished by a slight adjustment on the second item of Definition 3.3, changed to $n_\alpha \rightarrow^* \xrightarrow[u]{t_d} \rightarrow^* n \rightarrow^* \xrightarrow[v]{t'_d} \rightarrow^* n, u \neq v, v \in \mathbb{N}$. Here, $v \in \mathbb{N}$ means that v is not a function name F. With this, user-defined functions called within loops do not harm input uniqueness calculation.

Next step is updating the definition of exp. The function needs to keep track of which function call was last made (the $\star$ subscript). For this, we change it to $exp^\star_g$, where $\star$

represents this property, and is omitted when it is undefined (no function call was made). Thus, all current equations within the definition of exp are adjusted so that the $\star$ is also passed to recursive calls to it. The formula for n_x, however, is extended:

$$exp^{\star}_{g}(n_x) \quad = \quad \Psi_{n_x}(Exp)$$

in which

$$\begin{aligned} Exp = \; & \{f(e^1,\ldots,e^k) \mid \forall n^1,\ldots,n^k : n^i \xrightarrow{f_i} n_x \wedge e^i \in exp^{\star}_{g}(n^i)\} \\ & \cup \bigcup_{n' \xrightarrow{\tau} n_x} exp^{\star}_{g}(n') \\ & \cup exp^{\star'}_{g}(n') \quad (\text{if } n' \xrightarrow{\mathtt{R}_{\star'}} n_x) \\ & \cup exp^{\star}_{g}(n') \quad (\text{if } n' \xrightarrow{\mathtt{C}_{\star}} n_x) \end{aligned}$$

Notice that two new clauses are added: one for function returns and other for calls. Function return edges are always taken, updating the $\star$. Call edges are only taken if they represent the same return edge previously taken (as the graph is traversed "backwards", by following this definition return edges are taken before call edges).

In a similar fashion, $cexp$ is also updated:

$$\begin{aligned} cexp^{\star}_{g}(n) \quad = \quad & \bigcup_{n' \xrightarrow{t} n} cexp^{\star}_{g}(n') \cup \bigcup_{n' \xrightarrow{\mathtt{control}} n} exp^{\star}_{g}(n') \\ & \cup cexp^{\star'}_{g}(n') \quad (\text{if } n' \xrightarrow{\mathtt{R}_{\star'}} n_x) \\ & \cup cexp^{\star}_{g}(n') \quad (\text{if } n' \xrightarrow{\mathtt{C}_{\star}} n_x) \\ & \text{where } t \neq \{\mathtt{C},\mathtt{R}\} \end{aligned}$$

Finally, we can update the matching mechanism. Fortunately, with the $\star$ notation we can make the main change in the calculation of the information paths, leaving the rest of the matching process unmodified. The change is equivalent to the ones in exp and $cexp$:

$$ip^{\star}_{g}(n) \quad = \quad \begin{cases} \{\emptyset\} & \text{if } \not\rightarrow n \\ \bigcup\limits_{\substack{n' \xrightarrow{\tau} n \\ u}} ip^{\star}_{g}(n') \oplus (n', n, \tau, u) & \text{if } n \text{ is reached by one or more } \tau \text{ edges} \\ \bigotimes\limits_{\substack{n' \xrightarrow{f_i} n \\ u}} ip^{\star}_{g}(n') \oplus (n', n, f_i, u) & \text{if } n \text{ is reached by one or more } f_i \text{ edges} \\ ip^{\star'}_{g}(n') \oplus (n', n, \tau, u) & \text{if } n \text{ is reached by } n' \xrightarrow[u]{\mathtt{R}_{\star'}} n, \text{ for any } \star' \\ ip^{\star}_{g}(n') \oplus (n', n, \tau, u) & \text{if } n \text{ is reached by } n' \xrightarrow[u]{\mathtt{C}_{\star}} n \end{cases}$$

With this, the definition for the matching remains unchanged, as the calculation of information paths already handles C and R edges, turning them into τ edges. It is also straightforward how to change the algorithms of the last section to support the user-defined functions. Finally, since they are treated both in the semantics and on the graphs in a similar fashion to function inlining, including the functions on the theorems and proofs throughout this thesis is trivial. Notice that the definitions for both program and graph validity remain unchanged.

CHAPTER 4

Hybrid Static-Runtime Enforcer

In this chapter we extend our graph-based PCR approach in order to combine it with a runtime enforcement mechanism. With this, we present a practical hybrid static-runtime enforcer that is able to support policies that need both static and runtime information. We modify the static analyzer so that it generates a kind of report after the code analysis, in such a way that analysis is system independent. We define an intermediate step, between static analysis and runtime enforcement, named pre-load check, that translates the report from the previous step into the specific security labels of the target system, and then generates a checklist of conditions that need runtime information to be satisfied. Finally, we define a lightweight runtime enforcer which performs the checks only in the program points where they are necessary. Calls to the enforcer are injected in the application's code, prior to its execution, on the specific points where checks are needed, thus further reducing the overhead of the enforcer. Since this mechanism is presented in an implementation-oriented fashion, in this chapter we use the Java programming language and the Android mobile platform as the target technologies for demonstrating the approach.

In the next sections we present the hybrid enforcer, first by presenting some motivating examples, all based on real mobile applications, in Section 4.1,then giving an overview of it in Section 4.2. We then present the modification on graph-based PCR in Section 4.3, define the pre-load checker in Section 4.4, and finally define the runtime enforcer, including its code injection and experiments to measure its overhead in Section 4.5.

The contents of this chapter are presented in the paper [RCEC11].

4.1 Motivating Examples

In this section we present three running examples that will be used throughout this chapter. The examples are all within the context of mobile devices, and present problems which current popular mobile platforms (e.g. Android, Apple iOS) cannot handle; indeed, these problems cannot be handled by neither static nor runtime enforcement approaches, emphasizing the necessity for a combined approach. Since this chapter is implementation-oriented and is no longer necessary to discuss the details of graph-based PCR, the code of the examples is presented as standard Java-like algorithms, i.e. not pre-processed nor in SSA format.

Example 4.1 (Classification)**.** *Consider a policy that allows applications to read the contents of the phone's contact list, but not send it to low level channels (e.g., an arbitrary Internet connection). However, assume the user is allowed to mark as "trusted" certain output locations, such as a network connection, an SMS or an email address. Thus, information derived from the contact list can only be sent to trusted output channels. In this scenario, the static analyzer is needed to detect the flows of information within a program, while the runtime enforcer is needed to check the dynamic security label of the output channel. Algorithm 4.1 presents an example. In the following example algorithms we use underlined text to indicate input and output operations.*

Algorithm 4.1: Classification application

```
1 clist := getContactList();
2 counter := 0;
3 while hasNext(clist) do
4     contact := next(clist);
5     age := getAge(contact);
6     if age > 45 then counter := counter + 1;
7 text := "I have " + counter + " contacts over 45.";
8 addr := readFromInput();
9 sendSMS(addr, text);
```

Example 4.2 (Declassification)**.** *Consider a policy for location-based services. The policy states that a user's location is private in general and cannot be output. However, there are two allowed declassifications: (1) the timezone of a location, and (2) the result of a function that compares whether two locations are near to each other. In this scenario, an application can transmit its location to a different device using a secure connection. In particular, the application transmits data along with its corresponding security label to the other device (assuming that the underlying system platform supports this). Here, the static analyzer not only detects flows of information, but also points of the program that match the expressions allowed by the declassification policy. Again, the runtime enforcer checks for dynamic labels. See Algorithm 4.2 where isNear only works with arguments from a location input (such as a GPS).*

Example 4.3 (Iterative declassification)**.** *Now, consider a corporate application (Algorithm 4.3) in which a device accesses the records of several products, and it outputs the average of some property of the products (e.g. price, nutritional facts, cost, etc.). According to a declassification policy, the program can only output the average of a property for a given number of products (and not their single values). The static analyzer detects that the program conforms with the declassification policy, but the condition of the minimum amount of values the average has to contain is only checked during runtime.*

Algorithm 4.2: Declassification application

```
1 secureConn := secConnect("otherhost.somewhere.com");
2 myLoc := getLocation();
3 myTz := timezone(myLoc);
4 otherTz := recv(secureConn);
5 if myTz = otherTz then
6     send("ACK", secureConn);
7     otherLoc := recv(secureConn);
8     near := isNear(myLoc, otherLoc);
9     if near then print("Host is nearby!");
```

Algorithm 4.3: Iterative declassification application

```
1  sum := 0;
2  num := 0;
3  db := openDBConnection();
4  while !exitSignal do
5      rec := fetch(db);
6      prop := getProperty(rec);
7      sum := sum + prop;
8      num := num + 1;
9  avg := sum ÷ num;
10 output(avg);
```

4.2 Approach

Our approach consists of a hybrid static-runtime mechanism organized in three steps: static program analyzer, pre-load checker, and runtime enforcer. In practice, the first two steps perform most of the analysis, leaving the runtime enforcer to perform a few very precise (and thus efficient) checks. Figure 4.1 shows how the three steps interact with each other, while in the following we give an overview of their role.

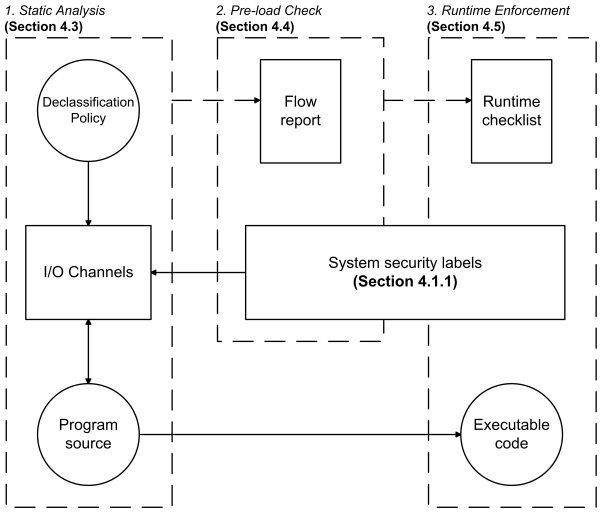


Figure 4.1: Overview of the 3-step enforcement

1. **Static analyzer:** it takes a *program* and identifies all its information flows, i.e. for each output operation, it identifies which input operations its value can potentially depend on (including implicit flows). Additionally, it takes a set of *declassification policies* and identifies which variables of the program hold expressions on inputs allowed by the policies. Thus, it downgrades the security level of those variables and of the corresponding flows of information. The information flows, combined with the matched declassifications, are included in a *flow report* of the program.

2. **Pre-load checker:** before the program is run, the checker takes the flow report from the previous step and checks the *security labels* of the system in which the program is about to run. The information flows with static labels are then validated at this step (i.e. `high` cannot flow to `low`). Flows containing I/O channels with dynamic security labels can only be checked at runtime, and thus are marked for checking in a *runtime checklist*. Also, declassifications from the previous step might have constraints associated with them, some of which may only be checked at runtime.

3. **Runtime enforcer:** the lightweight enforcer verifies that the conditions of the runtime checklist are satisfied at certain points of execution. The conditions may consist of checks of security labels of channels as they are accessed, and also of counting the number of times some loops in the program run. In order to reduce runtime

overhead, the calls to the enforcer are injected in the application bytecode, prior to the program's execution, on the specific program points that need checking.

4.2.1 Preliminary assumptions

Here we describe the assumptions made on the underlying system, which is part of the *Trusted Computing Base*. The considered programming language is assumed to have a well-defined set of I/O statements, which can be identified by the static analyzer. These I/O statements are "safe", in the sense that their behaviour is always the expected one. Also, functions referred by declassification policies (such as $timezone$, in Example 4.2) are also safe, meaning that they can not be abused or inverted in order to obtain the original value of its arguments. Policies using unsafe functions are considered malformed policies, and measuring the safety of a declassification policy is out of the scope of this thesis. We do not present user-defined functions in the examples of this chapter, but these can also be treated with the extension presented in Section 3.6.

The underlying system includes a security labeling system, and provides an API for handling the labels. We present the API, but leave its implementation unspecified, since this thesis focuses on the enforcement of policies by programs, rather than on the specification of a labeling system, a field with an already extensive published literature [Mye99, BWW08, SCH08]. Thus, the API is composed of:

- $getChannelLabel$ gets an I/O command and returns the security label of the associated channel. If values need to be known at runtime, whilst API is called before program's execution, the returned label is `runtime`.
- A special security label `data` is used to denote channels with a dynamic label, where each packet of data has a security label attached to it. For these kinds of channels:
  - $getDataLabel$ returns the label of a packet received from an input command.
  - $setDataLabel$ sets the label of a packet to be sent by an output command.
- $compareChannel$ takes two I/O commands and verifies if they access the same I/O channel of the system. It returns `OK` and `NO` in affirmative and negative cases, respectively, and `RT` (for *runtime*) in the case that the arguments' values need to be known at runtime, in order to perform the comparison.
- $maxLabel$ takes two labels and returns the most strict of them. If they are incomparable (at an equal level on the security lattice), returns a join of both.
- $\sqsubset$, $\sqsubseteq$, $\sqsupset$ and $\sqsupseteq$ are the comparison operators for labels.

4.3 Static Analyzer

The static analyzer is an adaptation of the graph-based PCR analysis that we introduced in the previous chapter. The modifications are motivated by the following:

- The static analyzer should now generate a report of all the information flows and declassification matchings within the program, rather than deciding by itself whether the program is safe or not.
- The analyzer should now be system independent with regards to the security labels, i.e.:
  - Analysis should be purely symbolic, i.e. information flows and declassification matching should be identified regardless of specific security labels, as these might differ depending on the system where the program will be executed. The interpretation of specific security labels is left for the other steps of the hybrid enforcement.
  - The matching between input nodes in program and policy should also be symbolic, in the sense that every input node in the former matches every input node in the latter. Again, the matching of input nodes is left for the other steps of the enforcement.

In this section, we first introduce some concepts and present the output of the static analyzer through the examples. Then, we proceed to define how we modify the PCR analysis to produce such an output.

Program point. A *program point* is used to identify an input (or output) operation in the program, and may be referenced by both the source and the compiled code. For each I/O operation detected by the static analyzer, a wrapper is generated around that operation, ensuring that the same program point used in the source will be recognized in the compiled code. We write θ^i to denote the *I/O operation* on channel θ at program point i. Note that this is different from the notation for sequential accesses on the channel, used in the previous chapters. While θ_i denotes the i-th access on channel θ, θ^i denotes the program statement at program point i.

Flow report. The static analyzer generates an output called *flow report*. It contains all flows of information in the program, one for each output operation in the program code, including declassification matchings. A flow is basically a relation between an output operation and a set of input operations whose values can influence it. While a formal definition of the flow report will be given at the end of this section, we first give an intuition of it via our examples.

Consider again the program of Example 4.1 (Classification): Table 4.1a presents the mappings of I/O statements to symbols (i.e. the Greek letters). The program points of the operations are also identified: here we use simply the line number where the statement takes place in the code. Table 4.1b presents the flow report. In particular, we have a single flow that states that output operation γ^9 (i.e., an output to channel γ that is made by the statement on program point 9) potentially reveals information about input operations α^4 and β^8.

To illustrate how the flow report can contain information about the declassification matchings detected on the code, let us consider again the program of Example 4.2 (Declassification), in which declassification policies are used. Table 4.2a summarizes the policy used, describing which expressions may be declassified. The top part of the table

Statement	I/O symbol	Type	Program point
$next(clist)$	α	Input	4
$readFromInput()$	β	Input	8
$sendSMS(addr, text)$	γ	Output	9

(a) Mappings of I/O channels to symbols

Flow report
$\{\alpha^4, \beta^8\} \rightsquigarrow \gamma^9$

(b) Flow report

Table 4.1: Static analyzer output for Example 4.1

identifies the expressions that can be declassified ($timezone(\ldots)$, $isNear(\ldots)$), their final nodes ($*_1$,$*_2$) and the new label to be applied (`low`) to the variables that hold these expressions. The input nodes (n_α, $*^{\texttt{in}}$) match specific input channels, identified in the bottom part of the table, where `any` stands for any input channel. The mapping between inputs and symbols works in the same way as in the previous example. Notice that, in this case (Table 4.2b), the same input channel β is accessed at two different points of the program (i.e. 4 and 7). The flow report (Table 4.2c) includes the declassifications detected by the static analyzer. For instance, $\{\alpha^2\} \mapsto^{X_1}$ `low` represents the case of variable $myTz$, set on line 3. This variable has a dependency with input α^2, but its content is matched by the declassification policy. Thus, its dependency with α^2 is changed to a dependency with the label of policy node $*_1$ (i.e. `low`). Finally, X_1 represents the set of constraints on the declassification policy that need to be checked in one of the next two phases (either at pre-load or at runtime). In other words, the expression $\{\alpha^2\} \mapsto^{X_1}$ `low` will eventually be translated to `low` if every constraint in X_1 is satisfied, and to α^2 otherwise.

Each element of a constraint set X is a pair with either one of two formats: (1) a pair (α^i, n_d), where the first element is an input operation in the code, and the second one is an input node in the policy, representing the constraint that α^i and n_d must represent the same input channel; or (2) a pair (i, exp), in which i is a program point and exp is an expression which represents a constraint on how many times program point i has to iterate on the running program.

Finally, we show the static analyzer output for Example 4.3 (Iterative declassification), showing the use of a loop counting declassification constraint. This policy is illustrated in Table 4.3a. Notice that the policy also enforces input uniqueness of the looping value, i.e. $(\alpha, *_1) \in uni(d)$. Since input uniqueness is entirely treated by the graph-based PCR static analyzer, we do not detail it here. The predicate $iter(*_1)$, related to the looping constraint, is discussed further ahead. The static analyzer works like in the previous examples (mappings in Table 4.3b and flow report in Table 4.3c).

Modifications on graph-based PCR. Here, we modify graph-based PCR to work together with the other two steps of our mechanism. As stated in the beginning of this section, the modifications are required since the analysis now must be system independent (i.e. labels for a same input can vary in different systems) and some security labels might be dynamic (i.e. their values being known only at runtime). Additionally, matching

Expression	Final node	Label
$timezone(n_\alpha)$	$*_1$	low
$isNear(n_\alpha, *^{\mathsf{in}})$	$*_2$	low

Input node	Input channel
n_α	$getLocation()$
$*^{\mathsf{in}}$	any

(a) Declassification policy

Statement	I/O symbol	Type	Program point
$getLocation$	α	Input	2
$recv(secureConn)$	β	Input	4
$send($"ACK"$, secureConn)$	δ	Output	6
$recv(secureConn)$	β	Input	7
$print($"Host is nearby!"$)$	γ	Output	9

(b) Mappings of I/O channels to symbols

Flow report
$\{\{\alpha^2\} \mapsto^{X_1} \mathtt{low}, \beta^4\} \rightsquigarrow \delta^6$
$\{\{\alpha^2\} \mapsto^{X_1} \mathtt{low}, \beta^4, \{\alpha^2, \beta^7\} \mapsto^{X_2} \mathtt{low}\} \rightsquigarrow \gamma^9$
$X_1 = \{(\alpha^2, n_\alpha)\}$
$X_2 = \{(\alpha^2, n_\alpha), (\beta^7, *^{\mathsf{in}})\}$

(c) Flow report

Table 4.2: Static analyzer output for Example 4.2

of declassification policies might include constraints that can only be checked at runtime. Thus, the analyzer no longer deems a program secure or not, but rather generates an output that will be used by the two subsequent steps.

We extend the policy graphs to accommodate the new constraints. Recall from Section 3.2 that the declassification policy graph has the form $d = (V, E, V_f, U)$, where V and E are the vertices (nodes) and edges, respectively, $V_f \subseteq V$ is the set of final nodes (also denoted by $fnodes(d)$), and U is the set of input uniqueness constraints. We make two modifications:

1. every input node n_d in the policy that matches an input node n_α in the program graph generates, upon matching, a constraint (α^i, n_d) for that policy matching, where i is the program point of the specific input operation that was matched, as the matching of the input channel will be made in one of the two subsequent enforcement steps;

2. we add another component, $iter$, which is a mapping from the policy nodes to expressions. When $iter(n_d)$ is defined, it returns a constraint on how many times the assignment of the variable that matches node n_d must iterate. The expression in $iter(n_d)$ has one free variable, named it, which represents the number of iterations.

Expression	Final node	Label
$(add(getProperty(n_\alpha)))^*$	$*_1$	`low`

Input node	Input channel
n_α	$fetch(openDBConnection())$

$iter(*_1) = it \geq 25$

(a) Declassification policy

Statement	I/O symbol	Type	Program point
$fetch(db)$	α	Input	5
$output(avg)$	γ	Output	10

(b) Mappings of I/O channels to symbols

Flow report
$\{\{\alpha^5\} \mapsto^{X_1} \texttt{low}\} \rightsquigarrow \gamma^{10}$
$X_1 = \{(\alpha^5, n_\alpha), (4, it \geq 25)\}$

(c) Flow report

Table 4.3: Static analyzer output for Example 4.3

Predicate $constr$ in Definition 4.1 defines how the constraints are generated.

With the notation defined, the flow report is generated from the program graph, via the process defined below. Here we use $id(C')$ to denote the program point of program statement C' and $label(n_f)$ to denote the security label of a policy final node n_f (i.e. the label to which a variable that matches that policy will be downgraded to).

