Enforcing Information-Flow Policies by Combining Static and Dynamic Analyses

Thèse

Andrew Bedford

Doctorat en informatique
Philosophiæ doctor (Ph. D.)

Québec, Canada

© Andrew Bedford, 2019

Enforcing Information-Flow Policies by Combining Static and Dynamic Analyses

Thèse

Andrew Bedford

Sous la direction de:

Josée Desharnais, directrice de recherche
Nadia Tawbi, codirectrice de recherche

Résumé

Le contrôle de flot d'information est une approche prometteuse permettant aux utilisateurs de contrôler comment une application utilise leurs informations confidentielles. Il reste toutefois plusieurs défis à relever avant que cette approche ne puisse être utilisée par le grand public. Plus spécifiquement, il faut que ce soit efficace, facile à utiliser, que ça introduise peu de surcharge à l'exécution, et que ça fonctionne sur des applications et langages réels. Les contributions présentées dans cette thèse nous rapprochent de ces buts.

Nous montrons qu'une combinaison d'analyse statique et dynamique permet d'augmenter l'efficacité d'un mécanisme de contrôle de flot d'information tout en minimisant la surcharge introduite. Notre méthode consiste en trois étapes : (1) à l'aide d'analyse statique, vérifier que le programme ne contient pas de fuites d'information évidentes; (2) instrumenter l'application (c.-à-d., insérer des commandes) pour prévenir les fuites d'information confidentielles à l'exécution; (3) évaluer partiellement le programme pour diminuer l'impact de l'instrumentation sur le temps d'exécution.

Pour aider les utilisateurs à identifier les applications qui sont les plus susceptibles de faire fuir de l'information confidentielle (c.à.d., les applications malicieuses), nous avons développé un outil de détection de maliciel pour Android. Il a une précision de 94% et prend moins d'une seconde pour effectuer son analyse.

Pour permettre aux utilisateurs de prioriser l'utilisation de ressources pour protéger l'information provenant de certaines sources, nous introduisons le concept de *fading labels.*

Pour permettre aux chercheurs de développer plus facilement et rapidement des mécanismes de contrôle de flot d'informations fiables, nous avons développé un outil permettant de générer automatiquement la spécification d'un mécanisme à partir de la spécification d'un langage de programmation.

Pour permettre aux chercheurs de plus facilement communiquer leurs preuves écrites en Coq, nous avons développé un outil permettant de générer des versions en langue naturelle de preuves Coq.

Abstract

Information-flow control is a promising approach that enables users to track and control how applications use their sensitive information. However, there are still many challenges to be addressed before it can be used by mainstream users. Namely, it needs to be effective, easy to use, lightweight, and support real applications and languages. The contributions presented in this thesis aim to bring us closer to these goals.

We show that a combination of static and dynamic analysis can increase the overall effectiveness of information-flow control without introducing too much overhead. Our method consists of three steps: (1) using static analysis, we verify that the program does not contain any obvious information leaks; (2) we instrument the program to prevent less obvious leaks from occurring at runtime; (3) we partially evaluate the program to minimize the instrumentation's impact on execution time.

We present a static-based malware detection tool for Android that allows users to easily identify the applications that are most likely to leak sensitive information (i.e., malicious applications). It boasts an accuracy of 94% and takes less than a second to perform its analysis.

We introduce the concept of fading-labels, which allows information-flow control mechanisms to prioritize the usage of resources to track information from certain sources.

We present a tool that can, given a programming language's specification, generate information-flow control mechanism specifications. This should allow researchers to more easily develop information-flow control mechanisms.

Finally, we present a tool that can generate natural-language versions of Coq proofs so that researchers may more easily communicate their Coq proofs.

Contents

Résumé iii

Abstract iv

Contents v

List of Figures viii

List of Listings ix

Remerciements xi

Foreword xii

Introduction 1

1 Basics of Information-Flow Control **6**
- 1.1 Noninterference . . . 6
- 1.2 Enforcement mechanisms . . . 11
- 1.3 Precision . . . 16
- 1.4 Conclusion . . . 18
- 1.5 Bibliography . . . 18

2 Enforcing Information Flow by Combining Static and Dynamic Analysis **21**
- 2.1 Résumé . . . 21
- 2.2 Abstract . . . 22
- 2.3 Introduction . . . 22
- 2.4 Programming Language . . . 24
- 2.5 Security Type System . . . 26
- 2.6 Inference Algorithm . . . 31
- 2.7 Instrumentation . . . 33
- 2.8 Related Work . . . 36
- 2.9 Conclusion . . . 38
- 2.10 Bibliography . . . 38

3 A Progress-Sensitive Flow-Sensitive Inlined Information-Flow Control Monitor **41**
- 3.1 Résumé . . . 41
- 3.2 Abstract . . . 42