Definition 4.1 (Flow report)**.** *Let C be a program, $g = G(C)$ the corresponding program graph and d a declassification policy graph. We have that:*

- *For a variable node n_x in g, the set of input operations and declassifications that potentially flow to n_x is defined as:*

$$flow_{g,d}(n_x) = \bigcup_{p \in mip(n_x)} flow_{p,d}(n_x)$$

where:

$$flow_{p,d}(n_x) = \begin{cases} \{\alpha_{id(x:=\alpha)}\} & \text{if } n_\alpha \to n_x \\ \{\bigcup_{n_y \to n_x} flow_{p,d}(n_y) \mapsto^X label(n_f)\} & \text{if } \exists n_f \in fnodes(d) : n_x \sim_{p,d} n_f \\ & \text{where } X = constr(n_x \sim_{p,d} n_f) \\ \bigcup_{n_y \to n_x} flow_{p,d}(n_y) & \text{otherwise} \end{cases}$$

- *For a simulation relation $\mathcal{R}$ between a node in g and a final node in d, the set of*

constraints for the matched declassification is defined as:

$$
\begin{aligned}
&constr(\mathcal{R}) = \\
&\{(\alpha_i, n_d) \mid (n_\alpha, n_d) \in \mathcal{R}, \exists n_x, n'_d : n_\alpha \rightarrow n_x, (n_x, n'_d) \in \mathcal{R}, i = id(x := \alpha)\} \cup \\
&\{(i, exp) \mid (n_x, n_d) \in \mathcal{R}, iter(n_d) = exp, \exists n_c : n_c \xrightarrow{\texttt{control}} n_x, \\
&\qquad i = id(\texttt{while}\ c\ \texttt{do})\}
\end{aligned}
$$

- *Finally, the flow report of program C is defined as:*

$$
\begin{aligned}
fr_d(C) &= \bigcup_{n_\gamma \in nodes(g)} fr_{g,d}(n_\gamma) \\
&\textit{where } g = G(C) \textit{ and:} \\
fr_{g,d}(n_\gamma) &= \\
&\{flow_{g,d}(n_x) \cup \bigcup_{n_c \xrightarrow{\texttt{control}} n_\gamma} flow_{g,d}(n_c) \rightsquigarrow \gamma_{id(\gamma:=x)} \mid n_x \in g : n_x \rightarrow n_\gamma\}
\end{aligned}
$$

Note that predicate $flow$ is a simple walk in the graph, whose implementation is straightforward. Algorithms from Section 3.5 can be extended to implement the flow report generation.

4.4 Pre-load Checker

The pre-load checker is the step responsible for matching the report generated by the static analyzer with the security labels of the specific system of execution. Each information flow from the flow report is checked by verifying the labels of the corresponding I/O channels. This is done by using the system labeling API. Flows containing only I/O channels with static labels are validated at this stage, while flows with dynamic labeled channels generate checks to be performed by the runtime enforcer. Also, the static analyzer's flow report may identify declassification matchings which contain additional constraints to be checked. The pre-load checker also verifies some of these constraints, and the ones that need runtime information are included in the runtime checklist.

In order to simplify definitions, two assumptions were made in our mechanism. First, we consider that every declassification that needs to be checked during runtime is *necessary*. That is, if the declassification constraints are not satisfied at runtime, then the program is marked as unsafe without further analysis. This also means that nested declassifications do not need to be checked, as the failure of the outermost one will result in stopping the execution. This assumption can be relaxed by extending our mechanism so that the runtime checklist contains information of what to do when a declassification fails: stop the program, still allow it or perform further checks, depending on the labels. In favor of clarity, we leave such extension for future work. Second, the flow report contains all inputs that an output can *possibly* depend on. In other words, our runtime enforcement is not permissive to the point of accepting safe executions of potentially unsafe programs.

Again, the mechanism can be extended so that runtime checks verify if unsafe branches are taken or not, but we leave this as future work.

The pre-load checker first translates the elements of the flow report to their corresponding labels in the target system. Each element in the runtime checklist has the format $(i, check)$, where i is a program point and $check$ a directive for a specific check to be performed. The possible directives are detailed in Table 4.4. The pre-load checker is defined at the end of this section (Definition 4.2), while we first give an intuition of its behaviour via our examples.

Name	Directive
`count_iter`	Count number of iterations of current command.
`eval(exp)`	Verify validity of expression `exp`.
`compare_ch(cmd)`	Checks whether the channel accessed by the current input command is the same as the channel relative to `cmd`.
`store_data_label`	Store data label of current input operation.
`store_ch_label`	Store the label of the input channel accessed by current input operation.
`check_input(pp)`	Check if label relative to input operation at program point `pp` is smaller or equal than that of the current output operation.
`set_data_label(label)`	Set the data label of the current output operation as `label`.
`check_output(label)`	Check if channel label of current output command is larger or equal than `label`.

Table 4.4: Runtime enforcer directives

In Example 4.1 (Classification), after the static analyzer does its job, the program is then compiled and the analyzer output is used by the pre-load checker just before the program is executed. The pre-loader translates each I/O operation to its corresponding label, as shown in Table 4.5a. However, notice that γ^9 translates to `runtime`, which means that its label can only be checked at runtime (as it depends on the value of variable $addr$). Based on this table, a checklist for the runtime enforcer is also generated, as shown in Table 4.5b. In this example, the checklist basically states that the label of the output statement of program point 9 needs to be checked and satisfy the constraint of being at least `hi`.

In Example 4.2 (Declassification), the pre-load checker must also check the policy constraints, which are all mappings between input operations on the code and the ones specified by the policy. These mapping can be checked entirely at this step, as shown in Table 4.6a. Table 4.6b shows the translation of the flows to labels. Notice that, although Definition 4.2 states that declassifications are translated to just the policy label when the check of the constraints does not fail, here we always show all the labels involved in

Flow	Labels
$\{\alpha^4, \beta^8\} \rightsquigarrow \gamma^9$	$\{\texttt{hi}, \texttt{low}\} \rightsquigarrow \texttt{runtime}$

(a) Label translations

Program point	Statement	Check condition
9	$sendSMS(\ldots)$	`check_output(hi)`

(b) Runtime checklist

Table 4.5: Pre-load checker output for Example 4.1

the declassification, for clarity. Recall that `data` stands for a label that is set for each transmission, as opposed to `runtime` (used in the previous example) which means that the whole channel has a single security label, which is known only at runtime. Finally, the runtime checklist is presented in Table 4.6c, with 3 items. The first tells the enforcer that the data label of input operation at program point 4 needs to be stored for further usage. Then, the second check treats the first flow: the output channel with automatic label must be labeled according to the inputs it depends on. Thus, the check is for the enforcer to assign a label to the data sent by that output operation, as the maximum label of all the inputs it can leak information on, i.e. the maximum between `low` and the data label of input at program point 4. Finally, the third check deals with the second flow: for output operation of program 9 to be safe, the label relative to input operation at program point 4, stored earlier, must be at most as strict as the label of the output command of program point 9 (which is `low`).

Constraint	Status
X_1:	
(α^2, n_α)	OK
X_2:	
(α^2, n_α)	OK
$(\beta^7, *^{\texttt{in}})$	OK

(a) Declassification constraints

Flow labels
$\{\{\texttt{high}\} \mapsto^{\texttt{OK}} \texttt{low}, \texttt{data}\} \rightsquigarrow \texttt{data}$
$\{\{\texttt{high}\} \mapsto^{\texttt{OK}} \texttt{low}, \texttt{data}, \{\texttt{high}, \texttt{data}\} \mapsto^{\texttt{OK}} \texttt{low}\} \rightsquigarrow \texttt{low}$

(b) Label translations

Program point	Statement	Check condition
4	$recv(\ldots)$	`store_data_label`
6	$send(\ldots)$	`set_data_label(max(in_label(4),low))`
9	$print(\ldots)$	`check_input(4)`

(c) Runtime checklist

Table 4.6: Pre-load checker output for Example 4.2

Finally, in Example 4.3 (Iterative declassification) notice that the static analyzer generates an iteration counting constraint for the declassification matching (Table 4.7a). The

second constraint denotes that the statement at program point 4 must iterate at least 25 times. Also, notice that this constraint cannot be verified at pre-load time, so the checker marks this as `RT`, meaning it needs to be checked at runtime. As for the checklist to be passed to the runtime enforcer, for the only flow to be safe, the declassification constraints must be all satisfied. Based on that, the checklist for the runtime enforcer has two items (Table 4.7c): a request for counting the number of times a loop will run, and then using that number to validate the output operation.

Constraint (X_1)	Status
(α^5, n_α)	`OK`
$(4, it \geq 25)$	`RT`

(a) Declassification constraints

Flow labels
$\{\{\texttt{high}\} \mapsto^{\texttt{RT}} \texttt{low}\} \rightsquigarrow \texttt{high}$

(b) Label translations

Program point	Statement	Check condition
4	`while ... do`	`count_iter`
10	$output(\dots)$	`eval(iter_count(4)` $\geq$ `25)`

(c) Runtime checklist

Table 4.7: Pre-load checker output for Example 4.3

In the definition below, we use f to denote a flow (from the flow report), dc to denote a declassification within a flow, l for a label and pf for a "partial flow": i.e. the left-hand side of a flow, consisting of a set of input statements and declassifications. We use $cmd(n_d)$ to return the command relative to policy node n_d (obtained from a simple lookup on the declassification policy mapping table). For a flow f, $from(f)$ and $to(f)$ return the left-hand side (set of inputs and declassifications) and right-hand side (output), respectively. The same applies for a declassification dc, with also $constr(dc)$ denoting its constraint set. Additionally, $label(\alpha^i)$ returns the label of an I/O operation, using the system labeling API (translating α^i to the corresponding I/O command). Consider that, when an input statement α^i of a flow f is translated to a label l, predicate $id(l)$ is set with the program point i of the input statement. Finally, in the definition of the checklist, `typewriter` font denotes enforcer directives (and thus treated as constant symbols), while standard mathematical notation denotes expressions which are actually evaluated. Function max is used over labels in the following way. If the set of input labels only contain static labels (e.g. `low`, `high`), it evaluates to the most strict label (`high`). However, if the set includes a dynamic label (`data` or `runtime`), it evaluates to `max(in_label(n),m)`, to be evaluated by the runtime enforcer, where `n` is the program point of the dynamic label and `m` the max of the static labels.

Definition 4.2 (Pre-load checker)**.** *Let C be a program, d a declassification policy and $fr_d(C)$ be the flow report for program C using policy d. The pre-load checker is defined by the following steps:*

1. *For each flow $f \in fr_d(C)$, the translation of f to security labels is defined as:*

$$
\begin{array}{rcl}
lbl(f) & = & lbl(from(f)) \rightsquigarrow label(to(f)) \\
lbl(pf) & = & \bigcup_{\alpha_i \in pf} label(\alpha_i) \cup \bigcup_{dc \in pf} lbl(dc) \\
lbl(dc) & = & \begin{cases} lbl(from(dc)) & \text{if } check(constr(dc)) = \texttt{NO} \\ to(dc) & \text{if } check(constr(dc)) \in \{\texttt{OK}, \texttt{RT}\} \end{cases} \\
check(X) & = & max(\bigcup_{x \in X} check(x))
\end{array}
$$

where max orders values as: $\texttt{NO} > \texttt{RT} > \texttt{OK}$

$$
check(x) = \begin{cases} compareChannel(\alpha_i, cmd(n_d)) & \text{if } x = (\alpha_i, n_d) \\ \texttt{RT} & \text{if } x = (i, exp) \end{cases}
$$

where $compareChannel$ is the function from the system labeling API.

2. *The static validation of program C with policy d is defined as:*

$$
\begin{array}{rcl}
validate(C, d) & \equiv & \forall f \in fr_d(C) : validate(f) \\
validate(f) & \equiv & \forall l \in lbl(from(f)), l \notin \{\texttt{runtime}, \texttt{data}\} : l \sqsubseteq label(to(f))
\end{array}
$$

3. *For the non-statically verifiable labels and constraints of C and d, the runtime checklist is defined as:*

$$
\begin{array}{rcl}
checklist(C, d) & = & \{checklist(f) \mid f \in fr_d(C)\} \\
checklist(f) & = & \\
\multicolumn{3}{l}{\{(id(l), \texttt{store\_data\_label}), (id(cmd(to(f))), \texttt{check\_input}(id(l))) \mid l \in lbl(from(f)) : l = \texttt{data}\}} \\
\multicolumn{3}{l}{\cup \{(id(l), \texttt{store\_ch\_label}), (id(cmd(to(f))), \texttt{check\_input}(id(l))) \mid l \in lbl(from(f)) : l = \texttt{runtime}\}} \\
\multicolumn{3}{l}{\cup \{(id(cmd(to(f))), \texttt{set\_data\_label}(max(lbl(from(f))))) \mid label(to(f)) = \texttt{data}\}} \\
\multicolumn{3}{l}{\cup \{(id(cmd(to(f))), \texttt{check\_output}(max(lbl(from(f))))) \mid label(to(f)) = \texttt{runtime}\}} \\
\multicolumn{3}{l}{\cup \{(i, \texttt{count\_iter}), (id(cmd(to(f))), \texttt{eval}(exp[\texttt{iter\_count}(i)/it])) \mid (i, exp) \in constr(dc), dc \in from(f)\}} \\
\multicolumn{3}{l}{\cup \{(i, \texttt{compare\_ch}(cmd(n_d))) \mid (\alpha_i, n_d) \in constr(dc), dc \in from(f), check((\alpha_i, n_d)) = \texttt{RT}\}}
\end{array}
$$

About `data` *input channels inside loops.* Since our enforcement is not permissive (i.e. does not accept safe executions of possible unsafe programs), storing labels of `data` input channels inside loops can cause problems, as the label would be overwritten at every iteration of the loop. To solve this problem without permissiveness, our approach, when trying to store a pre-existing `data` label, replaces the stored one with a "most strict join" of both. This can lead to imprecise analysis (i.e. over strict), but only in some rare cases: when a `data` input channel is read within a loop, with only some of its values (the lower

labeled) being aggregated together (higher labeled ones being discarded), and then sent to an output. However, as previously explained, our approach can be extended to be permissive, ruling out this imprecision.

Algorithm. Algorithm 4.4 is a straightforward implementation of the pre-load checker. Here, consider that $fr(C)$ is the flow report for program C. Also, when a label l is obtained from $getChannelLabel(cmd)$, an entry is made on $id(l)$ representing the original program point that generated the label. For the entries of the runtime checklist ($chklst$), text in `typewriter` font represents the runtime enforcer directives, here treated as constant strings, whereas text in standard $math$ notation represents statements that are actually evaluated by the pre-loader algorithm. Function max appears with two different uses: on lines 13 and 18 it is used to calculate a maximum result for constraint checking, using the ordering `NO` $>$ `RT` $>$ `OK`; on lines 31 and 33 it is used over labels by the following: if the set of input labels only contain static labels (e.g. `low`, `high`), it evaluates to the most strict label (`high`); if, however, the set includes a dynamic label (`data` or `runtime`), it evaluates to `max(in_label(n),m)`, where `n` is the program point of the dynamic label and `m` the max of the static labels. In the latter case, `max` will be evaluated at runtime.

4.5 Runtime Enforcer

The runtime enforcer has a very simple behaviour. As the program is executed, each check on the checklist is performed as its corresponding program point is achieved. The runtime enforcer itself is a simple program, containing different functions for each type of check, and its own state-tracking variables. In this section we consider a Java-based runtime environment. Thus, our enforcer is a Java class with only *static* methods and parameters. Consequently, only a single instance of the enforcer is instantiated for a monitored program. Calls to the enforcer class are injected in the target application's bytecode, after the pre-load check, just before execution. Here we treat this code injection as a preliminary step to the runtime enforcement, although it can also be considered a final stage of the pre-load checker.

Code injection. The approach of injecting calls to the runtime enforcer in the application bytecode, just before execution, brings advantages for two reasons. First, it keeps the runtime enforcement stage with minimal overhead, as the injected code is a simple method call containing all information needed for that check. This precludes the need for the enforcer to monitor every single instruction, and to iterate over the different types of check. Second, it connects the program points calculated by the static analyzer (over the source code) with the information available to the runtime enforcer (which works on the bytecode).

The injection is simple: for each check at the runtime checklist, a call for the enforcer to perform such a check is added just before the corresponding program point. We demonstrate the process via an example: consider a Java implementation of Example 4.1, in Figure 4.2. Figure 4.3 shows a snippet of the corresponding `.dex` bytecode, compiled for Android's Dalvik virtual machine, already with the injected code, identified by the

Algorithm 4.4: Pre-load checker

```
1  chklst := ∅;
2  foreach f ∈ fr(C) do
3      fromLbl := ∅;
4      toLbl := getChannelLabel(cmd(to(f)));
5      foreach e ∈ from(f) do
6          if e ∈ (In × ℕ) then fromLbl ∪= getChannelLabel(cmd(e));
7          else if e ∈ Declass then
8              cmax := 0;
9              foreach x ∈ constr(e) do
10                 if x ≡ (i, exp) then
11                     chklst ∪= (i,count_iter);
12                     chklst ∪=
                       (id(to(f)),eval(replace(exp, it,iter_count(i))));
13                     cmax := max(cmax, RT);
14                 else if x ≡ (α_i, n_d) then
15                     c := compareChannel(cmd(α_i), cmd(n_d));
16                     if c = RT then
17                         chklst ∪= (i,compare_ch(cmd(n_d)));
18                     cmax := max(cmax, c);
19             if cmax ∈ {OK, RT} then fromLbl ∪= to(e);
20             else foreach α_i ∈ from(e) do
21                 fromLbl ∪= getChannelLabel(cmd(α_i));
22         foreach l ∈ fromLbl do
23             if l = data then
24                 chklst ∪= (id(l),store_data_label);
25                 chklst ∪= (id(to(f)),check_input(id(l)));
26             else if l = runtime then
27                 chklst ∪= (id(l),store_ch_label);
28                 chklst ∪= (id(to(f)),check_input(id(l)));
29             else if l ⊐ toLbl then return false;
30         if toLbl = data then
31             chklst ∪= (id(to(f)),set_data_label(max(fromLbl)));
32         else if toLbl = runtime then
33             chklst ∪= (id(to(f)),check_output(max(fromLbl)));
```

comment lines. Bold font is used to point the output instruction for which the check is needed. Note that in the "debug info" of the bytecode, it can be seen that bytecode address 0049 (recalculated from its original value, after the code injection) corresponds to program point 29, the program point where the output happens in the Java source code. Here, the check `check_output` of the enforcer is injected right before the output command. Since the code injection is a simple (and technology dependent) process, we omit a detailed specification of it.

```
...    ...
15:    static void processContactList() {
16:        String [] clist;
17:        String contact, text, addr;
18:        int counter, age;
19:        clist = getContactList();
20:        counter = 0;
21:        while(hasNext(clist)) {
22:            contact = getContact(clist);
23:            age = getAge(contact);
24:            if(age > 45)
25:                counter = counter + 1;
26:        }
27:        text = "I have " + counter + " contacts over age 45";
28:        addr = readFromInput();
29:        sendSMS(addr, text);
30:        System.out.println(text);
31:    }
```

Figure 4.2: A Java implementation for Example 4.1

The enforcer program. The enforcer provides a method for each check type. For each case, a statement is executed and its result validated. If the statement is not satisfied (i.e. expression does not hold, or command cannot be executed) then the enforcer halts the calling thread, and reports the violation. For the considered Java enforcer, each check is implemented by a method, e.g. `check_output(label)` is implemented by method $Enforcer.checkOutput(i, c, label)$, where i and c are arguments representing the current program point and command, respectively. For simplicity, we omit a detailed implementation of the enforcer, but the behaviour of each check is a straightforward implementation of Table 4.4, in the previous section.

4.5.1 Overhead

We have implemented our runtime enforcer in Java, and measured both its processing and memory overhead, running with applications on an Android device. First, we discuss the theoretical limits for this overhead, and then we proceed to show our experimental results. For the memory overhead, the enforcer keeps two buffers, $iter\_count$ and in_label,

```
                     |[36c] Example1.processContactList:()V
                     ...
0003e4: 7100 0900 0000|  0034: invoke-static {}, Example1.readFromInput:()Ljava
                     |        /lang/String; // method@0009
0003ea: 0c01         |  0037: move-result-object v1
\\ Begin injected code
0003ec: 1302 1500    |  0038: const/16 v2, #int 21 // #0015
                     ...
000408: 7140 0100 3254|  0046: invoke-static {v2, v3, v4, v5}, Enforcer.checkOu
                     |        tput:(ILjava/lang/String;[Ljava/lang/Object;Ljav
                     |        a/lang/String;)V // method@0001
\\ End injected code
00040e: 7120 0a00 0100|  0049: invoke-static {v1, v0}, Example1.sendSMS:(Ljava/
                     |        lang/String;Ljava/lang/String;)V // method@000a
000414: 6201 0000    |  004c: sget-object v1, java.lang.System.out:Ljava/io/Pr
                     |        intStream; // field@0000
000418: 6e20 0b00 0100|  004e: invoke-virtual {v1, v0}, java.io.PrintStream.pri
                     |        ntln:(Ljava/lang/String;)V // method@000b
00041e: 0e00         |  0051: return-void
                     |  debug info
                     |    line_start: 14
                     |    parameters_size: 0000
                     |    0000: prologue end
                     ...
                     |    0049: advance pc
                     |    0049: line 29
                     ...
```

Figure 4.3: Dalvik bytecode snippet for code of Figure 4.2

which map a program point to an integer and a label, respectively. These buffers can be implemented either with standard arrays or hash tables. Note that entries on each of the two buffers point to different types of commands: entries in $iter\_count$ point to looping and entries in in_label to input commands. So, a worst-case scenario happens on a program made entirely by loops and inputs, all loops being referenced by policies, all inputs being dynamic, and a single output in the end, with all inputs flowing to it. In this case, for a program with n commands, exact $n - 1$ entries are made on the buffers, each using one memory word (32 or 64-bit). Note that, in practice: (1) the average case tends to use considerable less memory, e.g. in our 3 examples, the ratios of (number entries/number commands) were 0/9, 1/9 and 1/10, respectively; and (2) programs tend to use much more memory for their data than for their code, meaning that the bound of n entries in the buffers is usually low.