3.3 Introduction . . . 42
3.4 Source Language . . . 47
3.5 Security . . . 49
3.6 Type-Based Instrumentation . . . 50
3.7 Soundness . . . 62
3.8 Increasing Precision and Permissiveness . . . 64
3.9 Related Work . . . 66
3.10 Conclusion . . . 67
3.11 Bibliography . . . 68
3.12 Appendix: Examples . . . 71
3.13 Appendix: Proofs . . . 73

4 Andrana: Quick and Accurate Malware Detection for Android **84**
4.1 Résumé . . . 84
4.2 Abstract . . . 84
4.3 Introduction . . . 85
4.4 Background on Android . . . 86
4.5 Overview of Andrana . . . 88
4.6 Feature Extraction . . . 89
4.7 Classification and Evaluation . . . 92
4.8 Additional Experiments . . . 96
4.9 Related Work . . . 97
4.10 Conclusion . . . 99
4.11 Bibliography . . . 99

5 Information-Flow Control with Fading Labels **103**
5.1 Résumé . . . 103
5.2 Abstract . . . 104
5.3 Introduction . . . 104
5.4 Fading Labels . . . 105
5.5 Depth-Limited Noninterference . . . 106
5.6 Example of a Mechanism . . . 107
5.7 Discussion . . . 109
5.8 Bibliography . . . 111

6 Towards Automatically Generating Information-Flow Mechanisms **113**
6.1 Résumé . . . 113
6.2 Abstract . . . 114
6.3 Introduction . . . 114
6.4 Overview of Ott-IFC . . . 115
6.5 Discussion . . . 124
6.6 Conclusion and Future Work . . . 125
6.7 Bibliography . . . 127

7 Coqatoo: Generating Natural Language Versions of Coq Proofs **129**
7.1 Résumé . . . 129
7.2 Abstract . . . 129
7.3 Introduction . . . 130

7.4 Overview of Coqatoo 130
7.5 Comparison 133
7.6 Future Work 134
7.7 Bibliography 135

Conclusion **136**

List of Figures

1.1 A Java password checker and its PDG [12] 13
1.2 Example of operational semantics rules 14

2.1 Analysis of a program where an implicit flow may lead to a leak of information 23
2.2 A few rules of the structural operational semantics 25
2.3 Typing rules 28
2.4 Inference Algorithm 32
2.5 Instrumentation algorithm. 33
2.6 Implicit flow 35
2.7 The send of a high value on an unknown channel calls for instrumentation . . . 36
2.8 The **send** of a high value on an unknown channel calls for instrumentation . . 37
2.9 A more realistic example 37

3.1 Semantics of the source language 49
3.2 Instrumentation and typing rules for the source language 55

4.1 General flow of Andrana 88
4.2 Analysis time and memory usage distributions. 89
4.3 The progression of true positives ratio and accuracy on the test set for different ratios of training set and for each learning algorithm. It is calculated using the best configuration of hyperparameters outputted by a 5-fold cross-validation. . 95
4.4 Andrana's interface on Android. 97

5.1 PDG-like representation of Listing 5.1 106
5.2 Semantics of the read, assign and write commands when using fading labels 109

6.1 Evaluation-order graphs of the assign, sequence and conditional commands. . . 120

7.1 Proof tree of Listing 7.1 132
7.2 Example of a proof generated by the approach of Coscoy et al. 134

List of Listings

1 Insecure explicit flow 2
2 Insecure implicit flow 2
1.1 Attempts to explicitly leak information 7
1.2 Attempt to implicitly leak information 7
1.3 Attempt to implicitly leak a larger amount of information 8
1.4 Attempt to implicitly leak information is detected 8
1.5 Leaking information through an exception 8
1.6 Leaking a large amount of information through an exception 9
1.7 A program's progress can also leak information 9
1.8 Progress channels can leak a significant amount of information 10
1.9 Loop that always terminates 10
1.10 Leaking information through timing-channels 11
1.11 Preventing leaks through the timing channel 11
1.12 Possible instrumentation of a send command 14
1.13 Producing the public outputs 15
1.14 Producing the private outputs 15
1.15 A program with multi-faceted variables 15
1.16 Flow sensitive vs. flow insensitive 16
1.17 Example for context-sensitivity 17
1.18 Example for object-sensitivity 17
1.19 Example for path-sensitivity 17
3.1 Statically uncertain channel level 44
3.2 Statically uncertain variable level 45
3.3 Progress leak 45
3.4 Loop that always terminates 46
3.5 Leaking through monitor decision 46
3.6 Variable level sensitivity 46
3.7 We cannot be pessimistic about channel variables 52
3.8 Dangerous runtime halting 57
3.9 A guarded send can generate a progress leak 59
3.10 Modified variables 60
3.11 The security type of c's content is sensible 63
3.12 The security type of x's content is sensible 63
3.13 Constraint on the security type of a channel variable 64
3.14 Constraint on the security type of an integer variable 65
3.15 Example where _hc does not need to be updated with x_{ctx} 65
4.1 Instantiating an object and calling a function using reflection 91
4.2 Instanciating an object of a statically unknown class using reflection 91