As for the processing overhead, note that each injected code piece is a simple call to one of the enforcer's methods. These methods, in turn, are implemented with the execution and verification of a simple statement, with no loops. Thus, it is clear that the enforcer methods have, by themselves, constant complexity, and that the enforcer does not change the complexity of the monitored program. Once again, the number of checks added to the program is bounded by the number n of commands. But most practical cases do not reach the bound n, since only operations on dynamic I/O channels and declassification constraints generate checks. In our 3 examples, the ratios of (number checks/number commands) were 1/9, 3/9 and 2/10, respectively. It should be noted that, in the classical definition of a runtime execution monitor [Sch00], the runtime enforcer monitors *every*

command of the program. Our enforcer, though, does not necessarily need to monitor every instruction, since the task of identifying instructions that need monitoring is performed by the previous stages of our hybrid approach.

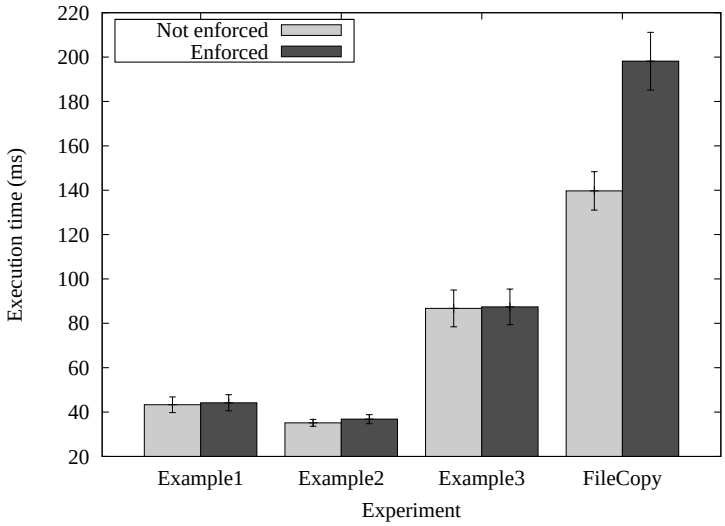
We have implemented Android versions of the three examples of this chapter, plus a number of benchmarking programs meant to stress the runtime enforcer performance. Unfortunately, there are only a few proposals for hybrid approaches in literature, and they all differ not only in how they are measured, but also on their specific goals. Thus, there is not yet a "standard benchmark" for hybrid static-runtime information flow and declassification analysis, making a direct comparison of performance with other approaches not possible at this moment. Our experiments have the purpose of showing that the overhead of our runtime component is negligible for most practical scenarios.

Each benchmark has a different "profile" for accessing I/O. *FileCopy* performs a copy between files, reading blocks of 1KB at a time. However, each block has a `data` security label. Thus, the runtime enforcer has to set the label of each write with the label from the previous read. This is an example of a program with extreme I/O access, all of which checked by the runtime enforcer. *FileEncrypt* is the same as the previous, but each block is encrypted before being written. With this, the program incurs a considerable processing time between I/O accesses. *InfGather* and *Statistics* are similar programs, which access inputs from 10 different sources, and then perform a single output, whose value depends on all previous inputs. In the former, all input channels have `runtime` security labels, which have to be checked during access, and then compared to the output label. In the latter, labels are static, but violate non-interference. However, some statistical calculation is done over the data, and a declassification policy allows such computation. Thus, the runtime enforcer is left to check if the input channels accessed by the program match the ones described by the policy, and also count the number of input accesses made by the main loop. Finally, *Loops* is a program made by several loops, all of which are small in size and have their number of iterations counted by the enforcer, presenting an extreme example of almost every instruction being checked. Java source code for these examples, as well as for the runtime enforcer implementation can be found in Appendix B.

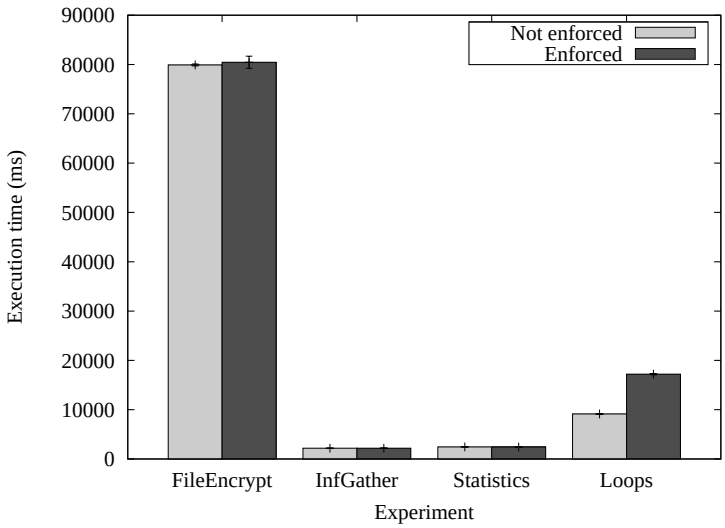
Each program was executed 50 times with and without the calls to the runtime enforcer, and their processing times and memory usage was observed. Figure 4.4 presents the processing times of the programs. Error bars are for confidence intervals of 95%.

Note that only the "extreme" examples incurred a large processing overhead. In *FileCopy*, there is almost no processing between I/O accesses. The enforcer gets the data label from each input read, and applies it to each output write. Thus, the enforcer nearly does the same amount of computing as the original program itself. Notice how the enforcer overhead becomes minimal when processing is added between the I/O accesses, in *FileEncrypt*. A similar thing happens in *Loops*, where the program is made entirely by loops, and the enforcer counts number of iterations on all of them. This way, the amount of injected code is large. In all other cases, overhead was almost imperceptible.

Table 4.8 presents the results for memory usage. *AllocCount* and *AllocSize* represent number of memory allocations and used memory size, respectively. Each cell represents the ratio between the value for running that program with and without the enforcer. As expected, the overhead on used memory is minimal, being at most 1.3%, for the *FileCopy*



(a)



(b)

Figure 4.4: Processing times of experiments

program, in which labels are stored in every I/O access. For programs in which loop counting is done, the number of memory allocations can increase noticeably with the enforcer, as seen in *Statistics* and *Loops*. However, since for each loop only an integer is used to count, the overhead on used memory size is still minimal.

Program	AllocCount	AllocSize
Example1	1.012	1.003
Example2	1.019	1.002
Example3	1.000	1.000
FileCopy	1.022	1.013
FileEncrypt	1.000	1.000
InformationGather	1.084	1.000
Statistics	1.253	1.001
Loops	3.413	1.001

Table 4.8: Memory usage ratio of experiments

CHAPTER 5

Open Problems

In this chapter we discuss some of the open problems left by this thesis. We first discuss how precise the graph-based PCR implements the expression-matching framework, in Section 5.1. Then we discuss two open problems that arise from PCR analysis: the loop counting problem in Section 5.2 and the algebraic equivalence problem in Section 5.3. In Section 5.4 we discuss the implementation of graph-based PCR on a real programming language, such as Java or C++. Finally, in Section 5.5 we present a future research question, on how to perform PCR analysis on compiled code, such as assembly language.

Along with the open problems, we also present research ideas on how to pursue their solution. These ideas are however insights, which need to be further investigated. Thus, we present them in a discursive manner, without a complete formalization to validate them, laying the foundation for future work in the field.

This thesis focus on the enforcement of declassification policies, rather than their specification. Thus, we do not include here open problems related to enhancing the representation format of the policies, as we consider this a separated domain of problems.

5.1 Information Path Filtering

As stated before, graph-based PCR safely approximates the expression-matching framework. In other words, if the framework rejects a program, then so does the implementation. However, the opposite is not true: a program deemed safe by the framework might be rejected by the implementation. One of the main reasons for this happens due to the way information paths are calculated: some of them are actually impossible to happen during execution, and might represent insecure flows of information. In this section we detail this problem, which we call *information path filtering*.

Consider Example 5.1 below, with its corresponding program graph in Figure 5.1. Here, from the code we can easily conclude that the possible values sent to output γ are $f(\alpha, \beta)$ and $f(\beta, \alpha)$. However, by analyzing the program graph, and recalling information path calculation from Definition 3.8, we can see that the graph-based implementation considers 4 different information paths reaching the node labeled γ: the paths that generate values $f(\alpha, \alpha)$ and $f(\beta, \beta)$ are also considered. This is a clear example of the imprecision of the graph-based implementation. If α and β are both secret input channels, and a declassification policy only allows the release of expressions $f(\alpha, \beta)$ and $f(\beta, \alpha)$, then the implementation rejects this program, as it assumes that the unsafe expressions $f(\alpha, \alpha)$ and $f(\beta, \beta)$ might eventually be output, which is not true.

Example 5.1. *Imprecision of the implementation:*

```
bool c1 := someCondition();
int x0, y0, r0;

if c1 then
    x1 := α;
    y1 := β;
else
    x2 := β;
    y2 := α;
x3 := φc1(x1, x2);
y3 := φc1(y1, y2);
r1 := f(x3, y3);
γ := r1;
```

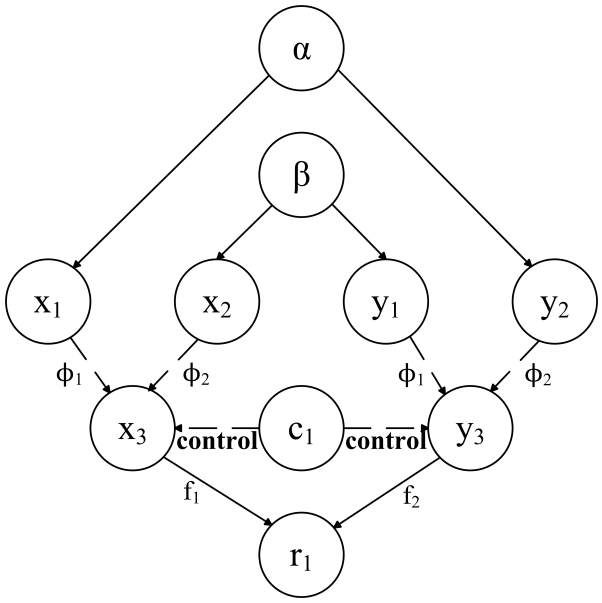


Figure 5.1: Program expression graph for Example 5.1

A straightforward approach for this problem would be, as the name implies, filtering the set of information paths prior to analysis, excluding those which are unreachable. The problem then lies in determining whether or not an information path is reachable. For the example above, this might look simple, as both nodes labeled x_3 and y_3 have an incoming `control` edge from node c_1. The unreachable information paths are those in which two nodes receiving `control` edges from a same third node receive ϕ edges with different indexes, i.e. one ϕ_1 and the other ϕ_2. However, some cases might not be so easily detectable, as we can see in Example 5.2. Here, the program graph is mostly the same as the previous example, with the exception that there are now two conditional variables: c_1 and c_2. Figure 5.2 shows the partial program graph relative to these two variables.

Example 5.2. *Non-trivial example of imprecision:*

`int` x_0, y_0, r_0;
`double` $a_1 := \theta$;
$a_2 := add(a_1, 5)$;
$a_3 := div(a_2, 10)$;
`bool` $c_1 := geq(a_3, 15)$;

`if` c_1 `then`
    $x_1 := \alpha$;
`else`
    $x_2 := \beta$;
$x_3 := \phi_{c_1}(x_1, x_2)$;
$a_4 := div(a_2, 10)$;
$c_2 := geq(a_4, 15)$;

`if` c_2 `then`
    $y_1 := \beta$;
`else`
    $y_2 := \alpha$;
$y_3 := \phi_{c_2}(y_1, y_2)$;
$r_1 := f(x_3, y_3)$;
$\gamma := r_1$;

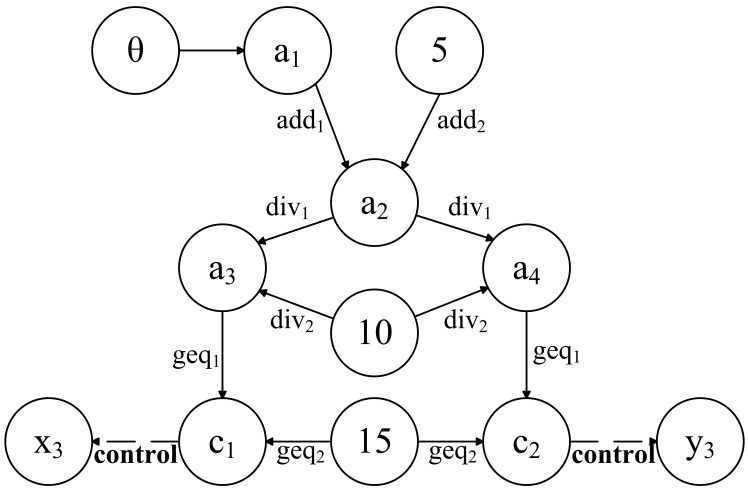


Figure 5.2: Program expression graph for variables c_1 and c_2, in Example 5.2

Note that the problem is essentially the same. However, some redundant programming makes it harder to detect the unreachable information paths just by inspecting the graph structure. The key to the problem lies in determining that nodes labeled c_1 and c_2 hold the same values, for each information path they are both present in. Fortunately, our graph matching mechanism is centered exactly around the notion of determining if two nodes

hold the same expressions: this is our policy simulation, of Definition 3.9. Thus, we can use policy simulation to determine whether two boolean nodes hold the same expression in a given information path, and from there determine if the information path takes two different ϕ-edges that consider that same boolean expression to be both true and false, respectively. These information paths would then be tagged as unreachable. A tentative definition of an unreachable information path is:

$$\begin{aligned} unreachable(p) \quad \equiv \quad & \exists n_a, n_b, n_{c1}, n_{c2}, n'_a, n'_b \in nodes(p) : n_{c1} \sim_{p,p} n_{c2}, n_{c2} \sim_{p,p} n_{c1}, \\ & n_{c1} \xrightarrow{\texttt{control}} n_a, n_{c2} \xrightarrow{\texttt{control}} n_b, n'_a \xrightarrow{\phi_i} n_a, n'_b \xrightarrow{\phi_j} n_b, i \neq j \end{aligned}$$

With $nodes(p)$ returning the set of nodes connected to the information path p [1]. Note that since our policy simulation is unidirectional, we need to check for both $n_{c1} \sim_{p,p} n_{c2}$ and $n_{c2} \sim_{p,p} n_{c1}$. The formal validation of such filter is left here as an open problem.

Here we have dealt with `if` statements, which are one of two statement types in our language that generate ϕ-edges in the graph. The other is the `while` statement. This kind of statement can also cause imprecision in the implementation, but this kind of imprecision is related to the policy representation format. This is due to the fact that the graph generates only a single maximal information path for a `while` command, and this information path captures all possible outcomes of the loop (i.e. not being run, and being run an arbitrary number of times). Thus, the problem we just discussed for the `if` statement do not happen here. What can happen however, is that the framework might specify a set of declassifiable expressions which only occur when a given loop runs for specific number of iterations. In the graph-based implementation, both the policy expression graph and the matching mechanism are not expressive enough to handle the counting of loop iterations. This problem, which affects the precision of the implementation, is treated here as a separated problem, which we name the *loop counting* problem, and discuss in Section 5.2.

Example 5.2 can be made more difficult to analyse if we change how the values held by variables c_1 and c_2 are constructed. For instance, in the example both variables hold the value $\frac{\alpha+5}{10} \geq 15$. We could, however, make variable c_2 hold the value $\neg(\frac{\alpha+5}{2\times5} < 15)$. Note that the values are still equivalent, but just syntactically different. With this, the nodes labeled c_1 and c_2 would no longer simulate each other, and the solution we proposed above would not work. This, however, is an instance of a larger problem, named the *algebraic equivalence* problem, which also affects the whole graph matching process, and is discussed in Section 5.3. We consider that a solution for this problem would be included in our definition of policy simulation, thus also treating cases that affect information path filtering, such as the aforementioned.

Using compiler optimization. Another field of research that can be of great help to further enhance PCR analysis as a whole is compiler optimization. In fact, the SSA form that we use to pre-process programs is used as an intermediate representation stage by many modern compilers. Compiler optimization is able to identify redundancies and unreachable parts of a code, and this information can be used during graph construction in

[1] Recall here that an information path is a set of edges

order to avoid representing expressions which are never calculated, thus making it more precise with respect to the framework. In particular, it can help in the following:

- Detecting conditionals which are always true or false. With this, graph construction can skip entire unreachable sub-graphs (when conditionals are always false), and turn ϕ-edges into `plain` edges (when conditionals are always true).
- Detecting conditionals which are bound to each other. A boolean variable's value might be bound to another, e.g. $c_2 = \neg c_1$. Modern compilers often detect and optimize such situations. In the program graph context, this would simplify e.g., `if` commands nested inside `while` loops, in which both conditionals are bound to each other.
- Detecting identical assignments done in both branches of a same conditional block. In our approach, a situation like this would create a control dependency between the condition and the assigned variable. However, as the assignment is the same in both branches, no information from the condition can be inferred by observing the value of the assigned variable, thus making the control dependency imprecise. Compiler optimization usually treats this scenario by moving the assignment outside of (before) the conditional block, solving the imprecision.

Since this thesis is situated within the field of language-based security, we leave integration with compiler optimization as future work. However, this integration might be somewhat simple, as compiler optimization can be performed before graph construction. With this, the graph construction rules might remain unchanged, and the pre-optimization would serve to ensure that the program does not contain certain structures that cause imprecision, such as the ones cited above.

Using the hybrid enforcer. The information path filtering problem can also be handled by a runtime enforcer, such as the one we propose in our hybrid enforcer. For this, we could add some extra information in the flow report, keeping track of which values conditionals must satisfy, for each flow to actually happen, and then have the enforcer keep track of these conditionals. For example, consider the flow:

$$\{\alpha^i, \beta^j\} \rightsquigarrow \gamma^k$$

We can then have this flow be extended with the following information:

Flow	Conditionals
$\alpha^i \rightsquigarrow \gamma^k$	$c_1 = \texttt{true}, c_2 = \texttt{false}$
$\beta^j \rightsquigarrow \gamma^k$	$c_1 = \texttt{true}, c_3 = \texttt{true}$

The runtime enforcer would then have to keep track of conditionals c_1, c_2 and c_3. Note that this extension would come at the cost of an increase on the overhead caused by the enforcer. Also note that the extension goes in the same direction of making the enforcer permissive, in the sense of being able to safely execute potentially unsafe programs. We do not define our enforcer to be permissive for the sake of simplicity, as we discuss in Section 4.4.

5.2 Loop Iteration Counting

The *loop counting* problem is a consequence of the inability of both the policy representation format and the matching mechanism to keep track of how many times a loop iterates. The problem occurs when it is desired to have a declassification policy that describes an iterative expression, but somehow limits the number of iterations for the expression to be declassifiable. Consider again Example 3.2, on page 38. Here, if the program calculates the average of a single input value, the mechanism will still deem it safe, as it complies with the declassification policy. Thus, it is desirable to have an additional constraint that, e.g. states that the policy is only applicable if the matching loop iterates a minimum number of times. This, along with the input-uniqueness restriction, would ensure a proper disclosure of the average.

This problem only occurs in the implementation, as in the framework the set of declassifiable expressions can contain only expressions with the accepted number of iterated elements. Thus, a different approach for implementing our expression-matching framework might altogether avoid it. As stated in the previous section, this problem adds to the imprecision of the implementation.

Graph analysis. One approach is to analyze the node that holds the loop conditional and try to determine all of its possible values. Consider again Example 3.2: here, one can easily see that the loop iterates $length(\alpha)$ times. An automated mechanism should inspect possible values of variable c_3. One of these can be $c_1 = leq(i_1, l_1) = leq(0, length(\alpha))$. The other is the value of iterative variable $c_2 = leq(i_2, length(\alpha))$, with i_2 also being iterative. The key to the solution is automatically determining that i_2 begins as 1 and is added 1 at each iteration. Thus, the mechanism could conclude that the loop runs $length(\alpha)$ times, with a similar constraint being applied to the policy. One possible research path to achieve this is by designing a walk on the graph that builds, for each node, a *regular tree expression* [CDG+07] that describes the iterative expressions held by that node. Again, we leave further investigation of this issue as an open problem.

Using compiler optimization. Again, a possible path for this problem lies in compiler optimization. Here, the solution would lie in using optimization techniques to find, for each looping construct, a "loop invariant", i.e. a set of bounds for the number of times the loop runs. These bounds would then be included in the program graph, and used during the matching process. Finding loop invariants is, however, a problem known to be difficult [SSM04].

Using the hybrid enforcer. The use of a runtime enforcer, such as in our hybrid mechanism, renders the problem trivial. As demonstrated in Chapter 4, our runtime enforcer uses runtime constraints in order to enforce loop iteration counting. These are specified along with the declassification policy, and the runtime enforcer counts the number of iterations of the specific loops which are relevant for the matching. The incurred overhead, in most practical cases, is very small, as demonstrated in Section 4.5.1. This shows that, for an environment in which using a hybrid system is possible, loop counting is not a difficult problem.

5.3 Algebraic Equivalence of Expressions

The *algebraic equivalence* problem is related to the fact that expressions might be syntactically different, but semantically equal [HO08]. For instance, the expression $e_1 = a(b+c)$ is the same as $e_2 = ab + ac$. Our current matching mechanism does not account for this problem. In other words, if a policy allows for the declassification of expression e_1 and a program calculates e_2, the program will not match the policy.

Since the framework is theoretical, one can state that the problem does not happen on it, but rather only in the implementation. However, since the framework is based on expression-matching, we believe that the problem will arise in most (if not any) implementation of it. Different approaches for the implementation might impact on the complexity of its solution, however.

Term Rewriting Systems. One possible research path for this problem is to design a term rewriting system (TRS) [KBV01] in order to convert both program and policy graphs to a *canonical form*. We believe that such a solution would need the following steps to be achieved:

1. Design a TRS for the standard logic-arithmetic operations of a programming language. This TRS should be a *equational based* TRS in which normal forms are sums of products of powers, e.g. $\alpha\beta^2 + \alpha^3\gamma^\pi + \delta^{(2\alpha+\beta-5)}$. As the TRS is built upon an equational specification (ES), then, for every rule in the TRS, there is an expression in the form $e_1 = e_2$. We know that using an algorithm such as Knuth-Bendix we can transform an equational specification into a TRS. For this, the equational specification must have an ordering over terms defined, which can be the lexicographic ordering over variable names and function symbols. It is known that such algorithms for building a TRS over an ES build a *complete* TRS, that is, a TRS with both termination and confluence properties. The confluence property also implies the unique normal form property. These properties are fundamental for the system to be tractable and applicable.

2. Convert the TRS to an *information path rewriting system* (IPRS). Thus, the system would make the same modifications of the TRS directly on the information paths [2]. Note that this step is not trivial, and that some work is necessary to keep the TRS's termination and non-ambiguity (confluence) properties. Notably:

    - Since nodes can have an outdegree larger than one, they can match more than one rule, creating ambiguity. A pre-processing on the information path is necessary, in which each node is "split" into a number of nodes equal to its outdegree, each generated node with one outgoing edge, and the same incoming edges of its progenitor. The process should be done in a "bottom-up" fashion, from outputs to inputs, until it reaches the direct children of input nodes. These, which represent distinct input accesses, will not match the rule

[2] If a policy has ϕ-edges outside of cycles, then consider that the policy is also broken into information paths.

(as their parents are not `var` nodes), and will be the only ones having multiple outgoing edges.

- Rewriting rules that identify multiple occurrences of a same input can only be applied if the structures on the graph represent the very same input operation, but not when they represent different accesses on a same channel. That is, two occurrences of α_1 represent a same ground term (i.e. $\alpha_1 + \alpha_1 = 2\alpha_1$), while α_2 has to be treated as a different ground term. Thus, since each immediate child of an input node represent a distinct input operation on the graph, the ground terms of the IPRS are the sets of two nodes and an edge in the form $n_\alpha \rightarrow n$.
- Cycles need a special treatment. The straightforward solution should be treating the *cycle roots* (i.e. the nodes that receive the two ϕ-edges) as "unmatchable". The practical consequence of this is that rules would be able to match patterns both outside and inside a cycle, but not a pattern that spans both areas.

3. Finally, define how to add new operations to the IPRS, such that it retains its fundamental properties (termination and confluence). This also requires some work, as standard TRS research does not tackle the problem of extending a TRS, whilst retaining its properties. However, it is important to note that more complex programming operations (such as enc, or the authentication ones from Example 3.1) tend to be simpler to be added to the IPRS, as they tend to have very few (or none at all) equivalence rules.

With this, the IPRS would be used to convert both program and policy graphs to canonical forms, prior to the matching. Since the solution would work at the matching process, it would also be applicable for the algebraic equivalence aspect of the information path filtering. Again, we leave a formalization of this approach as an open problem.

We believe that the solutions for the three aforementioned open problems (information path filtering, loop counting and algebraic equivalence) would make the implementation highly precise, with the only "rejected safe programs" being very specific examples of unreachable code. And even those might be treated by a pre-optimization of the code.

5.4 Graph-Based PCR on a Real Programming Language

A crucial step for the adoption of PCR analysis in industry is applying it to a real programming language, such as Java and C++. In this thesis we define the mechanism over a simple toy language for the sake of simplicity of the definitions. Our toy language is trivially shown to be Turing complete, and so programs implemented in other languages can be converted to it. In this section we discuss how can this toy language be extended to include constructs of modern programming languages. It is noteworthy that most of these extensions present *engineering*, rather than *research* problems, another reason for being left out of this thesis.

Control-flow constructs. These include statements such as `switch-case`, `repeat`, `do-while`, `break`, `continue`, `goto` (to static program labels) and others. Most of

these statements can be easily supported by our mechanism, as there are known SSA translation rules that handle these statements, using ϕ-functions on control-flow branches, in similar ways as to how `if` and `while` statements are treated.

Modularization. These include structures such as procedures, functions and object-orientation. We discuss in Section 3.6 how to extend our mechanism to support user-defined functions. That same approach, of generating a sub-graph of the function, and then using edges for call and return can be applied for most of such cases [RHS95]. In object-oriented programming, the update of instance variables will result in ϕ-functions being used every time such a variable is returned by a method. The same applies for static and global variables. Again, interprocedural SSA translation rules already exist [LDB+99, SVKW07] and might be adapted to our framework.

Pointers and arrays. Extensions of SSA to handle arrays and pointers are well-known [CCL+96, KS98, FKS00], and could be used to support these structures. The main goal lies in determining how these structures affect the precision of the graph-based implementation, and how to adjust it in order to to keep it precise. The main concern lies in arrays with different positions holding values of different security levels. The inclusion of such structures might incur in the need to extend the program expression graphs.

Unpredictable jumps. Exception control and computed `goto` statements (i.e. jump to arbitrary points of the code) pose bigger challenges. These structures potentially create implicit flows with larger portions of the code. For instance, using Java and C++ syntax for exception handling, the code inside a `catch` block receives an implicit flow of information from every variable that is read/written within the correspondent `try` block. Extensions of SSA to handle exceptions do exist though [GPF05], but the effect of such extensions on the precision of our implementation has yet to be investigated.

Concurrency. Most static analysis techniques have difficulties in supporting concurrency. Even though deep characterization and formalization of this aspect is out of the scope of this thesis, we do know that ϕ-functions can be used to join versions of a variable that can be assigned by different threads. With that, we can keep the soundness of our graph representation (i.e. each node representing the corresponding variable's possible expressions), and analysis can still be adapted to work properly. As with the other constructs, there are extensions on SSA meant to deal with concurrent programs in more precise ways [SGW94, LPM99]. Also noteworthy is that concurrency can be treated by the hybrid enforcer, by having variables which are shared between threads be modeled as I/O channels of the `data` type.

Extensions to standard SSA are known to be more expressive and allow to track more information about a program. For instance, weak dynamic single assignment form [OK03] aims to further help the analysis of loop and array based codes for parallel targets. Also, static single information form (SSI) [Ana99] extends SSA to achieve symmetry for both forward and reverse dataflow. Since SSA and its extensions are aimed mostly for compiler construction and code optimization, the applicability of such extensions on information flow analysis must be further investigated, for each case separately.

5.5 Policy Controlled Release on Assembly Code

We conclude this chapter with the discussion of a long-term open problem not only for PCR analysis, but for information flow and declassification analysis as a whole. This problem, as the section name implies, is the one of performing PCR analysis on low-level code, such as assembly code. The greatest benefit of such a solution would be a practical merge of the access control and malware detection fields. A mechanism that is able to detect flows of information performed by compiled untrusted programs, supporting declassification policies, will essentially render obsolete the two aforementioned fields used by industry today.

Here, a program graph of the assembly code can be built, with memory addresses being treated like program variables. In fact, there is also research for making SSA form on assembly code [LG99]. However, the use of assembly code raises some new challenges.

The first problem of such analysis is identifying implicit flows. Since assembly code lacks the "block structure" of high level programming languages, regions of the code that depend with a conditional must be found and identified, prior to analysis. As a simple example, consider both pieces of code in Figure 5.3. In this simple example it is clear to see how compilation to assembly code implies in the loss of the original structure that was present in the high-level code.

```
if x then
    y := 1;
else
    y := 2;
z := 3;
```

(a) High-level program

```
L1 :  bnz r1, L2      % if x ≠ 0 goto L2
      move r2 <- 1    % y := 1
      jmp L3
L2 :  move r2 <- 2    % y := 2
L3 :  move r3 <- 3    % z := 3
```

(b) Assembly program

Figure 5.3: Example of source to assembly compilation

A great challenge for such kind of analysis is handling control jumps to statically unpredictable points of the code. For instance, consider a `goto` (or `jump`) command that takes a memory reference as an argument. If the destination of the jump cannot be calculated at static time (i.e. it depends on an input command), then the whole code is a possible target for the jump, thus whichever control dependencies exist with the jumping instruction will be extended to the *whole* code. This is clearly too strict and some form of new analysis will be needed to tackle this problem.

Some research in bringing information flow control to low-level code already exists, although still far from reaching the expressive level that PCR achieves with source code. Some approaches rely on using specially typed assembly languages [BR05, YI06] to guarantee non-interference, but this violates our principle of separating the analysis from the code. Also with the aim of simplifying this kind of analysis, Java bytecode has become a target for information flow analysis [GS05], as it is slightly simpler to analyze than native machine code. Declassification policies are however not yet supported, and consequently neither is the full separation from program and policy that we aim.

Thus, we believe that the question to be answered with regards to PCR analysis in assembly code is not (yet) *how* to do it, but rather *if* it can be done. We conclude this chapter with a new research question, to be answered by future research:

Can the PCR framework be implemented for analysis of assembly code?

CHAPTER 6

Concluding Remarks

In this thesis we have addressed the problem of information flow and declassification analysis on legacy and untrusted programs. We recall the research question proposed in Section 1.3:

> How to check information flow on legacy, untrusted and mobile code?

We have answered this question by (i) decoupling declassification analysis from programs, (ii) designing decidable algorithms for checking declassification and (iii) adapting these mechanisms to work in multiple systems and with little to no runtime overhead.

6.1 Contributions

We have proposed 3 related mechanisms for information flow and declassification analysis, starting with a more theoretical one and ending with a more practical system, each building upon its predecessor. We start with a purely theoretical framework, then move to a computable implementation of the framework, and end with an extension of the implementation which works with existing technologies. All proposed mechanisms work on unannotated code, with declassification policies independent from it, thus satisfying the necessary conditions to answer the research question.

We have introduced an expression-matching *framework* that defines validity of a program with respect to the expressions calculated by it. The framework checks all the expressions a given program can possibly output and checks them against a set of expressions which are allowed to be declassified (the set of *declassifiable expressions*). We formalized a property that states that the program does not reveal any more information than that specified by such declassifiable expressions. We named this property Policy Controlled Release (PCR).

We have also developed a high level *implementation* of the framework, which uses a form of graphs to calculate whether a program satisfies PCR or not. While the PCR property that the framework refers to is undecidable, the algorithms we introduce for implementing it are recursive, with polynomial complexity. Indeed, we show that our implementation is a *safe approximation* of the framework, in the sense that some valid programs can be rejected, but not the opposite. We present a form of graphs to represent expressions calculated by a program, along with a matching mechanism based on automata simulation, as well as polynomial algorithms for the whole mechanism.

Finally, we present an *extension* of the implementation, in which we combine it with runtime components to achieve an enforcement mechanism that can be applied to current technologies and application examples. In order to tackle some aspects of information flow enforcement that static analysis does not cover, we combine our implementation with runtime enforcement techniques. We then show that this hybrid solution is not only able to handle problems that current solutions are incapable of, but also with a negligible runtime overhead for most cases. We use a mobile platform (Android) as the setting for our examples. In particular, the hybrid approach achieves a few accomplishments that current techniques do not: (1) it performs a system independent static analysis, due to the presence of system-specific labeling system that is decoupled from program and policy, (2) supports dynamic (runtime) security labels, (3) handles runtime declassification constraints, and (4) its runtime component is lightweight enough to be implemented on mobile devices.

Our work takes a first step in a new direction in the information-flow field. We believe the analysis of legacy and untrusted programs, along with a program-independent, policy-based declassification mechanism will represent an important step towards bridging the gap between academic research in the field and its widespread adoption in industry. In the rest of this chapter we present limitations of our work, in-depth discussion of some of the concepts presented in this thesis, and then future work.

6.2 Limitations

We built our mechanism over some simple assumptions: we use a simple imperative toy language; we define simple algorithms, with the purpose of demonstrating the tractability of the implementation, but leaving more optimized algorithms out of the scope; and we leave some operational issues untreated (but discussed). However, we pave the way for these assumptions to be relaxed, towards a mechanism that will be able to analyze legacy systems using newly created declassification policies.

Also, as previously mentioned, our graph-based implementation is a safe approximation of the framework. We discuss in Chapter 5 a few cases in which our analysis can deem a safe program unsafe. However, we do not know how many real programs would be affected by such imprecision. Also, we do not know if there is a limit (and what would that limit be) to the precision that any tractable implementation of the PCR framework can achieve. In other words: whether or not is it possible to have an implementation of PCR which is 100% precise.

The concept of declassification requires policies not to be invertible. That is, if a declassification policy allows $f(\alpha)$ to be disclosed, in a scenario where an f^{-1} function exists, then the policy is actually allowing α to be disclosed. Even if there is some mechanism to check if a given function has not been inverted throughout the code, nothing would prevent the inverse function from being applied outside of the program. This is a natural requirement for declassification, and we therefore we assume that well-formed policies do not allow invertible expressions to be declassified.

6.3 Discussion

Our flow-analysis is termination-insensitive [AHSS08], in the sense that it does not tackle information flows caused by observing the termination behaviour of the program. An example is a program which either executes a time consuming operation or quickly terminates, depending on the value of a secret input. By observing when the program terminates one can infer some information about the secret input. We can make the analysis termination-sensitive either by disallowing while loops under high conditionals or introducing a flow between the conditional in which the loop is declared to all the output channels in the program. However, we believe that both these approaches are too restrictive, resulting in the rejection of many safe programs. Disallowing the while loops under high conditionals would also make the program dependent on the policy, which is exactly what we want to avoid. We consider termination and timing channels to be easier to treat outside of the scope of program analysis, i.e. in the level of the operating system, which can impose limitations on the visibility of execution time and termination of processes. Thus, we do not deal with termination and timing channels.

Sabelfeld and Sands have defined four *dimensions of declassification* [SS05]: *what* can be declassified, i.e. which function f allows $\alpha' = f(\alpha)$ to be released; *where* in the program can declassification happen; *who* can perform it; and *when*, in terms of preconditions, can it happen (e.g. data can only be declassified if a certain database has at least n entries). The policies used by our framework specify the expressions over inputs that can be declassified, so they address the *what* dimension of declassification. This is done by automatically detecting which operations are applied over secret data, and determining whether or not the derived data is allowed to be disclosed. Also, our analysis precludes the need for the *where* dimension, since our enforcement matches a policy with any part of the program that satisfies it. In the case of legacy code, the programs are typically written without information-flow policies explicitly defined. For untrusted code, we have sought an approach that provides assurance without requiring to trust the programmer. Nevertheless, for cases in which the *where* dimension is required, it is straightforward to specify program points at which a particular policy may be applied by associating this condition with the policy itself; no code-annotations are required. On the other hand, utilizing the *where* dimension extensively (i.e. determining which parts of the code are allowed to declassify) would be contrary to our goal of making the policy program-independent. It is straightforward to extend our analysis to address the *who* dimension as well, as the system operator can control which policy graphs are used in analyzing the program based on *who* wrote the program and the policies, and *who* is going to observe the outputs from the output channels. The *when* dimension can be easily specified by a series of pre-conditions to be checked. In our hybrid enforcer, these conditions can also be checked at runtime, along with the pre-existing runtime checks. It is of note, however, that this dimension is usually program and/or system dependent, as pre-conditions are associated with specific states of the program and/or system.

6.4 Future Work

Our work raises a number of new problems and future work. In Chapter 5 we present and discuss a number of open problems. Graph-based PCR can be made applicable to most real scenarios by becoming more precise through a refinement of its information path calculation mechanism (Section 5.1) and being able to match policies and program points that calculate expressions which are syntactically different but semantically equivalent, such as $\alpha + \alpha$ and 2α (Section 5.3). Also, it could greatly benefit from a solution to the loop counting problem (Section 5.2) that works entirely on the static analyzer level, which would also further reduce runtime overhead in the hybrid enforcer. Extending our mechanism to work with a real programming language such as Java or C++ would allow this kind of analysis to be used in industry (Section 5.4).

We believe that open problems are divided into the following 3 main paths for future work in the field:

- Quantify how precise graph-based PCR implements the expression-matching framework, in order to determine how many safe programs could be deemed unsafe. If this number is not negligible, then either improve the precision of our implementation or present another, more precise one. In other words, answer the question: *How precisely can the PCR expression-matching framework be implemented, in a computable fashion?*

- Extend both framework and implementation to work on a complete existing programming language, such as Java or C++. From that, study the language-dependency of the approach and answer the question: *Can PCR analysis be defined in a language-independent way?*

- Bring the concepts of PCR analysis to a lower level programming environment, such as bytecode or assembly code. With this, information flow analysis would be possible to be done in compiled programs, without the need for the corresponding source code. This would greatly enhance the applicability of the analysis, since many programs are provided only in their compiled form, especially the ones downloaded from the Internet from untrusted sources. This would allow, for instance, using information flow to detect malware (viruses, worms, etc.), in a much more efficient manner than that of current anti-virus technologies, that rely on libraries of *known* malware. For this, a number of research challenges must be tackled, so that the following question might be answered: *Can the PCR framework be implemented for analysis of assembly code?*

Bibliography

[AB04] Torben Amtoft and Anindya Banerjee. Information Flow Analysis in Logical Form. In *SAS '04: 11th International Static Analysis Symposium*, pages 100–115, 2004. 5, 11

[ABB06] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *POPL '06: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 91–102, 2006. 5, 11

[AF09] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. *SIGPLAN Notices*, 44:20–31, December 2009. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/1667209.1667223`. 15

[AHSS08] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS '08: Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333 – 348. Springer, 2008. 105

[AI08] Luca Aceto and Anna Ingolfsdottir. Deciding Bisimilarity over Finite Labelled Transition Systems is P-complete. *Information and Computation*, pages 1–8, 2008. 59

[Ana99] C.S. Ananian. *The Static Single Information Form*. Ph.D. thesis, Massachusetts Institute of Technology, 1999. 99

[AS05] Aslan Askarov and Andrei Sabelfeld. Security-Typed Languages for Implementation of Cryptographic Protocols: A Case Study. In *ESORICS '05: Proceedings of the 10th European Symposium on Research in Computer Security*, pages 197–221, 2005. 13

[AS07a] Aslan Askarov and Andrei Sabelfeld. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *SP '07: Proceedings of the 28th IEEE Symposium on Security and Privacy*, pages 207–221. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2848-1. 13, 30

[AS07b] Aslan Askarov and Andrei Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *PLAS '07: Proceedings of the 2nd Workshop on Programming Languages and Analysis for Security*, pages 53–60. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-711-7. 12

[AS09] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF '09: Proceedings of the 22nd IEEE Computer Security Foundations Symposium*, pages 43–59. IEEE Computer Society, Washington, DC, USA, 2009. 13, 16

[BBM94] Jean-Pierre Banâtre, Ciarán Bryce, and Daniel Le Métayer. Compile-Time Detection of Information Flow in Sequential Programs. In *ESORICS '94: Proceedings of the 3rd European Symposium on Research in Computer Security*, pages 55–73. Springer-Verlag, London, UK, 1994. ISBN 3-540-58618-0. 5, 11

[BC85] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, 1985. ISSN 0164-0925. 11

[BLW09] Lujo Bauer, Jay Ligatti, and David Walker. Composing expressive runtime security policies. *ACM Transactions on Software Engineering and Methodology*, 18:9:1–9:43, June 2009. ISSN 1049-331X. URL `http://doi.acm.org/10.1145/1525880.1525882`. 15