4.3 Dynamically built class name 96
5.1 Derived values 105
6.1 Insecure explicit flow 114
6.2 Insecure implicit flow 115
6.3 Syntax of a simple imperative language 116
6.4 Small-step semantics of boolean expressions 116
6.5 Small-step semantics of arithmetic expressions 117
6.6 Small-step semantics of commands 118
6.7 Ott-IFC's output for the "read" command 121
6.8 Ott-IFC's output for the "write" command 122
6.9 Ott-IFC's output for the "assign" command 122
6.10 Ott-IFC's output the big-step version of the "assign" command 122
6.11 Ott-IFC's output for the "sequence" command 123
6.12 Ott-IFC's output for the "if" command 123
6.13 Ott-IFC's output for the "while" command 124
6.14 Program accepted by flow-sensitive analyses 125
7.1 Proof script given as input 131
7.2 State before executing the first intros tactic 131
7.3 State after executing the first intros tactic 131
7.4 Coqatoo's output in annotation mode 133
7.5 Improved precision using abstract interpretation 137

Remerciements

J'aimerais tout d'abord remercier mes directrices de recherche, Josée Desharnais et Nadia Tawbi, pour m'avoir donné la chance de travailler à leurs côtés. Je n'aurais pas pu demander de meilleures directrices! Vous êtes brillantes, passionnées et vous n'hésitez pas à aider les gens autour de vous! Je tiens particulièrement à vous remercier pour m'avoir toujours encouragé à explorer mes idées, même lorsque celles-ci n'étaient pas directement liées au sujet de recherche. C'est sans hésitation et de tout mon coeur que je vous recommanderais à un autre étudiant.

J'aimerais aussi remercier Jean-Philippe Lachance, David Landry, Sébastien Garvin, Souad El-Hatib et Loïc Ricaud pour nous avoir aidés à implémenter les divers outils et pour avoir rendu le laboratoire plus vivant.

J'aimerais aussi remercier mes amis, sans qui, je n'aurais peut-être jamais commencé à programmer et ma carrière aurait été très différente.

Finalement, j'aimerais remercier toute ma famille pour leur appui, particulièrement mes parents. C'est grâce à vous si je suis où je suis et qui je suis aujourd'hui.

Foreword

We present in this thesis six different papers which have all been peer-reviewed and presented in various conferences. Here are some information regarding the authors and their roles for each paper.

Enforcing Information Flow by Combining Static and Dynamic Analysis

This paper was written by Andrew Bedford, Josée Desharnais, Théophane G. Godonou and Nadia Tawbi. It was published in the proceedings of the *International Symposium on Foundations & Practice of Security* in 2013. Théophane G. Godonou, a graduate student supervised by Nadia Tawbi and Josée Desharnais, designed the type-based instrumentation algorithm. Andrew Bedford, an intern at the time, implemented the algorithm and helped improve its final form. All of the authors contributed ideas and participated in the paper's writing process.

A Progress-Sensitive Flow-Sensitive Inlined Information-Flow Control Monitor

This paper was written by Andrew Bedford, Stephen Chong, Josée Desharnais, Elisavet Kozyri and Nadia Tawbi. It was published in the proceedings of the *IFIP International Information Security and Privacy Conference* in 2016. Andrew Bedford, Josée Desharnais and Nadia Tawbi are the main authors of this paper. Together, they designed the type systems, instrumentation algorithms and proved their correctness. Stephen Chong, a professor and collaborator from Harvard, provided feedback throughout the project.

After winning the conference's *Best Student Paper Award*, the paper was published in a special issue of the journal *Computers & Security*. The journal version of the paper, which is the version presented Chapter 3, includes new results and contributions. Elisavet Kozyri, a graduate student and collaborator from Cornell, helped write the final journal version.

Andrana: Quick and Accurate Malware Detection for Android

This paper was written by Andrew Bedford, Sébastien Garvin, Josée Desharnais, Nadia Tawbi, Hana Ajakan, Frédéric Audet and Bernard Lebel. It was published in the proceedings of the *International Symposium on Foundations & Practice of Security* in 2016. Andrew Bedford

is the main author of this paper. Sébastien Garvin, an intern under the supervision of Andrew Bedford, Josée Desharnais and Nadia Tawbi, helped implement Andrana's feature extraction functionality. Hana Ajakan, a graduate student under the supervision of professor François Laviolette, trained and tested Andrana's classifiers using different machine-learning algorithms. Frédéric Audet and Bernard Lebel, industrial collaborators from Thales Research & Technology Canada, provided the datasets of Android applications and helped determine the appropriate set of features to extract.