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, 1994. ISSN 0164-0925. 22

[BN02] Anindya Banerjee and David A. Naumann. Secure Information Flow and Pointer Confinement in a Java-like Language. In *CSFW '02: Proceedings of the 15th IEEE Workshop on Computer Security Foundations*, page 253. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-7695-1689-0. 11

[BNR08] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive Declassification Policies and Modular Static Enforcement. In *SP '08: Proceedings of the 29th IEEE Symposium on Security and Privacy*, pages 339–353, 2008. 13, 30, 31

[BP03] G. Bilardi and K. Pingali. Algorithms for computing the static single assignment form. *Journal of the ACM*, 50(3):375–425, 2003. ISSN 0004-5411. 21

[BR05] Gilles Barthe and Tamara Rezk. Non-interference for a jvm-like language. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '05, pages 103–112. ACM, New York, NY, USA, 2005. ISBN 1-58113-999-3. URL `http://doi.acm.org/10.1145/1040294.1040304`. 100

[BWW08] Sruthi Bandhakavi, William H. Winsborough, and Marianne Winslett. A Trust Management Approach for Flexible Policy Management in Security-Typed Languages. In *CSF '08: Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 33–47, 2008. 12, 73

[CCL+96] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 253–267. Springer-Verlag, London, UK, 1996. ISBN 3-540-61053-7. URL `http://portal.acm.org/citation.cfm?id=647473.760381`. 99

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. 2007. release October, 12th 2007. 46, 96

[CFR+91] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4): 451–490, 1991. ISSN 0164-0925. 20, 21

[CHH02] David Clark, Chris Hankin, and Sebastian Hunt. Information flow for algol-like languages. *Computer Languages*, 28(1):3–28, 2002. 11

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 198–209. New York, NY, USA, 2004. ISBN 1-58113-961-6. 12

[CM08] Stephen Chong and Andrew C. Myers. End-to-End Enforcement of Erasure and Declassification. In *CSF '08: Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 98–111, 2008. 12, 16

[CMVZ06] S. Chong, A. C. Myers, K. Vikram, and L. Zheng. *Jif Reference Manual*, June 2006. URL `http://www.truststc.org/pubs/548.html`. 6, 12

[CNC10] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *Proceedings of the Thirteen Information Security Conference (ISC 2010)*, pages 331–345, 2010. 15

[Den76] Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976. 5, 11

[FKS00] Stephen Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *Proceedings of the 7th International Static Analysis Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 155–174. Springer, 2000. URL `http://citeseer.ist.psu.edu/fink00unified.html`. 99

[GD04] Pablo Giambiagi and Mads Dam. On the secure implementation of security protocols. *Science of Computer Programming*, 50(1-3):73–99, 2004. ISSN 0167-6423. 14

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *SP '82: Proceedings of the 3rd IEEE Symposium on Security and Privacy*, pages 11–20, 1982. 3, 11

[GM04] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 186–197. ACM, New York, NY, USA, 2004. ISBN 1-58113-729-X. 14

[GPF05] Andreas Gal, Christian W. Probst, and Michael Franz. Structural encoding of static single assignment form. *Electronic Notes in Theoretical Computer Science*, 141(2):85–102, 2005. URL `http://dx.doi.org/10.1016/j.entcs.2005.02.045`. 99

[GS05] Samir Genaim and Fausto Spoto. Information flow analysis for java bytecode. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer Berlin / Heidelberg, 2005. URL `http://dx.doi.org/10.1007/978-3-540-30579-8_23`. 100

[HKM05] Boniface Hicks, Dave King, and Patrick McDaniel. Declassification with Cryptographic Functions in a Security-Typed Language. Technical Report NAS-TR-0004-2005, Network and Security Center, Department of Computer Science, Pennsylvania State University, January 2005. (*updated May 2005*). 12

[HKMH06] Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: high-level policy for a security-typed language. In *PLAS '06: Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, pages 65–74. ACM, New York, NY, USA, 2006. ISBN 1-59593-374-3. 6, 12, 13

[HKS06] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *ISSSE '06: Proceedings of the IEEE International Symposium on Secure Software Engineering*. IEEE, 2006. 11

[HO08] Joe Hendrix and Hitoshi Ohsaki. Combining equational tree automata over ac and aci theories. In *RTA '08: Proceedings of the 19th International Conference on Rewriting Techniques and Applications*, pages 142–156. Springer-Verlag, 2008. 97

[HS09] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009. Supersedes ISSSE and ISoLA 2006. 11

[KBV01] J. W. Klop, Marc Bezem, and R. C. De Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, New York, NY, USA, 2001. ISBN 0521391156. 97

[KKL+01] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-mac: A run-time assurance tool for java programs. *Electronic Notes in Theoretical Computer Science*, 55(2):218 – 235, 2001. ISSN 1571-0661. URL `http://www.sciencedirect.com/science/article/B75H1-4DD87N7-F1/2/70fb100cffc254de625bcd7bddc4f3da`. RV'2001, Runtime Verification (in connection with CAV '01). 15

[KS98] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 107–120, 1998. URL `http://doi.acm.org/10.1145/268946.268956`. 99

[LBW05] J Ligatti, Lujo Bauer, and D Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, February 2005. 7

[LBW09a] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12: 19:1–19:41, January 2009. ISSN 1094-9224. URL `http://doi.acm.org/10.1145/1455526.1455532`. 15

[LBW09b] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System and Security*, 12:19:1–19:41, January 2009. ISSN 1094-9224. URL `http://doi.acm.org/10.1145/1455526.1455532`. 15

[LDB+99] Shih-Wei Liao, Amer Diwan, Robert P. Bosch, Jr., Anwar Ghuloum, and Monica S. Lam. SUIF Explorer: an interactive and interprocedural parallelizer. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48, 1999. URL `http://citeseer.ist.psu.edu/liao00suif.html`. 99

[LG99] Allen Leung and Lal George. Static single assignment form for machine code. *SIGPLAN Notices*, 34:204–214, May 1999. ISSN 0362-1340. URL `http://doi.acm.org/10.1145/301631.301667`. 100

[LG07] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 218–232. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2819-8. URL `http://portal.acm.org/citation.cfm?id=1270382.1270653`. 15

[LGBJS07] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Asian computing science conference on Advances in computer science: secure software and related issues*, ASIAN'06, pages 75–89. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-77504-8, 978-3-540-77504-1. URL `http://portal.acm.org/citation.cfm?id=1782734.1782741`. 15

[Li09] Weisong Li. Algorithms for Computing Weak Bisimulation Equivalence. In *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 241–248. IEEE, July 2009. ISBN 978-0-7695-3757-3. URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5198508`. 59

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *SSYM '05: Proceedings of the 14th Conference on USENIX Security Symposium*, page 18. USENIX Association, Berkeley, CA, USA, 2005. 7, 14

[LPM99] Jaejin Lee, David A. Padua, and Samuel P. Midkiff. Basic compiler algorithms for parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1999. URL `http://doi.acm.org/10.1145/301104.301105`. 99

[LR10] Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 87–100. Springer-Verlag, Berlin, Heidelberg, 2010. ISBN 3-642-15496-4, 978-3-642-15496-6. URL `http://portal.acm.org/citation.cfm?id=1888881.1888889`. 15

[LZ05] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. *SIGPLAN Notices*, 40(1):158–170, 2005. ISSN 0362-1340. 12

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-115007-3. 10, 43, 47

[Mye99] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241. ACM, New York, NY, USA, 1999. ISBN 1-58113-095-3. 5, 11, 73

[NJK+07] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *In Proceeding of the Network and Distributed System Security Symposium (NDSS'07)*, 2007. 15

[OK03] Carl Offner and Kathleen Knobe. Weak dynamic single assignment form. Technical Report TR-HPL-2003-169, HP Labs, Nov 2003. URL `http://www.hpl.hp.com/techreports/2003/HPL-2003-169R1.html`. 99

[PS03] Francois Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003. ISBN 0164-0925. 5, 11

[QDXW04] Long Qin, Si Duanfeng, Han Xinhui, and Zou Wei. A hybrid security framework of mobile code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '04, pages 390–395. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2209-2-1. URL `http://portal.acm.org/citation.cfm?id=1025117.1025528`. 15

[RBdH+10] Bruno P. S. Rocha, Sruthi Bandhakavi, Jerry den Hartog, William H. Winsborough, and Sandro Etalle. Towards Static Flow-based Declassification for Legacy and Untrusted Programs. In *SP '10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 93–108. IEEE Computer Society, 2010. 17, 19, 35

[RBdH+11] Bruno P. S. Rocha, Sruthi Bandhakavi, Jerry den Hartog, William H. Winsborough, and Sandro Etalle. Information Flow and Declassification Analysis for Legacy and Untrusted Programs. *Submitted*, 2011. 17, 19, 35

[RCEC11] Bruno P. S. Rocha, Mauro Conti, Sandro Etalle, and Bruno Crispo. Hybrid Static-Runtime Information Flow and Declassification Enforcement. In *Submitted*, 2011. 17, 69

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, New York, NY, USA, 1995. 99

[RS10] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, CSF '10, pages 186–199. IEEE Computer Society, Washington, DC, USA, 2010. ISBN 978-0-7695-4082-5. URL `http://dx.doi.org/10.1109/CSF.2010.20`. 15

[Sch00] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3:30–50, February 2000. ISSN 1094-9224. URL `http://doi.acm.org/10.1145/353323.353382`. 7, 86

[SCH08] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A Language for Enforcing User-defined Security Policies. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 369–383. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3168-7. URL `http://portal.acm.org/citation.cfm?id=1397759.1398040`. 73

[SGW94] Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *Proceedings of the Hawaii International Conference on Systems Sciences*, pages 43–52, 1994. URL `http://citeseer.ist.psu.edu/stoltz94extended.html`. 99

[SH08] Nikhil Swamy and Michael Hicks. Verified enforcement of stateful information release policies. In *PLAS '08: Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 21–32. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-936-4. 11, 12

[SHTZ06] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. Managing policy updates in security-typed languages. In *CSFW '06: Proceedings of the 19th IEEE Workshop on Computer Security Foundations*, pages 202–216, 2006. 12

[SM03a] Andrei Sabelfeld and Andrew C. Myers. A Model for Delimited Information Release. In *ISSS '03: International Symposium on Software Security*, pages 174–191, 2003. 12

[SM03b] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, 2003. 3, 11

[SMH01] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101. Springer-Verlag, London, UK, 2001. ISBN 3-540-41635-8. URL http://portal.acm.org/citation.cfm?id=647348.724331. 15

[SPB09] Avraham Shinnar, Marco Pistoia, and Anindya Banerjee. A language for information flow: dynamic tracking in multiple interdependent dimensions. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 125–131. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-645-8. URL http://doi.acm.org/10.1145/1554339.1554354. 12

[SR10] Andrei Sabelfeld and Alejandro Russo. From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research. In Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov, editors, *Perspectives of Systems Informatics*, volume 5947 of *Lecture Notes in Computer Science*, pages 352–365. Springer Berlin / Heidelberg, 2010. 15

[SS05] Andrei Sabelfeld and David Sands. Dimensions and Principles of Declassification. In *CSFW '05: Proceedings of the 18th IEEE Workshop on Computer Security Foundations*, pages 255–269, 2005. 4, 11, 105

[SSM04] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '04, pages 318–329. ACM, New York, NY, USA, 2004. ISBN 1-58113-729-X. URL http://doi.acm.org/10.1145/964001.964028. 96

[SST07] Paritosh Shroff, Scott Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 203–217. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2819-8. URL http://portal.acm.org/citation.cfm?id=1270382.1270652. 15

[SVKW07] Stefan Staiger, Gunther Vogel, Steffen Keul, and Eduard Wiebe. Interprocedural Static Single Assignment Form. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 1–10, 2007. URL http://dx.doi.org/10.1109/WCRE.2007.31. 99

[TPF+09] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 87–97. New York, NY, USA, 2009. ISBN 978-1-60558-392-1. 7, 14

[TZ05] Stephen Tse and Steve Zdancewic. A Design for a Security-Typed Language with Certificate-Based Declassification. In *ESOP '05: 14th European Symposium on Programming*, pages 279–294, 2005. 12

[VIS96] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4:167–188, 1996. 5, 11

[YI06] D. Yu and N. Islam. A typed assembly language for confidentiality. *Lecture notes in computer science*, pages 162–179, 2006. 100

[YWZK09] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 291–304. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-752-3. URL `http://doi.acm.org/10.1145/1629575.1629604`. 16

[YZLL11] Junfeng Yu, Shengzhi Zhang, Peng Liu, and ZhiTang Li. Leakprober: a framework for profiling sensitive data leakage paths. In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 75–84. ACM, 2011. 15

[Zda04] Steve Zdancewic. Challenges in Information-flow Security. In *PLID '04: Proceedings of the First International Workshop on Programming Language Interference and Dependence*. Verona, Italy, August 2004. 5

APPENDIX A

Proofs

A.1 Proof of Lemma 2.19

Proof. The proof is by construction. We inductively construct a relation Q between the two traces such that it is a correspondence. In the base case, if S has zero steps then we are done by relating $\langle \tau, \omega_0 \rangle$ and $\langle \tau, \omega_0' \rangle$. This is true because there are no variables and ouputs defined yet and the safe inputs correspond since $\pi \approx_D \pi'$. In the induction step, a correspondence relation Q between partial run S of length n and partial run S' of length m can be extended to the partial run $S.\langle o_{n+1}, \langle C_{n+1}, \sigma_{n+1}, \pi \rangle\rangle$ of length $n+1$ (if it exists) and an extension of run S' by zero or more steps. When the active commands of both the runs match, i.e. $head(C_n) = head(C'_m)$, the relation Q is extended to include the pair $(n+1, m+1)$. We prove that $Q' = Q \cup \{(n+1, m+1)\}$ is still a correspondence by a straightforward case analysis of the type of the active command. When the active commands in both the runs do not match, then the commands have level H, and the two runs are extended until they both reach commands with level L. Extending the correspondence is straightforward in these high regions because the nondeclassifiable nature of the control context makes output impossible and maintanence of state compatibility hold trivially. The complete proof is as follows:

Case 1: $head(C_n) = head(C'_m)$. Since our program semantics given in Figure 2.1 are deterministic, both runs take a single execution step producing $S_{n+1} = \langle o_{n+1}, \langle C_{n+1}, \sigma_{n+1}, \pi \rangle\rangle$ and $S'_{m+1} = \langle o'_{m+1}, \langle C'_{m+1}, \sigma'_{m+1}, \pi' \rangle\rangle$. Given $(n, m) \in Q$, we extend the relation Q with $(n+1, m+1)$ obtaining Q'. We have to prove that this extension preserves the correspondence property. We prove the following cases, based on the structure of the command $head(C_n)$:

- **Case (Skip):** $C_n = C'_m = \texttt{skip}; C$ and $head(C_n) = \texttt{skip}$. From the semantics we have $S_{n+1} = \langle \tau, \langle C, \sigma_n, \pi \rangle\rangle$ and $S'_{m+1} = \langle \tau, \langle C, \sigma'_m, \pi' \rangle\rangle$. Since there are no output actions and the states do not change, it is easily checked that the induction obligation—$Q' = Q \cup \{(n+1, m+1)\}$ is also a correspondence—follows from the induction hypothesis.

- **Case (Input):** $C_n = C'_m = x := \alpha; C$ and $head(C_n) = x := \alpha$. From the semantics we have $S_{n+1} = \langle \tau, \langle \texttt{skip}; C, \sigma_{n+1}, \pi \rangle\rangle$ and $S'_{m+1} = \langle \tau, \langle \texttt{skip}; C, \sigma'_{m+1}, \pi \rangle\rangle$.

  1. Since $o_{n+1} = o'_{m+1} = \tau$, the outputs generated by the two traces do not change; therefore, $o(S_0) \ldots o(S_{n+1}) = o(S'_0) \ldots o(S'_{m+1})$.

2. If $cds(x, C, D)$, then from Proposition 2.8 we have that $cds(\alpha, C, D)$ holds. Consequently, from the induction hypothesis we know that $I_{\sigma_n}(\alpha) = I_{\sigma'_m}(\alpha)$ and therefore $I_{\sigma_{n+1}}(\alpha) = I_{\sigma_n}(\alpha) + 1 = I_{\sigma'_m}(\alpha) + 1 = I_{\sigma'_{m+1}}(\alpha)$. The PC of α remains the same in all the four states. From the induction hypothesis, we know that $PC_{\sigma_n}(\alpha) = PC_{\sigma'_m}(\alpha)$, and from this the semantics gives us $PC_{\sigma_{n+1}}(x) = PC_{\sigma'_{m+1}}(x)$. Also $E_{\sigma_{n+1}}(x) = \alpha_{I_{\sigma_n}(\alpha)} = \alpha_{I_{\sigma'_m}(\alpha)} = E_{\sigma'_{m+1}}(x)$. Therefore, $\sigma_{n+1} \asymp_{(C,D)} \sigma'_{m+1}$.
3. The *L-cont* statements also do not change.

Therefore Q' is a correspondence relation.

- **Case (Output):** $C_n = C'_m = \gamma := x; C$ and $head(C_n) = \gamma := x$. From the semantics we have $S_{n+1} = \langle out(\gamma, V(E_{\sigma_n}(x), \pi)), \langle \texttt{skip}; C, \sigma_{n+1}, \pi\rangle\rangle$ and $S'_{m+1} = \langle out(\gamma, V(E_{\sigma'_m}(x), \pi')), \langle \texttt{skip}; C, \sigma'_{m+1}, \pi'\rangle\rangle$.

  1. Since outputs happen only under declassifiable conditionals, this case can only occur when $safe(\gamma, C, D)$ and $safe(x, C, D)$, which means that $cds(x, C, D)$ and $dds(x, C, D)$. From the induction hypothesis, $\sigma_n \asymp_{(C,D)} \sigma'_m$. This combines with $cds(x, C, D)$ to give us $E_{\sigma_n}(x) = E_{\sigma'_m}(x)$. It now follows from $dds(x, C, D)$ that $public(E_{\sigma_n}(x), D)$. It then follows from $\pi \approx_D \pi'$ and Lemma 2.11 that $V(E_{\sigma_n}(x), \pi) = V(E_{\sigma'_m}(x), \pi')$. Therefore the outputs in both the transitions are the same.
  2. Since none of the inputs or variables changed, the relevant parts of σ_{n+1} and σ'_{m+1} remain the same. Only the O and PC of the output γ change. $O_{\sigma_{n+1}}(\gamma) = O_{\sigma_n}(\gamma) \cup V(E_{\sigma_n}(x), \pi)$ and $O_{\sigma'_{m+1}}(\gamma) = O_{\sigma'_m}(\gamma) \cup V(E_{\sigma'_m}(x), \pi')$. $PC_{\sigma_{n+1}}(\gamma) = PC_{\sigma_n}(\gamma) \cup PC_{\sigma_n}(x)$ and $PC_{\sigma'_{m+1}}(\gamma) = PC_{\sigma'_m}(\gamma) \cup PC_{\sigma'_m}(x)$. From the observations in the first item, we have $O_{\sigma_{n+1}}(\gamma) = O_{\sigma'_{m+1}}(\gamma)$ and $PC_{\sigma_{n+1}}(\gamma) = PC_{\sigma'_{m+1}}(\gamma)$. Consequently $\sigma_{n+1} \asymp_{(C,D)} \sigma'_{m+1}$.
  3. The *L-cont* commands remain the same.

  Therefore $Q' = Q \cup \{(n+1, m+1)\}$ is a correspondence relation.

- **Case (Assign)**: $C_n = C'_m = x := f(y_1, \ldots, y_k); C$ and $head(C_n) = x := f(y_1, \ldots, y_k)$. From the semantics we have $S_{n+1} = \langle \tau, \langle \texttt{skip}; C, \sigma_{n+1}, \pi\rangle\rangle$ and $S'_{m+1} = \langle \tau, \langle \texttt{skip}; C, \sigma'_{m+1}, \pi'\rangle\rangle$.

  1. Since $o_{n+1} = o'_{m+1} = \tau$ and from induction hypothesis $o(S_0) \ldots o(S_n) = o(S'_0) \ldots o(S'_m)$, we have $o(S_0) \ldots o(S_{n+1}) = o(S'_0) \ldots o(S'_{m+1})$.
  2. If $cds(x, C, D)$, then from Proposition 2.8 it follows that $cds(y_1, C, D) \wedge \ldots \wedge cds(y_k, C, D)$. From the induction hypothesis, $(E_{\sigma_n}(y_1) = E_{\sigma'_m}(y_1)) \wedge \ldots \wedge (E_{\sigma_n}(y_k) = E_{\sigma'_m}(y_k))$ and therefore $E_{\sigma_{n+1}}(x) = \mathbf{f}(E_{\sigma_n}(y_1), \ldots, E_{\sigma_n}(y_k)) = \mathbf{f}(E_{\sigma'_m}(y_1), \ldots, E_{\sigma'_m}(y_k)) = E_{\sigma'_{m+1}}(x)$.

- **Case (Phi):** $C_n = C'_m = x := \phi_c(a,b); C$ and $head(C_n) = \phi_c(a,b)$. From the semantics we have $S_{n+1} = \langle \tau, \langle \texttt{skip}; C, \sigma_{n+1}, \pi \rangle \rangle$ and $S'_{m+1} = \langle \tau, \langle \texttt{skip}; C, \sigma'_{m+1}, \pi' \rangle \rangle$.

  1. The first part of the proof of this case is similar to part (1) of the proof of *Case (Input)*.
  2. If $cds(x, C, D)$, then from Proposition 2.8 we also know that $cds(c, C, D)$, $cds(a, C, D)$ and $cds(b, C, D)$. From the induction hypothesis we know that $E_{\sigma_n}(c) = E_{\sigma'_m}(c)$ and therefore the same expression is assigned to x in both cases. Also from the induction hypothesis $E_{\sigma_n}(a) = E_{\sigma'_m}(a)$ and $E_{\sigma_n}(b) = E_{\sigma'_m}(b)$; consequently $E_{\sigma_{n+1}}(x) = E_{\sigma'_{m+1}}(x)$. Similarly, we have $PC_{\sigma_{n+1}}(x) = PC_{\sigma'_{m+1}}(x)$.

- **Case (Depends):** $C_n = C'_m = depends(\theta, c); C$ and $head(C_n) = depends(\theta, c)$. From the semantics we have that $S_{n+1} = \langle \tau, \langle \texttt{skip}; C, \sigma_{n+1}, \pi \rangle \rangle$ and $S'_{m+1} = \langle \tau, \langle \texttt{skip}; C, \sigma'_{m+1}, \pi' \rangle \rangle$. If $cds(\theta, C, D)$, then from Proposition 2.8 we know that $cds(c, C, D)$. Thus from the induction hypothesis we have $PC_{\sigma_n}(c) = PC_{\sigma'_m}(c)$ and $PC_{\sigma_n}(\theta) = PC_{\sigma'_m}(\theta)$ and consequently $PC_{\sigma_{n+1}}(\theta) = PC_{\sigma'_{m+1}}(\theta)$.

- **Case (If):** $C_n = C'_m = \texttt{if}\ c\ \texttt{then}\ C_{then}\ \texttt{else}\ C_{else}; C$ and $head(C_n) = \texttt{if}\ c\ \texttt{then}\ C_{then}\ \texttt{else}\ C_{else}$. From the semantics we have that $S_{n+1} = \langle \tau, \langle C_{n+1}, \sigma_n, \pi \rangle \rangle$ and $S'_{m+1} = \langle \tau, \langle C'_{m+1}, \sigma'_m, \pi' \rangle \rangle$.

  1. When $cds(c, C, D)$ holds, from our induction hypothesis we have $E_{\sigma_n}(c) = E_{\sigma'_m}(c)$ and consequently $V(E_{\sigma_n}(c), \pi) = V(E_{\sigma'_m}(c), \pi')$. Therefore, in this case both the traces take the same branch and $C_{n+1} = C'_{m+1}$.
  2. When $cds(c, C, D)$ does not hold, C_{n+1} may not be equal to C'_{m+1}. However, by definition of Γ in this case, we have *L-cont*(C_{n+1}) = *L-cont*(C'_{m+1}) = *L-cont*(C).

  Since, additionally, the state, the environment, and the output do not change, we have the fact that $Q' = Q \cup \{(n+1, m+1)\}$ is a correspondence.

- **Case (While):** $C_n = C'_m = \texttt{while}\ \overline{C}\ ;\ c\ \texttt{do}\ C_{while}; C$ and $head(C_n) = \texttt{while}\ \overline{C}\ ;\ c\ \texttt{do}\ C_{while}$. The proof for the `while` case is similar to that of the `if`, except that when $cds(c, C, D)$ does not hold, *L-cont*(C_{n+1}) = *L-cont*(C_n) = *L-cont*(C'_m) = *L-cont*(C'_{m+1}).

Case 2: $head(C_n) \neq head(C'_m)$. Notice that we enter this case only after *Case (If)* or *Case (While)* is encountered in Case 1 and the variable representing the conditional expression c is not in cds. Since our program is safe, the output statements cannot occur inside a conditional expression that is not in cds. Therefore, neither $head(C_n)$ nor $head(C'_m)$ is an output statement. Therefore, we can also note that $o_n = o'_m = \tau$. Similarly, any variables and inputs that are modified by these commands are not in cds. In particular, this means that we have $\langle C_n, \sigma_n, \pi \rangle \xrightarrow{\tau} \langle C_{n+1}, \sigma_{n+1}, \pi \rangle$ and $\langle C'_m, \sigma'_m, \pi' \rangle \xrightarrow{\tau}$

$\langle C'_{m+1}, \sigma'_{m+1}, \pi\rangle$. There are now two subcases, depending on whether $\Gamma(head(C_{n+1}))$ is H or L.

1. When $\Gamma(head(C_{n+1})) = H$, we let $Q' = Q \cup \{(n+1, m)\}$ and show that Q' is a correspondence. For this, we observe that since Q is a correspondence $\sigma_n \asymp_{(C,D)} \sigma_{n+1}$. This follows from the above discussion that none of the modified variables or inputs inside a command that is typed H are cds. From induction hypothesis we have $\sigma_n \asymp_{(C,D)} \sigma'_m$. Therefore, by Lemma 2.17 we have $\sigma_{n+1} \asymp_{(C,D)} \sigma'_m$. Since from the induction hypothesis $o(S_0)\ldots o(S_n) = o(S'_0)\ldots o(S'_m)$ and since $o_{n+1} = \tau$, $o(S_0)\ldots o(S_{n+1}) = o(S'_0)\ldots o(S'_m)$. $\Gamma(head(C_{n+1})) = \Gamma(head(C'_m)) = H$. *L-cont*$(C_{n+1}) =$ *L-cont*(C_n) and therefore from the induction hypothesis and transitivity we have *L-cont*$(C_{n+1}) =$ *L-cont*(C'_m). Therefore the new relation Q' is a correspondence.

2. When $\Gamma(head(C_{n+1})) = L$, we let $Q' = Q \cup \{(n, m+1), \ldots, (n, m+i)\} \cup \{(n+1, m+i+1)\}$ such that for each $j \in \{m+1, \ldots, m+i\}$, $\Gamma(head(C_j)) = H$ and $\Gamma(head(C_{m+i+1})) = L$. Consequently, we have $o_j = \tau$ for $j \in \{m+1, \ldots m+i+1\}$ and therefore each of $o(S'_0)\ldots o(S'_j) = o(S'_0)\ldots o(S'_m)$. From induction hypothesis $o(S_0)\ldots o(S_n) = o(S'_0)\ldots o(S'_m)$. Therefore, we have that $o(S_0)\ldots o(S_n) = o(S'_0)\ldots o(S'_j)$ for each $j \in \{m+1, \ldots, m+i+1\}$. The (output-equivalence) is preserved in Q'. Similar to the first case, none of the modified variables or inputs in the partial runs $S'_m \ldots S'_{m+i+1}$ or $S_n S_{n+1}$ are cds. Therefore $\sigma_n \asymp_{(C,D)} \sigma_{n+1}$ and $\sigma'_m \asymp_{(C,D)} \sigma'_j$ for each $j \in \{m+1, \ldots, m+i+1\}$. From induction hypothesis we have $\sigma_n \asymp_{(C,D)} \sigma'_m$. Therefore, by Lemma 2.17 we have $\sigma_n \asymp_{(C,D)} \sigma'_j$ for $j \in \{m+1, \ldots, m+i\}$ and $\sigma_{n+1} \asymp_{(C,D)} \sigma'_{m+1}$. From the induction hypothesis we know that $(n, m) \in Q$ and Q is a correspondence relation. Therefore *L-cont*$(C_n) =$ *L-cont*$(C'_m) = C_{n+1}$, since from our assumption we know that $head(C_{n+1})$ is the first command typed L. We also have the fact that $\Gamma(head(C_n)) = \Gamma(head(C'_{m+1})) = \ldots = \Gamma(head(C'_{m+i})) = H$, but the *L-cont* of each of these configurations remains constant. So *L-cont*$(C_n) =$ *L-cont*$(C'_{m+1}) =$ $\ldots =$ *L-cont*$(C'_{m+i}) = C_{n+1}$. Taking all these facts into consideration, Q' is a correspondence.

□

A.2 Proof of Theorem 3.6

To prove this theorem we use the following lemmas and notation: we write $C \leqq C'$ if C is a subterm of C' or if C is `skip`. In this context we use $C \equiv C'$ to denote synctatic identity. Since the theorem is not affected by control context annotations on the graph, we omit the u index when we refer to the G function.

Lemma A.1. *If $\langle C_0, \sigma_0, \pi_0\rangle \rightarrow^* \langle C_1; C_2; ...; C_n, \sigma, \pi\rangle$ with C_i non-compositional, then $C_i \leqq C_0$ (for $i = 1, \ldots, n$).*

Proof. By inspecting the semantics one sees that any transition $\langle C, \sigma, \pi\rangle \xrightarrow{o} \langle C', \sigma', \pi'\rangle$ replaces the component $head(C)$ by `skip` (rules (Input), (Output), (Assign), (Phi), (Depends)) or a subterm of the head (rules (If), (While), (Skip)). The remaining components remain unchanged (rule (Seq)). The lemma follows directly. □

Lemma A.2. *If $C \leqq C'$ then $G(C) \precsim G(C')$.*

Proof. We show monotonicity of G by a straightforward induction on the structure of the program C'.

Case 1: If C' is `skip` or $x := \alpha$ or $\gamma := x$ or $x := y$ or $x := f(y_1, \ldots, y_k)$ or $x := \phi_c(a, b)$ or $depends(\theta, c)$ then $C \equiv$ `skip` or $C \equiv C'$. Thus, from the definition of G, it is clear that either $G(C) \simeq \emptyset$ or $G(C) \simeq G(C')$.

Case 2: If C' is C_1 ; C_2, `if` c `then` C_1 `else` C_2 or `while` C_1 ; c `do` C_2 then $C \equiv C'$, $C \leqq C_1$ or $C \leqq C_2$. If $C \equiv C'$ then [def. G] $G(C) \simeq G(C')$. If $C \leqq C_i$, for $i = \{1, 2\}$, then by the induction hypothesis we have that $G(C) \precsim G(C_i)$. But by the definition of G we know that $G(C_i) \precsim G(C')$. Thus, $G(C) \precsim G(C')$. □

Lemma A.3. *For a starting program C_0 such that $\langle C_0, \sigma_0, \pi_0\rangle \rightarrow^* \langle C, \sigma, \pi\rangle$ and any transition $\langle C, \sigma, \pi\rangle \rightarrow \langle C', \sigma', \pi'\rangle$ with $head(C) \leqq C_0$ and $P(\sigma, G(C_0))$ we have that $P(\sigma', G(C_0))$ holds.*

Proof. We use induction on the derivation of the transition. (Note that only (Seq) represents a step, all other cases are base cases.)

Case (Seq): Then $C \equiv C_1; C_2$ for some C_1, C_2 and $\langle C_1, \sigma, \pi\rangle \xrightarrow{o} \langle C_1', \sigma', \pi'\rangle$. As $head(C_1) \equiv head(C) \leqq C_0$ and $P(\sigma, G(C_0))$, we apply the induction hypothesis for this transition, to give us $P(\sigma', G(C_0))$.

Cases (Skip),(While),(If): Then $\sigma' = \sigma$ so $P(\sigma', G(C_0))$.

Case (Depends): Then $C \equiv depends(\theta, c)$ for some θ, c and $\sigma' = \sigma$ except for component PC on θ. Thus we need to show (for $P_{PC}(\sigma', G(C_0))$) that $(\theta, type(\theta))$ is in $G(C_0)$ and $PC_{\sigma'}(\theta) \subseteq cexp(n_\theta)$.

- Note that $depends(\theta, c) \equiv head(C) \leqq C_0$ thus $n_\theta, n_c \in$ [def. G] $G(C)$ and $G(C) \precsim$ [lemma A.2] $G(C_0)$.
- $PC_\sigma(\theta) \subseteq$ [hyp. $P_{PC}(\sigma, G(C_0))$] $cexp(n_\theta)$.
- Note that $(n_c, n_\theta, t) \in G(depends(\theta, c)) \precsim G(C_0)$ (namely with $t =$ `control`) thus $PC_\sigma(c) \subseteq$ [hyp. $P_{PC}(\sigma, G(C_0))$] $cexp(n_c) \subseteq$ [def. $cexp$] $cexp(n_\theta)$.
- Note that $(n_c, n_\theta,$ `control`$) \in G(C_0)$ therefore $\{E_\sigma(c)\} \subseteq$ [hyp. $P_E(\sigma, G(C_0))$] $exp(n_c) \subseteq$ [def. $cexp$] $cexp(n_\theta)$.

Case (Phi): Then $C \equiv x := \phi_c(a, b)$ for some x, c, a, b and $\sigma' = \sigma$ except for components PC and E on x. The reasoning for P_{PC} given the change to PC is as for case (Depends) above. For $P_E(\sigma', G(C_0))$ we need to show that $(x,$ `var`$)$ is in $G(C_0)$ and $E_{\sigma'}(x) \in exp(n_x)$.

- Note that $(x := \phi_c(a,b)) \equiv head(C) \leqq C_0$ thus $(x, \texttt{var}) \in$ [def. G] $G(C) \precsim$ [lemma A.2] $G(C_0)$.
- Note that $E_{\sigma'}(x) = EV(\phi_c(a,b), \sigma)$ is either $E_\sigma(a)$ or $E_\sigma(b)$. As there exists u_1 and u_2 such that (n_a, n_x, ϕ_1, u_1) and (n_b, n_x, ϕ_2, u_2) (definition of G, Lemma A.2) are in $G(C_0)$ we have $\{E_\sigma(a), E_\sigma(b)\} \subseteq exp(n_x)$ from the assumption $P_E(\sigma, G(C_0))$.

Case (Assign): Then $C \equiv x := f(y_1, \ldots, y_k)$ for some $x, f, y_1, \ldots, y_k$ and $\sigma = \sigma'$ except for components E and PC on x. The reasoning for P_{PC} given the change to PC is similar to (but slightly simpler than, as there is no condition) that for case (Depends). For P_E we need to show that $(x, \texttt{var})$ is in $G(C_0)$ and $E_{\sigma'}(x) \subseteq exp(n_x)$.

- Note that $(x := f(y_1, \ldots, y_k)) \equiv head(C) \leqq C_0$ thus $(x, \texttt{var}) \in$ [def. G] $G(C) \precsim$ [lemma A.2] $G(C_0)$.
- Note that $E_\sigma(y_i) \subseteq exp(n_{y_i})$ and $(y_i, x, f_i, u) \in edges(G(C_0))$ (for $i = 1 \ldots, k$ and some u) thus $\{f(E_\sigma(y_1), \ldots, E_\sigma(y_n))\} \subseteq$ [def. exp] $exp(n_x)$.

Case (Output): Then $C \equiv \gamma := x$ for some γ, x and $\sigma = \sigma'$ except for components O and PC on γ. The reasoning for P_{PC} is again the same. For P_O we need to show that $(\gamma, \texttt{out}) \in G(C_0)$ and $O_\sigma(\gamma) \cup \{E_\sigma(x)\} \subseteq exp(n_\gamma)$.

- Note that $(\gamma := x) \equiv head(C) \leqq C_0$ thus we have that $(\gamma, \texttt{out}) \in$ [def. G] $G(C) \precsim$ [lemma A.2] $G(C_0)$.
- The assumption $P_O(\sigma, G(C_0))$ already gives $O_\sigma(\gamma) \subseteq exp(n_\gamma)$ while $P_E(\sigma, G(C_0))$ gives $\{E_\sigma(x)\} \subseteq exp(n_x) \subseteq$ [$(n_x, n_\gamma, t) \in G(C_0)$, def. exp] $exp(n_\gamma)$.

Case (Input): Then $C \equiv x := \alpha$ for some x, α and $\sigma = \sigma'$ except for components E and PC on x and I on α. The reasoning for P_{PC} and P_E is as before. For $P_I(\sigma', G(C_0))$ we need to show that $(\alpha, \texttt{in}) \in G(C_0)$. As $(x := \alpha) \equiv head(C) \leqq C_0$ and $(\alpha, \texttt{in}) \in G(C)$ this is direct from Lemma A.2. □

We can now give the proof of the theorem:

Proof. By induction on the length of the (sub)trace we show that if:

$$\langle C_0, \sigma_0, \pi_0 \rangle \quad \rightarrow^* \quad \langle C, \sigma, \pi \rangle$$

in which $\rightarrow^*$ represents zero or more steps taken in the operational semantics, then $P(\sigma, G(C_0))$.

Base: For the case of zero steps taken from the initial configuration is trivial as no variable is yet defined in E, O and PC, and $I(\alpha) = 1$ for all input channels α thus $P(\sigma_0, G(C_0))$.

Step: For the step, consider any (sub) trace

$$\langle C_0, \sigma_0, \pi_0 \rangle \rightarrow^* \langle C_{n-1}, \sigma_{n-1}, \pi_{n-1} \rangle \rightarrow \langle C_n, \sigma_n, \pi_n \rangle$$

with $P(\sigma_{n-1}, G(C_0))$. Then, by Lemma A.1 we have that $head(C_{n-1}) \leqq C_0$. Thus, using Lemma A.3 we get $P(\sigma_n, G(C_0))$. □

A.3 Proof of Theorem 3.13

For this theorem, we first present a supporting definition and two supporting lemmas.

Definition A.4 (Well-formed graph)**.** *An expression graph is said to be well-formed if:*

1. *Each* `var` *node has its set of incoming edges composed of either:*
   (a) *Exactly one* `plain` *edge.*
   (b) k *edges labeled* $f_i, i = 1..k$*, for function* f *with* k *arguments.*
   (c) *Exactly 2* ϕ *edges (*ϕ_1 *and* ϕ_2*) and one* `control` *edge.*
2. *Each* `out` *node has its set of incoming edges composed of 0 or more* `plain` *edges and, if a non-zero number of these exist, additionally 0 or more* `control` *edges.*
3. *Each* `in` *node has its set of incoming edges composed of 0 or more* `control` *edges.*
4. *Each* `const` *node has indegree equals to zero.*
5. *Nodes of types* `var` *and* `const` *have their set of outgoing edges composed of 0 or more edges, of any type.*
6. *Each* `in` *node has its set of outgoing edges composed of 0 or more* `plain` *edges.*
7. *Each* `out` *node has outdegree equal to zero.*

Lemma A.5. *For a well-formed graph, if* $n' \ (\xrightarrow{\tau})^* \ n$ *then* $exp(n') \subseteq exp(n)$.

Proof. Definition of well formed graph guarantees that a node with an incoming τ edge is either an output or a variable node. The definition of exp then directly gives that if $n' \xrightarrow{\tau} n$ then $exp(n') \subseteq exp(n)$. Simple induction on length of the sequence of τ steps gives the result. □

Lemma A.6. *For a well-formed graph we have that if* $(n^i)_{i=1..k}(\xrightarrow{\tau})^* \xrightarrow{f_i} n'(\xrightarrow{\tau})^* n$ *and* $e^i \in exp(n^i)$ *then* $f(e^i, \ldots, e^k) \in exp(n)$.