Information-Flow Control with Fading Labels

This paper was written by Andrew Bedford. It was published in the proceedings of the *International Conference on Privacy, Security and Trust* in 2017 as an extended abstract. Professors Josée Desharnais and Nadia Tawbi provided feedback on the ideas presented in the paper. The version presented in Chapter 5 is an extended version which includes an example of a mechanism using fading labels.

Towards Automatically Generating Information-Flow Control Mechanisms

This paper was written by Andrew Bedford. Though it was not published, it was peer-reviewed and presented at the *ACM SIGPLAN Symposium on Principles of Programming Languages Student Research Competition* in 2018 under the title *Generating Information-Flow Control Mechanisms from Programming Language Specifications*. The version presented in Chapter 6 is an improved version that takes into account some of the feedback received during the competition.

Coqatoo: Generating Natural Language Versions of Coq Proofs

This paper was written by Andrew Bedford. It was presented at the *International Workshop on Coq for Programming Languages*, a POPL co-hosted workshop, in 2018. To foster open discussion of cutting edge research, the papers from this conference are not published; they are instead made readily available on the workshop's website. The version presented in Chapter 7 is an improved version that takes into account some of the feedback received during and after the workshop.

Introduction

Modern operating systems rely mostly on role-based access-control mechanisms to protect users information. However, role-based access control mechanisms are insufficient as they cannot regulate the propagation of information once it has been released for processing. For example, according to the permission table below, program β should not have access to `private.txt`. Yet nothing prevents program α from copying `private.txt`'s information into `public.txt`, which program β has access to, thereby giving program β access to `private.txt`'s information.

	Program α	Program β
`private.txt`	read write	`- -`
`public.txt`	read write	read write

The problem is that such a leak is hidden in the code of program α. To address this issue, a research trend called *language-based information-flow security* has emerged. As the name suggests, the idea is to use language-based techniques, such as program monitoring, rewriting and type checking, to analyze a program's code and to prevent information from flowing to an undesired destination (e.g., information from a private file should not be saved in a public file). Mechanisms that enforce such policies are called *information-flow control mechanisms*. The policy that is usually enforced by these mechanisms, called *noninterference*, states that private information may not interfere with the publicly observable behavior of a program. In other words, someone observing the public outputs of a program should not be able to deduce anything about its private inputs, even if the observer has access to the program's source code. To enforce noninterference, a mechanism must take into account two basic types (among others) of information flows: *explicit flows* and *implicit flows*.

An insecure explicit information flow occurs when private information directly influences public information. For example in Listing 1, the value that is written to `publicFile` depends on the value of `x`, which in turn depends on the value of `privateValue`. Hence, any output of `x` would reveal something about `privateValue`.

```
x := privateValue + 42;
write x to publicFile
```

Listing 1: Insecure explicit flow

An insecure implicit information flow occurs when private information influences public information through the control-flow of the program. For example in Listing 2, the value that is written to `publicFile` depends on the condition `privateValue > 0`. So someone observing the content of `publicFile` could learn whether or not `privateValue` is greater than zero.

```
if (privateValue > 0) then
  write 0 to publicFile
else
  write 1 to publicFile
end
```

Listing 2: Insecure implicit flow

Insecure explicit and implicit flows can be identified and prevented using static, dynamic or hybrid mechanisms. Static information-flow control mechanisms analyze a program before its execution to determine if it satisfies the appropriate information flow requirements. They introduce no runtime overhead, but accept or reject a program in its entirety. Dynamic information-flow control mechanisms accept or reject individual executions at runtime, without performing any preliminary static program analysis, but can introduce significant runtime overheads. Hybrid information-flow control mechanisms combine static and dynamic program analysis. They strive to achieve the benefits of both static and dynamic mechanisms: precise (i.e., per-execution) enforcement of security, with low runtime overhead.

While promising, there are still many challenges to be addressed before information-flow control mechanisms become effective, easy to use and lightweight. Namely, we need flexible and fully automatic mechanisms that introduce as little overhead as possible during execution, while providing as much protection as possible. They must be capable of handling real-world applications and real-world language features.

To that end, we present in this thesis six papers which introduce new ways of identifying precisely when dynamic information-flow control should be necessary using static analysis (Chapters 2, 3 and 4), new ways of parameterizing the precision of an analysis (Chapter 5) and new tools to help researchers develop sound mechanisms (Chapters 6 and 7). More specifically, the contributions of each paper as follows.

Contributions

Enforcing Information Flow by Combining Static and Dynamic Analysis (Chapter 2)

In this chapter, we present a type system that statically checks noninterference for a simple imperative programming language. To the usual two main security levels, public and private, we add a new value, *unknown*. This new value captures the possibility that we may not know, before execution, whether some information is public or private. Standard two-valued analyses have no choice but to be pessimistic when faced with uncertainty and hence generate false positives. If uncertainty arises during the analysis, we tag the instruction in cause. In a second step, instrumentation at every such point together with dynamic analysis will allow us to obtain a more precise result than purely static approaches. Using this approach, we can reduce the number of false positives, while introducing a light runtime overhead by instrumenting only when there is a need for it.

A Progress-Sensitive Flow-Sensitive Inlined Information-Flow Control Monitor (Chapter 3)

Most programs today can interact with an external environment during their execution. This means that a program's outputs can be used to track its progress (i.e., where it is in its execution), and so, can be used to leak information without being detected by traditional information-flow mechanisms.

In this chapter, we adapt the approach presented in Chapter 2 to take into account leaks through progress and extended our approach to support more than simply two levels of information (private, public). One particularity of this extended version is that we use sets of levels during static analysis to represent the possible levels that can be associated with a variable during execution. When general lattices are enforced, we found sets of levels to be a more accurate representation than our previously used *unknown* level. Thus, the distinction that our static analysis makes between outputs that *never* leak information and outputs that *may* leak information is more precise when using sets of levels. In this chapter, we also prove that our monitor is sound and that the semantics of the original program is preserved, as long as the program is secure.

Andrana: Quick and Accurate Malware Detection for Android (Chapter 4)

Wanting to apply our approach to real-world applications, we turned our attention to Android. We quickly realized that enforcing information-flow policies on real-world applications can result in significant overheads and false positives, even when using hybrid mechanisms. So, in order to identify applications that are most likely to leak a user's sensitive information (i.e., malicious applications), and hence in most need of monitoring, we have developed a

malware detection tool for Android called *Andrana*. Instead of relying on the more frequently used dynamic analysis, it uses static analysis to detect an application's features, and machine learning algorithms to determine if these features are sufficient to classify an application as a malware.

While Andrana is not the first tool of this kind, it equals others in terms of accuracy (94%) and surpasses most in terms of speed, taking less than a second to perform its analysis and classification. Compared to antiviruses, which mostly use pattern matching algorithms to identify known malware (i.e., they look for specific sequences of instructions), Andrana's main advantage is that it can not only detect known malware and their variations, but also unknown malware.

Information-Flow Control with Fading Labels (Chapter 5)

While working with mobile devices for Andrana, we sought other ways to reduce the overhead that is introduced by information-flow control mechanisms. We realized that existing mechanisms generally invest the same amount of resources to protect information of varying importance. However, in systems where resources are limited such as smartphones and tablets, it may be more appropriate to spend more resources to protect information that is more important (e.g., passwords), and less resources to protect information that is less important (e.g., current location). To address this issue, we introduce in this chapter the concept of *fading labels*.

Fading labels are labels whose taint stops propagating after a fixed amount of uses (i.e., they fade away). By parameterizing mechanisms so that labels associated to important information fade away more slowly than those associated to less important information, they allow mechanisms to use more resources to track important information than other information. Since leaks of information may occur once a label fades away, mechanisms that use them may not always enforce noninterference. What they do enforce is a relaxed version of noninterference, called *depth-limited noninterference*, which we also introduce in this chapter.

Towards Automatically Generating Information-Flow Control Mechanisms (Chapter 6)

From our own experience, we know that implementing an information-flow control mechanism for Android can be a difficult and time-consuming task due to the numerous and subtle ways through which information may flow in a program. Part of the problem comes from the fact that, while techniques to prevent insecure explicit and implicit flows are well known and widely used, they still need to be applied manually when designing new mechanisms. This can lead to errors, failed proof attempts and time wasted.

In order to make this task less laborious, we present in this chapter a tool called *Ott-IFC* that

can, given a programming language's specification, automatically apply those techniques and generate information-flow control mechanism specifications. More specifically, it analyzes the syntax and semantics of an imperative programming language on which we want to enforce noninterference, and uses rewriting rules to generate a hybrid runtime monitor's semantics. To the best of our knowledge, this is the first tool of this kind.

Coqatoo: Generating Natural Language Versions of Coq Proofs (Chapter 7)

While working on Ott-IFC, we sought other ways to facilitate the development of information-flow control mechanisms. One important aspect of any such mechanism is proving that they are sound. Since Ott-IFC uses Ott as its input and output language, specifications can be exported to proof assistants, such as Coq and Isabelle, so that the user may complete their implementation and proofs. However, a disadvantage of using proof assistants is that the resulting proofs can be sometimes hard to read and understand, particularly for less-experienced proof assistant users.

To address this issue, we have developed a tool called *Coqatoo* that can generate natural language versions of Coq proofs using a combination of static and dynamic analysis. It generates natural language proofs from high-level proof scripts instead of the low-level proof-terms used by previous work. By adopting this approach, it can avoid the inherent verbosity that comes from using low-level proof-terms and avoid losing valuable information such as the tactics that are used, the user's comments and the variable names. Furthermore, contrarily to previous approaches, it gives the user direct control over the verbosity of the generated natural language version: the more detailed a proof script is, the more detailed its natural language version will be.