Proof. By Lemma A.5 it is sufficient to show that if for some immediate predecessors of n, $n'^i (i = 1..k)$, we have $(n'^i)_{i=1..k} \xrightarrow{f_i} n$ and $e^i \in exp(n^i)$ then $f(e^i, \ldots, e^k) \in exp(n)$. As n must be a `var` node (Definition A.4) this is directly clear from the definition of exp. □

Now we can present the proof of the Theorem:

Proof. Note that we can construct exp as the union $exp(n) = \bigcup_{m\in\mathbb{N}} exp^m(n)$ with $exp^m(n)$ the monotonically growing sequence given by (only possible nodes on the program graph shown):

$$\begin{array}{rcl} exp^0 & = & \emptyset \\ exp^{m+1}(n_N) & = & \{N\} \\ exp^{m+1}(n_\alpha) & = & \{\alpha_i \mid i \in \mathbb{N}\} \\ exp^{m+1}(n_\gamma) & = & \bigcup\limits_{n'\to n_\gamma} exp^m(n') \\ exp^{m+1}(n_x) & = & \\ \Psi_{n_x}\left(\begin{array}{cl} \bigcup\limits_{n'\xrightarrow{\tau}n_x} & exp^m(n') \\ \cup & \{f(e^1,\ldots,e^k) \mid \forall n^1,\ldots,n^k : n^i \xrightarrow{f_i} n_x \wedge e^i \in exp_g(n^i)\} \end{array}\right) & \end{array}$$

Because any element of $exp(n)$ is introduced to $exp^m(n)$ at some finite stage m, it is sufficient to show that for all m we have:

$$\forall n, n' : (n, n') \in \mathcal{R} \Rightarrow exp^m(n) \subseteq exp(n') \tag{A.1}$$

which is done by induction on m. For $m = 0$ there is nothing to prove. Assume formula A.1 holds for some $m \in \mathbb{N}$. We now show this property also holds for $m + 1$ by treating the four possible types of node n:

n_N Assume $(n_N, n') \in \mathcal{R}$ then $n_N \simeq n'$. Thus n' is a constant node with label N or $*$. This gives $N \in \exp(n')$. But then:

$$exp^{m+1}(n_N) = \{N\} \subseteq exp(n')$$

n_α The same as the previous case.

n_γ Assume $(n_\gamma, n') \in \mathcal{R}$ and $n^1, \ldots, n^k$ are the nodes that can reach n_γ (i.e., $n^i \to n_\gamma$) then $n_\gamma \simeq n'$ (and thus the type of n' is also `out`) and there exist nodes $n'^1, \ldots n'^k$ such that $(n^i, n'^i) \in \mathcal{R}$ ($i = 1..k$) and $n'^i \ (\xrightarrow{\tau})^* \ n'$. But then:

$$\begin{array}{rcl} exp^{m+1}(n_\gamma) & = & \text{(def.}exp\text{)} \bigcup_{i=1..k} exp^m(n^i) \\ & \subseteq & \text{(ind.hyp.)} \bigcup_{i=1..k} exp(n'^i) \\ & \subseteq & \text{(lemma A.5)} \ exp(n') \end{array}$$

n_x Assume $(n_x, n') \in \mathcal{R}$. First, we need to show that for sets of expressions S and S', we have that $S \subseteq S' \Rightarrow \Psi_{n_x}(S) \subseteq \Psi_{n'}(S')$. But since Ψ is a filter, its monotonicity is trivial, i.e. $\Psi_n(S) \subseteq \Psi_n(S')$, for some n. With that, we only have to show that $\Psi_{n_x}(S) \subseteq \Psi_{n'}(S)$. We have that $n_x \sim_{g,d} n'$, thus from Definitions 3.9 and 3.3 we conclude that $\forall \alpha : (\alpha, n') \in uni(d) \Rightarrow (\alpha, n_x) \in uni(g)$. It follows from (def.$\Psi$) that $\Psi_{n_x}(S) \subseteq \Psi_{n'}(S)$. Now we proceed to show that $exp^m(n_x) \subseteq exp(n')$. The case where $n' \xrightarrow{\tau} n$ is the same as the previous case. Assume $n^1, \ldots, n^k$ are the

nodes that can reach n_x by an f step ($n^i \xrightarrow{f_i} n_x$), then n' is also of type `var` and there exist nodes $(n^i, n'^i) \in \mathcal{R}$ and $(n'^i)_{i=1,,k} \overset{f}{\Rightarrow} n'$. But then:

$$\begin{array}{rcl} exp^{m+1}(n_x) & = & \text{(def.}exp\text{)}\ \{f(e^1,\ldots,e^k) \mid e^i \in exp^m(n^i)\ \text{(for } i = 1..k)\} \\ & \subseteq & \text{(ind.hyp.)}\ \{f(e^1,\ldots,e^k) \mid e^i \in exp(n'^i)\ \text{(for } i = 1..k)\} \\ & \subseteq & \text{(lemma A.6)}\ exp(n') \end{array}$$

□

A.4 Proof of Theorem 3.15

Proof. For all 3 cases, we do the following: we apply Theorem 3.6 on the first component of the LHS; use definitions of $\overline{dds}$ and $\overline{cds}$ to unfold the second component of the LHS; and unfold definition of $\overline{public}$ on the RHS. The resulting equation, on all cases, will be a variation of Theorem 3.13, and thus proved the same way. □

A.5 Proof of Theorem 3.21

Proof. Items 1 and 2 of Definition 3.17 are achieved by Definition 3.18 and Lemma 3.20, respectively. For item 3, we basically have to show that, for some program C and policy d:

$$\overline{valid}(G(C), d) \Rightarrow valid(C, int(d))$$

Unfolding the definitions of $\overline{valid}$, $\overline{safe}$, $valid$ and $safe$, we have:

$$\begin{array}{l} \forall n_\gamma \in nodes(G(C)) : \overline{dds}(n_\gamma, G(C), d) \wedge \overline{cds}(n_\gamma, G(C), d) \Rightarrow \\ \quad \forall \gamma \in \textbf{Out} : dds(\gamma, C, int(d)) \wedge cds(\gamma, C, int(d)) \end{array}$$

However, by the definition of G and Theorem 3.6, we know that every node $n_\gamma \in nodes(G(C))$ represents an output channel $\gamma \in \textbf{Out}$. Thus, we can remove the quantifiers on both sides of the equation:

$$\overline{dds}(n_\gamma, G(C), d) \wedge \overline{cds}(n_\gamma, G(C), d) \Rightarrow dds(\gamma, C, int(d)) \wedge cds(\gamma, C, int(d))$$

Now, we use Theorem 3.15 on the left side of the equation:

$$\begin{array}{l} \forall \sigma \in states(C), e \in O_\sigma(\gamma), e' \in PC_\sigma(\gamma) : \overline{public}(e, d) \wedge \overline{public}(e', d) \Rightarrow \\ \quad dds(\gamma, C, int(d)) \wedge cds(\gamma, C, int(d)) \end{array}$$

Unfolding the remaining definitions of dds and cds on the program we have:

$$\begin{array}{l} \forall \sigma \in states(C), e \in O_\sigma(\gamma), e' \in PC_\sigma(\gamma) : \overline{public}(e, d) \wedge \overline{public}(e', d) \Rightarrow \\ \quad \forall \sigma \in states(C), e \in O_\sigma(\gamma), e' \in PC_\sigma(\gamma) : public(e, int(d)) \wedge public(e', int(d)) \end{array}$$

Finally, we remove the quantifiers common to both sides to obtain:

$$\overline{public}(e, d) \wedge \overline{public}(e', d) \Rightarrow public(e, int(d)) \wedge public(e', int(d))$$

which is Lemma 3.19. □

APPENDIX B

Source Code

In this Appendix we include the source code of experiments performed in Section 4.5.1. Note that source code is sanitized for readability. For instance, preamble of files, containing lists of `import` directives are omitted. We present the source code of all benchmark examples used, as well as the simple runtime enforcer implementation. On the benchmark examples, code comments identify lines that were included during code injection. Enforced and non-enforced versions of the benchmarks differ only by those lines. Finally, note that the implementation of the runtime enforcer aims for simplicity, and its performance can be improved in a number of ways.

Example 1

```
1  public class Example1 {
2
3    void processContactList() throws Exception {
4      String[] clist;
5      String contact, text, addr;
6      int counter, age;
7
8      clist = getContactList();
9      counter = 0;
10
11     for (int i=0; i<clist.length; i++) {
12       contact = clist[i];
13       age = getAge(contact);
14
15       if (age > 45) counter++;
16     }
17
18     text = "I have " + counter + " contacts over age 45.";
19
20     addr = readFromInput();
21     // Code injection: following 1 line has been injected
22     RuntimeEnforcer.checkOutput(21, Example1Enforced.class.getMethod("
           sendSMS", String.class, String.class), new Object[]{addr,text},
            new Label("hi", 10));
23     sendSMS(addr,text);
24   }
25
```

```
26    String[] getContactList() throws Exception {
27      ArrayList<String> contacts = new ArrayList<String>();
28      Scanner s = new Scanner(new File(Constants.FILE_PATH + "contacts.
            ex1"));
29
30      while (s.hasNext()) contacts.add(s.nextLine());
31
32      s.close();
33
34      String[] contArray = new String[contacts.size()];
35      contArray = contacts.toArray(contArray);
36      return contArray;
37    }
38
39    int getAge(String contact) {
40      return Integer.parseInt(contact.substring(contact.lastIndexOf(' ')
            +1));
41    }
42
43    String readFromInput() throws Exception {
44      Scanner s = new Scanner(new File(Constants.FILE_PATH + "address.ex1
            "));
45      String address = s.nextLine();
46      s.close();
47
48      return address;
49    }
50
51    public void sendSMS(String addr, String text) throws Exception {
52      BufferedWriter out = new BufferedWriter(new FileWriter(new File(
            Constants.FILE_PATH + "output.ex1")));
53      out.write("SMS sent\n");
54      out.write("To: " + addr + "\n");
55      out.write("Content: " + text + "\n");
56      out.close();
57    }
58
59    public static void main(String[] args) throws Exception {
60      new Example1().processContactList();
61    }
62  }
```

Example 2

```
1   public class Example2 {
2
3     public void run() throws Exception {
4       String secureConn, myLoc, myTz, otherTz;
5
6       secureConn = secConnect("otherhost.somewhere.com");
7       myLoc = getLocation();
8       myTz = timezone(myLoc);
```

```
9       otherTz = recv(secureConn);
10      // Code injection: following 1 line has been injected
11      RuntimeEnforcer.storeDataLabel(14, otherTz);
12
13      if (myTz.equals(otherTz)) {
14        String packet = "ACK";
15        // Code injection: following 1 line has been injected
16        RuntimeEnforcer.setDataLabel(20, packet, new Label("low", 1));
17        send(packet,secureConn);
18        String otherLoc = recv(secureConn);
19        boolean near = isNear(myLoc, otherLoc);
20        if (near) {
21          // Code injection: following 1 line has been injected
22          RuntimeEnforcer.checkInput(25, this.getClass().getMethod("print
              ", String.class), new Object[]{"Host is nearby!"}, 14);
23          print("Host is nearby!");
24        }
25      }
26    }
27
28    public String secConnect(String addr) throws Exception {
29      BufferedWriter out = new BufferedWriter(new FileWriter(new File(
            Constants.FILE_PATH + "connection.ex2")));
30      out.write("Secure connect with host: " + addr + "\n");
31
32      Scanner s = new Scanner(new File(Constants.FILE_PATH + "key.ex2"));
33      String key = s.nextLine();
34      s.close();
35
36      out.write("Key: " + key + "\n");
37      out.close();
38
39      return Constants.FILE_PATH + "connection.ex2";
40    }
41
42    public String getLocation() throws Exception {
43      Scanner s = new Scanner(new File(Constants.FILE_PATH + "location.
            ex2"));
44      String loc = s.nextLine();
45      s.close();
46
47      return loc;
48    }
49
50    // simulating the kind of computation performed by this type of
          function
51    public String timezone(String loc) {
52      double lon = Double.parseDouble(loc.substring(loc.lastIndexOf(' ')
            +1));
53      lon += 180;
54      int i = (int)(lon / 30);
55      String str = "undef";
56      switch (i) {
```

```
57        case 0:
58          str = "A"; break;
59        case 1:
60          str = "B"; break;
61        case 2:
62          str = "C"; break;
63        case 3:
64          str = "D"; break;
65        case 4:
66          str = "E"; break;
67        case 5:
68          str = "F"; break;
69        case 6:
70          str = "G"; break;
71        case 7:
72          str = "H"; break;
73        case 8:
74          str = "I"; break;
75        case 9:
76          str = "J"; break;
77        case 10:
78          str = "K"; break;
79        case 11:
80          str = "L"; break;
81      }
82      return str;
83    }
84
85    public String recv(String conn) throws Exception {
86      String str = null;
87      Scanner s = new Scanner(new File(conn));
88      s.nextLine();
89      s.nextLine();
90
91      if (s.hasNextLine()) str = "56.5376547, -47.1392201";
92      else str = "E";
93
94      s.close();
95      return str;
96    }
97
98    public void send(String text, String conn) throws Exception  {
99      BufferedWriter out = new BufferedWriter(new FileWriter(new File(
            conn), true));
100
101     out.write("Sent: " + text + "\n");
102     out.close();
103   }
104
105   public void print(String text) {
106     System.out.println(text);
107   }
108
```

```
109   public boolean isNear(String loc1, String loc2) {
110     double x1,y1,x2,y2;
111     x1 = Double.parseDouble(loc1.substring(0, loc1.indexOf(',')));
112     y1 = Double.parseDouble(loc1.substring(loc1.lastIndexOf(' ')+1));
113     x2 = Double.parseDouble(loc2.substring(0, loc2.indexOf(',')));
114     y2 = Double.parseDouble(loc2.substring(loc2.lastIndexOf(' ')+1));
115
116     return (Math.abs(x1-x2) <= 1) && (Math.abs(y1-y2) <= 1);
117   }
118
119   public static void main(String[] args) throws Exception {
120     new Example2().run();
121   }
122 }
```

Example 3

```
1  public class Example3 {
2
3    public void run() throws Exception {
4      int sum, num;
5      double avg;
6      Scanner db;
7
8      sum = num = 0;
9      db = openDBConnection();
10
11     while (db.hasNextLine()) {
12       // Code injection: following 1 line has been injected
13       RuntimeEnforcer.countIter(15);
14       String rec = fetch(db);
15       int prop = getProperty(rec);
16       sum += prop;
17       num++;
18     }
19
20     avg = (double)sum / (double)num;
21     // Code injection: following 1 line has been injected
22     RuntimeEnforcer.eval(25, RuntimeEnforcer.iterCount[15] >= 25);
23     output("" + avg);
24     db.close();
25   }
26
27   public Scanner openDBConnection() throws Exception {
28     Scanner s = new Scanner(new File(Constants.FILE_PATH + "connection.
           ex3"));
29     String status = s.nextLine();
30     if (!status.equals("Connection OK")) throw new Exception("Wrong
           status!");
31
32     return s;
33   }
```

```
34
35    public String fetch(Scanner db) {
36      return db.nextLine();
37    }
38
39    public void output(String data) throws Exception {
40      BufferedWriter out = new BufferedWriter(new FileWriter(new File(
            Constants.FILE_PATH + "output.ex3")));
41
42      out.write("Average: " + data + "\n");
43      out.close();
44    }
45
46    public int getProperty(String rec) {
47      return Integer.parseInt(rec.substring(rec.lastIndexOf(' ')+1));
48    }
49
50    public static void main(String[] args) throws Exception {
51      // Code injection: following 1 line has been injected
52      RuntimeEnforcer.initCounter(60);
53      new Example3().run();
54    }
55  }
```

FileCopy

```
1   public class FileCopy {
2
3     public void run() throws Exception {
4       FileInputStream in = new FileInputStream(new File(Constants.
            FILE_PATH + "input.fc"));
5       FileOutputStream out = new FileOutputStream(new File(Constants.
            FILE_PATH + "output.fc"));
6
7       int i;
8       byte[] b = new byte[1024];
9
10      while ((i = in.read(b)) != -1) {
11        // Code injection: following 2 lines have been injected
12        RuntimeEnforcer.storeDataLabel(19, b);
13        RuntimeEnforcer.setDataLabel(22, b, RuntimeEnforcer.inLabel.get
              (19));
14        out.write(b, 0, i);
15      }
16
17      in.close();
18      out.close();
19    }
20
21    public static void main(String[] args) throws Exception {
22      new FileCopy().run();
23    }
```

```
24 }
```

FileEncrypt

```
1  public class FileEncrypt {
2    Cipher cipher;
3    SecretKeySpec aesKeySpec;
4
5    public void run() throws Exception {
6      cipher = Cipher.getInstance("AES");
7      KeyGenerator kgen = KeyGenerator.getInstance("AES");
8      kgen.init(128);
9      SecretKey key = kgen.generateKey();
10     byte[] aesKey = key.getEncoded();
11     aesKeySpec = new SecretKeySpec(aesKey, "AES");
12
13     File input = new File(Constants.FILE_PATH + "input.fc");
14     File enc = new File(Constants.FILE_PATH + "enc.fc");
15     File dec = new File(Constants.FILE_PATH + "dec.fc");
16
17     encrypt(input, enc);
18     decrypt(enc, dec);
19   }
20
21   private void encrypt(File in, File out) throws Exception {
22     cipher.init(Cipher.ENCRYPT_MODE, aesKeySpec);
23
24     FileInputStream is = new FileInputStream(in);
25     CipherOutputStream os = new CipherOutputStream(new FileOutputStream
         (out), cipher);
26
27     copy(is, os);
28
29     is.close();
30     os.close();
31   }
32
33   private void decrypt(File in, File out) throws Exception {
34     cipher.init(Cipher.DECRYPT_MODE, aesKeySpec);
35
36     CipherInputStream is = new CipherInputStream(new FileInputStream(in
         ), cipher);
37     FileOutputStream os = new FileOutputStream(out);
38
39     copy(is, os);
40
41     is.close();
42     os.close();
43   }
44
45   private void copy(InputStream in, OutputStream out) throws Exception
       {
```

```
46      int i;
47      byte[] b = new byte[1024];
48
49      while ((i = in.read(b)) != -1) {
50        // Code injection: following 2 lines have been injected
51        RuntimeEnforcer.storeDataLabel(67, b);
52        RuntimeEnforcer.setDataLabel(70, b, RuntimeEnforcer.inLabel.get
              (19));
53        out.write(b, 0, i);
54      }
55    }
56
57    public static void main(String[] args) throws Exception {
58      new FileEncrypt().run();
59    }
60  }
```

InformationGather

```
1   public class InformationGather {
2
3     public void run() throws Exception {
4
5       byte[] data = null;
6
7       for (int num=0; num <= 9; num++) {
8         String filename = "input." + num;
9         FileInputStream in = new FileInputStream(new File(Constants.
              FILE_PATH + filename));
10        ByteArrayOutputStream out = new ByteArrayOutputStream();
11
12        int i;
13        byte[] b = new byte[1024];
14
15        while ((i = in.read(b)) != -1) out.write(b, 0, i);
16        // Code injection: following 1 line has been injected
17        RuntimeEnforcer.storeChLabel(25, in.getClass().getMethod("read",
              byte[].class), new Object[]{in,b});
18
19        byte[] file = out.toByteArray();
20
21        in.close();
22        out.close();
23
24        if (data == null) data = file;
25        else for (int j=0; j<file.length; j++) data[j] += file[j];
26      }
27
28      FileOutputStream os = new FileOutputStream(new File(Constants.
            FILE_PATH + "output.ig"));
29      // Code injection: following 1 line has been injected
```

```
30      RuntimeEnforcer.checkOutput(39, os.getClass().getMethod("write",
            byte[].class), new Object[]{data}, RuntimeEnforcer.inLabel.get
            (25));
31      os.write(data);
32      os.close();
33    }
34
35    public static void main(String[] args) throws Exception {
36      new InformationGather().run();
37    }
38  }
```

Loops

```
 1  public class Loops {
 2
 3    public void run() throws Exception {
 4      FileInputStream in = new FileInputStream(new File(Constants.
            FILE_PATH + "input.fc"));
 5      ByteArrayOutputStream out = new ByteArrayOutputStream();
 6
 7      int i;
 8      byte[] b = new byte[1024];
 9
10      while ((i = in.read(b)) != -1) out.write(b, 0, i);
11      // Code injection: following 1 line has been injected
12      RuntimeEnforcer.compareCh(20, in.getClass().getMethod("read", byte
            [].class), new Object[]{in,b}, FileInputStream.class.getMethod(
            "read", byte[].class), null);
13
14      byte[] data = out.toByteArray();
15
16      in.close();
17      out.close();
18
19      int buffer = 0;
20
21      for (int j=0; j<data.length; j++) {
22        // Code injection: following 1 line has been injected
23        RuntimeEnforcer.countIter(30);
24        buffer ^= data[j];
25      }
26
27      for (int j=0; j<data.length/2; j++) {
28        // Code injection: following 1 line has been injected
29        RuntimeEnforcer.countIter(35);
30        int x = data[j] % 14;
31        int y = data[j] >> 2;
32        buffer &= (x ^ y);
33      }
34
35      for (int j=data.length/2; j<data.length; j++) {
```

```
36        // Code injection: following 1 line has been injected
37        RuntimeEnforcer.countIter(42);
38        data[j] <<= 2;
39      }
40
41      for (int j=0; j<data.length; j += 2) {
42        // Code injection: following 1 line has been injected
43        RuntimeEnforcer.countIter(47);
44        data[j] ^= 0xAA;
45      }
46
47      for (int j=0; j<data.length; j += 10) {
48        // Code injection: following 1 line has been injected
49        RuntimeEnforcer.countIter(52);
50        data[j] ^= buffer;
51      }
52
53      FileOutputStream os = new FileOutputStream(new File(Constants.
            FILE_PATH + "output.loops"));
54      // Code injection: following 6 lines have been injected
55      boolean eval = RuntimeEnforcer.iterCount[30] >= (data.length-2);
56      eval = eval && RuntimeEnforcer.iterCount[35] < data.length;
57      eval = eval && RuntimeEnforcer.iterCount[42] < data.length;
58      eval = eval && RuntimeEnforcer.iterCount[47] < RuntimeEnforcer.
            iterCount[30];
59      eval = eval && RuntimeEnforcer.iterCount[52] < RuntimeEnforcer.
            iterCount[47];
60      RuntimeEnforcer.eval(64, eval);
61      os.write(data);
62      os.close();
63    }
64
65    public static void main(String[] args) throws Exception {
66      // Code injection: following 1 line has been injected
67      RuntimeEnforcer.initCounter(70);
68      new Loops().run();
69    }
70  }
```