Chapter 1

Basics of Information-Flow Control

To help understand the content of this thesis, we present in this chapter the basics of information-flow control. We first present the security policy that is enforced by most information-flow control mechanisms and how information may flow in a program. We then present different ways by which this policy can be enforced. An initiated reader can skip this chapter. For the sake of completeness, this chapter includes notions that are not necessary to understand the content of this thesis; they are denoted with a red star (*).

1.1 Noninterference

Noninterference [1] is the security policy that is enforced by most information-flow mechanisms. Intuitively, it states that private information should not interfere with the publicly observable behavior of a program. In other words, someone observing the public outputs of a program should not be able to deduce anything about its private inputs, even if the observer has access to the program's source code.

More formally, it states that for any two executions of a program whose initial memories differ only on the private inputs, if one execution can produce some publicly observable output, then the other execution also produces the same publicly observable output. This means that an observer of the public output could not distinguish the two executions, and thus would learn nothing about the private inputs. A mechanism that correctly enforces this policy is said to be *sound*. Sound mechanisms may have false positives (i.e., reporting that there is a leak of information when there is not), but must not have any false negatives (i.e., reporting that there is no leak of information when there is).

This policy follows Bell-Lapadula's model of *no read up* (i.e., a public user may not read private information), *no write down* (i.e., private information may not be written down to public channels) [13]. Information-flow mechanisms strive to implement this model by tracking out its violations, not only at the command level (i.e., by considering how information flows

from one command to another), but also at the covert channel level (i.e., considering how information could be leaked through different covert channels such as a program's progress, a phone's vibration or the execution time of an application).

1.1.1 Information flows

To present the different ways by which information may flow, let us consider the following scenario: a program has access to private information (e.g., credit card numbers) and wants to leak it to a public channel. For the sake of simplicity, let us assume that this information is a natural number. Note that we use a **send to** command in our examples to express the fact that an output is produced, but some works consider assignments as outputs.

Explicit flows An insecure explicit information flow occurs when private information influences public information through a data dependency. For example in Listing 1.1, the value that is sent to `publicChannel` depends on the value of `x`, which in turn depends on the value of `privateInfo`. Hence, any output of `x` would reveal something about `privateInfo`.

```
x := privateInfo + publicInfo;
send x to publicChannel
```

Listing 1.1: Attempts to explicitly leak information

To track these dependencies, the usual approach is to associate sensitive information with a label, which is then propagated whenever the information is used, a process called *tainting*. In other words, each program variable has a label which represents the level of information that it contains. In this example, `x`'s value depends on both private and public information. To avoid leaking information, the label associated to `x` must be at least as restrictive as the two. For this reason, the variable `x` is labeled as "private", meaning that the information contained within the variable is considered to be private. Before executing the **send** command, enforcement mechanisms use these labels to verify that no leak is about to occur. Since in this case the program is trying to send `x`, which is considered private, to `publicChannel`, which is considered public, the program's execution is stopped by the mechanism to prevent a leak from occurring.

Implicit flows An insecure *implicit flow* [2] occurs when a conditional (e.g., the guard condition of an **if** or **while** command) depends on private information and alters the program's publicly observable behavior. For example, in Listing 1.2, the value received by the public channel (`0` or `1`) reveals whether or not `privateInfo` is greater than `100`.

```
if privateInfo > 100
    then x := 0
    else x := 1 end;
send x to publicChannel (*leak*)
```

Listing 1.2: Attempt to implicitly leak information

Although only one bit of information is leaked in this example, implicit flows can be used to leak an arbitrary amount of information [3]. For instance, consider the program of Listing 1.3 which iterates through the possible values of `privateInfo`. When the exact value is found, it is sent to the public channel.

```
leakedNumber := false;
i := 0;
while (leakedNumber = false)
 if privateInfo = i then (*exact value found*)
  send i to publicChannel; (*leak*)
  leakedNumber := true
 end;
 i := i + 1
end
```

Listing 1.3: Attempt to implicitly leak a larger amount of information

One way to detect and prevent implicit flows is to keep track of the context in which commands are executed using a program counter variable, usually named *pc*. Listing 1.4 illustrates how the use of such a variable can prevent the leak that occurs in Listing 1.2, and its usage is explained in comments.

```
(*pc is initially public*)
if privateInfo > 100 then (*pc is now private because of the condition*)
 x := 0 (*because x is assigned in a private context, x will be considered private*)
else
 x := 1 (*same thing here*)
end;
(*pc goes back to public*)
send x to publicChannel (*leak detected because x is private and publicChannel is public*)
```

Listing 1.4: Attempt to implicitly leak information is detected

Exception flows* Exceptions can be raised (and caught) by the programmer to handle exceptional cases or by the runtime environment when an error occurs (e.g., a division by zero, stack overflow, out of bounds). Like conditionals, they can alter the control flow of a program and create implicit flows [4]. For example in Listing 1.5, whether or not the execution of the division throws an exception reveals information on `privateInfo`'s value.

```
try {
 x := 1/privateInfo;
 send "privateInfo ≠ 0" to publicChannel
} catch (Exception e) { (*Division by zero*)
 send "privateInfo = 0" to publicChannel }
```

Listing 1.5: Leaking information through an exception

Like implicit flows, exceptions can be used to leak more than one bit at a time [3]. Listing 1.6 shows that it is possible to leak the exact value of `privateInfo` using exceptions.

```
try {
 i := 0;
 while (i < MAX_NAT) do
  x := 1/(privateInfo-i) (*will divide by zero when i=privateInfo*)
 end
}
catch (Exception e) {
    (*Division by zero*)
 send "privateInfo = " + i to publicChannel
}
```

Listing 1.6: Leaking a large amount of information through an exception

In this example, an exception is thrown when `i` equals `privateInfo` and, consequently, the subsequent **send** leaks `privateInfo`'s exact value. A simple but restrictive way to prevent such leaks is to forbid operations that can throw exceptions on variables that are not public [4]. This means that the denominator of a division operation must be public, the index of an array must be public, etc.

Progress flows Most programs are interactive, meaning that they can interact with an external environment during their execution. This means that a program's outputs can be used to track its progress (i.e., where it is in its execution), and so, can be used to leak information without being detected by traditional information-flow mechanisms.

For example, if a user executes the program of Listing 1.7 and observes the string `"Starting program..."` on the public channel, then the user knows that the program has executed its first command. If the same user then observes the string `"Ending program..."`, then the user learns that `privateInfo` is less than or equal to `42`. On the other hand, if the user does not receive that last output, then the user learns that `privateInfo` is greater than `42`. In each case, information about the value of `privateInfo` is leaked.

```
send "Starting program..." to publicChannel;
while privateInfo > 42 do
 skip (*Infinite loop*)
end;
send "Ending program..." to publicChannel
```

Listing 1.7: A program's progress can also leak information

Like implicit flows, progress flows can be used to leak large amounts of information [3]. The program of Listing 1.8 is an example of such a leak.

```
i := 0;
while (i < MAX_NAT) do
 send i to publicChannel;
 if (i = privateInfo) then
  while true do skip end (*leak*)
 end;
 i := i + 1
end
```

Listing 1.8: Progress channels can leak a significant amount of information

This program is similar to the one in Listing 1.3. It also iterates through the possible values of `privateInfo`. The difference is that it sends all these possible values to the public channel so that the program's progress can be observed. When the condition `i = privateInfo` is true, the program becomes stuck in an infinite loop. Knowing this, a user can conclude that the last value received on the public channel is the value of `privateInfo`.

A mechanism that takes such leaks into account is said to be *progress-sensitive*. The most common way to prevent leaks through progress channels is to forbid loops whose execution depends on confidential information [5, 6], but it leads to the rejection of many secure programs, such as the following program that always terminates.

```
while privateInfo > 0 do
 privateInfo := privateInfo - 1
end; (*loop will always end, no matter the value of privateInfo*)
send 42 to publicChannel (*no leak*)
```

Listing 1.9: Loop that always terminates

Alternatively, to accept such programs, Moore et al. [7] use an oracle to determine the termination behavior of loops. If the oracle determines that a loop always terminates (like the one in Listing 1.9), then no following output could leak information through progress channels. On the other hand, if the oracle says that it may diverge, then an enforcer must take into account the fact that an output following the loop's execution could leak information.

Timing flows* If we consider that a user can, not only observe the progress of a program, but also time its execution, then this execution time can be used to leak sensitive information. Indeed, by timing how long the program of Listing 1.10 takes to complete, an observer could deduce whether or not `privateInfo` is greater than `0`.

```
send "Starting" to publicChannel;
if privateInfo > 0 then
  (*this branch takes 5 minutes to execute*)
else
  (*this branch takes 1 second to execute*)
end;
send "Finished!" to publicChannel
```

Listing 1.10: Leaking information through timing-channels

Mechanisms that take this kind of leak into account are said to be *timing-sensitive* [8]. The most common way to prevent such leaks is to ensure that the execution time of both branches is always the same. This is done by executing a "dummy" version of the other branch (see Listing 1.11).

```
send "Starting" to publicChannel;
if privateInfo > 0 then
  (*this branch takes 5 minutes to execute*)
  (*add 1 second of dummy operations*)
else
  (*this branch takes 1 second to execute*)
  (*add 5 minutes of dummy operations*)
end;
send "Finished!" to publicChannel
```

Listing 1.11: Preventing leaks through the timing channel

1.2 Enforcement mechanisms

Noninterference can be enforced using different mechanisms. There are three classes of mechanisms: static, dynamic and hybrid.