Statistics

```
1   public class Statistics {
2
3   public void run() throws Exception {
4
5       byte[] data = null;
6
7       for (int num=0; num <= 9; num++) {
8         // Code injection: following 1 line has been injected
9         RuntimeEnforcer.countIter(17);
10        String filename = "input." + num;
```

```
11        FileInputStream in = new FileInputStream(new File(Constants.
              FILE_PATH + filename));
12        ByteArrayOutputStream out = new ByteArrayOutputStream();
13
14        int i;
15        byte[] b = new byte[1024];
16
17        while ((i = in.read(b)) != -1) out.write(b, 0, i);
18        // Code injection: following 1 line has been injected
19        RuntimeEnforcer.compareCh(26, in.getClass().getMethod("read",
              byte[].class), new Object[]{in,b}, FileInputStream.class.
              getMethod("read", byte[].class), new Object[]{Constants.
              FILE_PATH + filename});
20
21        byte[] file = out.toByteArray();
22
23        in.close();
24        out.close();
25
26        if (data == null) data = file;
27        else for (int j=0; j<file.length; j++) data[j] += file[j];
28      }
29
30      for (int j=0; j<data.length; j++) data[j] /= 10;
31
32      FileOutputStream os = new FileOutputStream(new File(Constants.
            FILE_PATH + "output.s"));
33      // Code injection: following 1 line has been injected
34      RuntimeEnforcer.eval(42, RuntimeEnforcer.iterCount[17] >= 5);
35      os.write(data);
36      os.close();
37    }
38
39    public static void main(String[] args) throws Exception {
40      // Code injection: following 1 line has been injected
41      RuntimeEnforcer.initCounter(50);
42      new Statistics().run();
43    }
44  }
```

Runtime Enforcer

```
1   public class RuntimeEnforcer {
2
3     public static int[] iterCount;
4     public static HashMap<Integer, Label> inLabel;
5     private static LabelingSystem labelingSystem;
6
7     static {
8       inLabel = new HashMap<Integer, Label>();
9       labelingSystem = new MockLabelingSystem();
10    }
```

```
11
12    public static void initCounter(int progSize) {
13      iterCount = new int[progSize];
14    }
15
16    public static void countIter(int progPoint) {
17      iterCount[progPoint]++;
18    }
19
20    public static void eval(int progPoint, boolean expr) {
21      if (!expr) abort("Expression on iteration count not satisfied.",
            progPoint);
22    }
23
24    public static void compareCh(int progPoint, Method cmd, Object[] args
        , Method otherCmd, Object[] otherArgs) {
25      if (labelingSystem.compareChannel(cmd, args, otherCmd, otherArgs)
            != ChannelSimilarity.OK) abort("Input channels not the same.",
            progPoint);
26    }
27
28    public static void storeDataLabel(int progPoint, Object data) {
29      inLabel.put(progPoint, labelingSystem.getDataLabel(data));
30    }
31
32    public static void storeChLabel(int progPoint, Method cmd, Object[]
        args) {
33      Label l = labelingSystem.getChannelLabel(cmd, args);
34      if (inLabel.containsKey(progPoint)) l = max(new Label[]{l,inLabel.
            get(progPoint)});
35      inLabel.put(progPoint, l);
36    }
37
38    public static void checkInput(int progPoint, Method cmd, Object[]
        args, int pp) {
39      if (inLabel.get(pp).getValue() > labelingSystem.getChannelLabel(cmd
            , args).getValue()) abort("Input label higher than output label
            .", progPoint);
40    }
41
42    public static void setDataLabel(int progPoint, Object data, Label
        label) {
43      if (!labelingSystem.setDataLabel(data, label)) abort("Data label
            could not be set.", progPoint);
44    }
45
46    public static void checkOutput(int progPoint, Method cmd, Object[]
        args, Label label) {
47      if (labelingSystem.getChannelLabel(cmd, args).getValue() < label.
            getValue()) abort("Output label lower than input label.",
            progPoint);
48    }
49
```

```
50    private static void abort(String str, int pp) {
51      System.err.println("Runtime enforcer violation: " + str);
52      System.err.println("Program point of the check: " + pp);
53      System.err.println("Aborting!");
54      System.exit(1);
55    }
56
57    public static Label max(Label[] labels) {
58      Label max = labels[0];
59      for (int i=1; i<labels.length; i++) {
60        if (labels[i].getValue() > max.getValue()) max = labels[i];
61      }
62      return max;
63    }
64  }
```

List of Symbols

Expression-Matching Framework

IO	Domain of input/output channels, **IO** $=$ **In** $\cup$ **Out** (page 24).
Var	Domain of program variables (page 24).
$Prog$	Domain of programs (page 24).
Σ	Domain of program states (page 25).
Π	Domain of execution environments (page 25).
Ω	Domain of configurations (page 25).
Obs	Domain of configuration transition labels (page 25).
x, y, z, a, b	Program variables, members of **Var** (page 24).
c	Boolean program variable, used in conditionals (page 24).
N	Constant value (page 24).
f	Function, represented as a syntactic object (page 26).
f	Semantic evaluation of function f to a value (page 26).
α, β	Input channels, members of **In** (page 24).
γ, δ	Output channels, members of **Out** (page 24).
θ	Input or output channel, member of **IO** (page 24).
ρ	Input/output channel, or program variable, member of **Var** $\cup$ **IO** (page 24).
θ_n	n-th value read from/written to channel θ (page 24).
C	Program, member of $Prog$ (page 24).
C_i	Sub-program of C (page 24).

$head(C)$	First non-compositional statement of C, a.k.a. the active command of C (page 24).
σ	Program state, $\sigma \in \Sigma = \langle E, I, O, PC \rangle$ (page 25).
E_σ	Mapping from variables to expressions on indexed input channels (page 25).
I_σ	Mapping from input channels to numeric index of next value to be read (page 25).
O_σ	Mapping from output channels to sets of expressions on indexed inputs that could be sent over that output (page 25).
PC_σ	Mapping from variables and channels (input/output) to sets of expressions on indexed inputs which are conditionally dependent to the mapped variable/channel (page 25).
π	Execution environment, $\pi \in \Pi = \mathbf{In} \times \mathbb{N} \to \mathbf{Val}$ (page 25).
ω	Configuration, $\omega \in \Omega = \langle C, \sigma, \pi \rangle$ (page 25).
τ	Non-observable transition, $\tau \in \mathbf{Obs}$ (page 25).
o	Configuration transition label, $o \in \mathbf{Obs}$ is either τ or $out(\gamma, v)$ for some output channel γ and value v (page 25).
$f[x \overset{\odot}{\leftarrow} n]$	A variant of f where the value assigned to x is $f(x) \odot n$, where $\odot$ is any operator of the right type. Also, $f[x \leftarrow n](x) = n$ (page 25).
σ_{init}	Initial state (page 27).
t, t'	Runs of a program C in an environment π (page 27).
$o(t)$	The sequence of (visible) output actions in t (page 28).
$t \equiv_{\text{out}} t'$	Holds if $o(t) = o(t')$ (page 28).
$\omega(t)$	The sequence of configurations in t (page 28).
T, T'	Sets of runs (page 28).
$T \equiv_{\text{out}} T'$	Holds if $\forall t \in T : \exists t' \in T' : t \equiv_{\text{out}} t'$ (page 28).
$Run(C, \pi)$	All the runs of program C in environment π (page 28).
D	Set of declassifiable expressions, $D \subseteq \mathbf{Exp}\langle \mathbf{In} \times \mathbb{N} \rangle$ (page 28).
$public(e, D)$	Expression e is public according to D (page 28).

$states(C)$	Every possible state program C can achieve (page 28).
$dds(\rho, C, D)$	ρ is data dependency safe in every state achieved by C, with respect to D (page 29).
$cds(\rho, C, D)$	ρ is control dependency safe in every state achieved by C, with respect to D (page 29).
$safe(\rho, C, D)$	ρ is safe in every state achieved by C, with respect to D (page 29).
$valid(C, D)$	C is valid with respect to D (page 29).
$\pi_1 \approx_D \pi_2$	Environments π_1 and π_2 are D-equivalent (page 30).
$\mathcal{R}(\pi, D)$	The knowledge of π revealed by D (page 30).
$\mathcal{K}(\pi, C)$	The knowledge of π that can observed in C (page 31).
PCR	The Policy Controlled Release property (page 31).
$\Gamma_C(C')$	The type of statement C' of program C, $\Gamma_C(C') \in \{H, L\}$ (page 32).
$\sigma_1 \asymp_{(C,D)} \sigma_2$	States σ_1 and σ_2 are compatible for C and D (page 33).
L-cont(C)	The low continuation of C, i.e. its first sub-statement which is not typed high (page 33).
Q	A correspondence relation between two runs t and t' (page 34).

Graph-Based Implementation

$\mathcal{G}$	Domain of program expression graphs (page 41).
Vertex	Domain of graph vertices (nodes) (page 41).
Edge	Domain of graph edges (page 41).
$\mathcal{D}$	Domain of policy expression graphs (page 45).
g	Program expression graph, $g \in \mathcal{G} = (V, E)$ (page 41).
V	Set of vertices, $V \subseteq$ **Vertex** (page 41).
E	Set of edges, $E \subseteq$ **Edge** (page 41).
$n = (l, t)$	A graph vertex (node), $n \in$ **Vertex**, where l is a label (name) and t is a type, $t \in \{\texttt{var}, \texttt{in}, \texttt{out}, \texttt{const}\}$ (page 41).
$e = (n, n', t, u)$	A graph edge, $e \in$ **Edge**, where n and n' are the origin and destination vertices, respectively, t is a type $t \in \{\texttt{plain}, \texttt{control}, \phi_i, f_i\}$, where f is any function name and $i \in \mathbb{N}$ and $u \in \mathbb{N}$ is a looping context (page 41).

d	Policy expression graph, $d \in \mathcal{D} = (V, E, V_f, U)$ (page 45).
V_f	Set of final vertices of a policy, $V_f \subseteq V$, for some V (page 45).
U	Set of input uniqueness restrictions of a policy, $U \subseteq \textbf{In} \times V$, for some V (page 45).
$n \xrightarrow[u]{t} n'$	A constructor that returns an edge (n, n', t, u) (page 41).
$\precsim$, $\prec$	Operators for graph subsets (page 42).
$G(C)$	Function that builds the graph of program C (page 42).
$edges(g)$	The set of edges of graph g (page 42).
$nodes(g)$	The set of nodes of graph g (page 42).
$n \xrightarrow[u]{t}_g n'$	Predicate that holds if $(n, n', t, u) \in edges(g)$ (page 42).
$n \xrightarrow[u]{t} n'$	Same as above, but when g is implicit in the context. This overloads notation for edge constructor. Usage of each is always clear from the context (page 42).
$n \xrightarrow{t} n'$	Same as above, with u irrelevant (page 43).
$n \rightarrow n'$	Same as above, with $t =$ `plain` (page 43).
t_d	An edge type that is different from `control` (page 43).
$n \xrightarrow{w}{}^* n'$	There exists a path (excluding `control` edges) between nodes n and n', with w being the sequence of labels on this path (page 43).
$n \rightarrow^* n'$	Same as above, with w irrelevant (page 43).
$n \xrightarrow[u]{w}{}^* n'$	Same as $n \xrightarrow{w}{}^* n'$, with the whole path in the same looping context u (page 43).
$\xrightarrow{\phi}$	Either $\xrightarrow{\phi_1}$ or $\xrightarrow{\phi_2}$ (page 43).
$\xrightarrow{\tau}$	An edge whose type is either `plain` or ϕ (page 43).
$\not\rightarrow n$	Indegree of n is zero (page 43).
$type(n)$	Type of node n (page 43).
$uni(g)$	Set of all pairs of input uniqueness relations in graph g, $uni(g) \subseteq$ **In** $\times$ **Vertex** (page 43).
$exp_g(n)$	All possible expressions held by node n in graph g (page 44).

$cexp_g(n)$	All conditional expressions the held by node n in graph g can depend upon (page 44).
$P(\sigma, g)$	State σ and graph g correspond to each other (page 45).
$*^t$	A wildcard label for a policy node, where t is the matching type, omitted when equals to `var` (page 45).
$fnodes(d)$	Set of final vertices (nodes) of policy graph d (page 46).
$uni(d)$	Set of input uniqueness restrictions of policy graph d (page 46).
p	Denotes an information path of some graph, $p \precsim g$, for some $g \in \mathcal{G}$ (page 47).
$ip_g(n)$	The set of information paths that reach node n in graph g (page 47).
$mip_g(n)$	The set of maximal information paths that reach node n in graph g (page 47).
$n \simeq n'$	Nodes n and n' are similar, i.e. they have the same type and either the labels are the same or one of them is a wildcard $*$ (page 47).
$n \sim_{p,d} n'$	Node n, in the information path p, simulates node n', in the policy graph d. $\sim_{p,d}$ denotes the largest policy simulation relation between p and d (page 48).
$\overline{dds_p}(n, d)$	Node n is data dependency safe in information path p, with respect to policy d (page 49).
$\overline{dds}(n, g, d)$	Node n is data dependency safe in graph g, with respect to policy d (page 49).
$\overline{cds}(n, g, d)$	Node n is control dependency safe in graph g, with respect to policy d (page 49).
$\overline{safe}(n, g, d)$	Node n is safe in graph g, with respect to policy d (page 49).
$\overline{valid}(g, d)$	Graph g is valid with respect to policy d (page 49).
$\overline{public}(e, d)$	Expression e is public according to policy d (page 51).
Pol	A set of declassifiable expressions, $Pol \subseteq \mathbf{Exp}\langle \mathbf{In} \times \mathbb{N} \rangle$ (page 52).
$\langle Pol, Prog, valid \rangle$	PCR framework (page 52).
$\overline{Pol}$	A domain for declassification policies (page 53).
$\mathbb{P}$	A policy interpretation function $\mathbb{P} : \overline{Pol} \to Pol$ (page 53).
$\overline{Prog}$	A domain for abstractions of programs (page 53).

$\mathbb{C}$ A program abstraction function $\mathbb{C} : Prog \rightarrow \overline{Prog}$ (page 53).

$\mathbb{V}$ A validation function $\mathbb{V} : \overline{Prog} \times \overline{Pol} \rightarrow Bool$, which, for a program C and a policy d, satisfies $\mathbb{V}(\mathbb{C}(C), d) \Rightarrow valid(C, \mathbb{P}(d))$ (page 53).

$\langle(\overline{Pol}, \mathbb{P}), (\overline{Prog}, \mathbb{C}), \mathbb{V}\rangle$ An implementation of the PCR framework $\langle Pol, Prog, valid\rangle$ (page 53).

$int(d)$ The policy interpretation function for policy expression graphs, $int : \mathcal{D} \rightarrow \wp(\mathbf{Exp}\langle\mathbf{In} \times \mathbb{N}\rangle)$ (page 53).

$\overline{states}(G(C))$ The set of all program states in C that $G(C)$ can represent (page 53).

$\mathcal{C}(\cdot)$ The worst-case time complexity of computation $\cdot$, which can be a function or operator (page 56).

Hybrid Static-Runtime Enforcer

θ^i The I/O operation on channel θ, performed at program point i (page 74).

$I \rightsquigarrow \gamma^i$ A flow that denotes that each element in the set of input operations and declassification matchings in I potentially flows to output operation γ^i (page 74).

$I \mapsto^X l$ The representation of a declassification matching, denoting that the input operations in set I flow to a variable which matches a declassification policy, changing its label to l. X represents the set of constraints associated with the matched declassification (page 75).

$label(n_f)$ Security label associated to final policy node n_f (page 77).

$id(C)$ The program point of statement C (page 77).

$flow_{g,d}(n_x)$ The set of input operations and declassifications (according to policy d) that potentially flow to variable node $n_x \in nodes(g)$ (page 77).

$constr(\mathcal{R})$ The set of constraints associated to the declassification matching that corresponds to the node simulation relation $\mathcal{R}$ (page 78).

$fr_d(C)$ The flow report of program C, according to policy d (page 78).

$fr_{g,d}(n_\gamma)$ The flow report relative to output node $n_\gamma \in g$, according to policy d (page 78).

f A flow (page 81).

dc	A declassification within a flow (page 81).
l	A security label (page 81).
pf	A partial flow, i.e. the left-hand side of a flow (page 81).
$from(f)$	The left-hand side of a flow (also used for a declassification dc) (page 81).
$to(f)$	The right-hand side of a flow (also used for a declassification dc) (page 81).
$constr(dc)$	The constraint set related to declassification matching dc (page 81).
$label(\alpha^i)$	The security label of I/O operation α^i (page 81).
$id(l)$	The program point of the input operation that was translated to label l (page 81).
$lbl(f)$	The translation of elements of flow f to their respective security labels. Also used for a partial flow pf and a declassification dc (page 82).
$check(X)$	The static validation of declassification constraint set X. Also used for a single element $x \in X$ (page 82).
$validate(C, d)$	The static validation of program C with policy d (page 82).
$validate(f)$	The static validation of flow f (page 82).
$checklist(C, d)$	The runtime checklist of program C with policy d (page 82).
$checklist(f)$	The runtime checklist relative to flow f (page 82).

List of Figures

1.1 Type-based approach for declassification 6
1.2 Roadmap of the core sections of chapters 2 and 3 17

2.1 Program semantics 26

3.1 Expression graph for variable v_3 of authentication program 36
3.2 Policy graph for example of authentication program 37
3.3 Expression graph for variable a_4 39
3.4 Expression graph for variable c_3 39
3.5 Declassification policy graph for average example 40
3.6 Encryption program and its matching policy 41
3.7 Graph building function. Note that $fresh()$ returns a previously unused value and that ϵ is used to denote an unspecified index. 42
3.8 Example of information path calculation 48
3.9 Examples of graph matching 50
3.10 Algorithm 3.3 executing on a simple graph with a cycle 62
3.11 Algorithm 3.3 executing on a graph with nested cycles 63

4.1 Overview of the 3-step enforcement 72
4.2 A Java implementation for Example 4.1 85
4.3 Dalvik bytecode snippet for code of Figure 4.2 86
4.4 Processing times of experiments 88

5.1 Program expression graph for Example 5.1 92
5.2 Program expression graph for variables c_1 and c_2, in Example 5.2 93
5.3 Example of source to assembly compilation 100

List of Algorithms

3.1 Input uniqueness calculation 55
3.2 Program expression graph validation 58
3.3 Calculation of information paths 61

4.1 Classification application 70
4.2 Declassification application 71
4.3 Iterative declassification application 71
4.4 Pre-load checker 84

Summary

Information Flow and Declassification Analysis for Legacy and Untrusted Programs

Standard access control mechanisms are often insufficient to enforce compliance of programs with security policies. For this reason, information flow analysis has become a topic of increasing interest. In such type of analysis, the main property to be checked is called non-interference, which basically states that the publicly observable behaviour of a program is entirely independent of its secret, secure input values.

However, simple non-interference is too restrictive for specifying and enforcing information flow policies in most programs. Exceptions to non-interference are provided using declassification policies. Several approaches for enforcing declassification have been proposed in the literature. In most of these approaches, the declassification policies are embedded in the program itself or heavily tied to the variables in the program being analyzed, thereby providing at best little separation between the code and the policy. Consequently, the previous approaches essentially require that the code be trusted, since to trust that the correct policy is being enforced, we need to trust the source code.

In this thesis, we propose a novel framework for information flow analysis, with support to declassification policies, related to the source code being analyzed via its I/O channels. The framework supports many of the of declassification policies identified in the literature. Based on flow-based static analysis, it represents a first step towards a new approach that can be applied to untrusted and legacy source code to automatically verify that the analyzed program complies with the specified declassification policies. We present a framework in which expressions over input channel values that could be output by the program are compared to a set of declassification requirements. We build an implementation of such framework, which works by constructing a conservative approximation of the such expressions, and by determining whether all of them satisfy the declassification requirements stated in the policy. We introduce a representation of such expressions that resembles tree automata. We prove that if a program is considered safe according to our analysis then it satisfies a property we call Policy Controlled Release, which formalizes information-flow correctness according to our notion of declassification policy. We demonstrate, through examples, that our approach works for several interesting and useful declassification policies, including one involving declassification of the average of several confidential values. Finally, we extend the static analyzer to build a practical

hybrid static-runtime enforcement mechanism, consisting of 3 steps: static analysis, pre-load checking, and runtime enforcement. We demonstrate how the hybrid mechanism is able to enforce real-world policies which are unable to be treated by standard approaches from industry. Also, we show how this goal is achieved by keeping the static analysis step system independent, and the runtime enforcement with minimum runtime overhead.

Curriculum Vitae

Bruno Pontes Soares Rocha was born on June 26, 1983 in Belo Horizonte, Brazil. From 2002 until 2007 he pursued both his Bachelor's and Master's degrees in Computer Science, at the Federal University of Minas Gerais (UFMG), located in Belo Horizonte, Brazil. During these studies, he focused his work and early research contributions on mobile computing, distributed systems, complex networks and network security applications. His undergraduate research was chosen amongst the top 10 undergraduate computer science research projects in Brazil for the year of 2006. Also, in 2005 he was ranked higher than 99.8% of all applicants to the Brazilian National Postgraduate Exam in Computer Science (POSCOMP). During this period he also worked as a software engineer for nearly four years, and held a start-up company for the entire year of 2006.

In February 2008 Bruno started as a Ph.D. candidate on the newly created Security group (SEC) at the Technische Universiteit Eindhoven (TU/e), under the guidance of Prof. Dr. Sandro Etalle and Dr. Jerry den Hartog. He started to work on fundamental aspects of security, and soon decided to pursue research in information flow and declassification analysis. For this, he started to work jointly with Prof. Dr. William H. Winsborough, from the University of Texas at San Antonio, U.S. Towards the end of his Ph.D., he was an intern at Google Inc. at Mountain View, California, from July until October 2011. There, he worked in the Security team of Chrome OS.

The present dissertation contains the results of his work from 2008 to 2012.