1.2.1 Static mechanisms

Static information-flow control mechanisms analyze a program before its execution to determine whether the program's execution will satisfy the appropriate information flow requirements. When facing uncertainty, static mechanisms are forced to approximate to be sound. While these approximations may lead to false positives (i.e., reporting that there is a leak of information when there is not), they do not lead to false negatives (i.e., reporting that there is no leak of information when there is).

Static enforcement mechanisms typically rely on one of two techniques for their analysis: type systems or program dependence graphs.

Type systems A *type system* is a set of rules that assign *types* to the various constructs of a program (e.g., variables, expressions, functions, statements) [11]. These types can then be used to detect potential execution errors (e.g., a function that takes an integer as parameter is instead given a string). For example, they can be used to detect that the command `output(x)` is not valid in the following program.

```
function output(string s) { (*)*output is typed as a function that takes a string as
   argument and returns nothing*)
 print "Output:" + s
}
x := 5; (*5 is an int, so x is typed as an int*)
y := "Hello"; (*"Hello" is a string, so y is typed as a string*)
output(y); (*ok because y is a string*)
output(x) (*error because x is not a string*)
```

The same idea can be applied to enforce noninterference [9, 10]. The only difference is that the types that are associated with the constructs now must contain information about their security level. Using these types, it is then possible to verify that the programs do not contain any illicit flow of information. For example in the listing below, variable `x` will be typed as a public integer, and similarly for variable `secret`. Using these types, an analysis will then be able to conclude that the **send** command is insecure and leaks information.

```
x := 5; (*5 is public, so x is typed as public*)
receive secret from secretFile; (*secretFile contains private information, so secret is
   also private*)
send secret to publicFile (*insecure explicit flow detected because secret is private and
   publicFile is public*)
```

Program dependence graphs A *program dependence graph* (PDG) [12] is a visual representation of the information flows that can occur in a program. In a PDG, each node represents a program statement or an expression and the edges represent dependence, of which there are two kinds:

- Data dependence: A solid edge, written $x \longrightarrow y$, means that statement x assigns a variable that is used in statement y.
- Control dependence: A dotted edge, written $x \dashrightarrow y$, means that the execution of y depends on the value of expression x (which is typically the condition of an if/while command).

A path from node x to node y means that information can flow from x to y. Conversely, if there are no paths from private inputs to public outputs, then the program is noninterferent. For example, in Figure 1.1 there is a path from node `pw` (`password`) to node `return match` which means that the value of `password` influences the value returned by the function `check`.

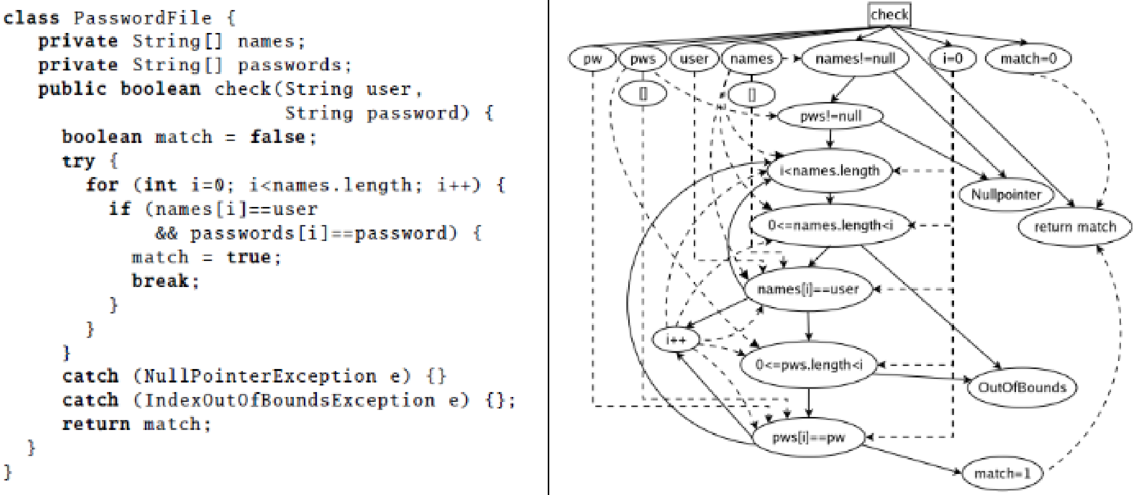

Figure 1.1: A Java password checker and its PDG [12]

1.2.2 Dynamic mechanisms

Dynamic information-flow control mechanisms accept or reject individual executions at run-time, without performing any static program analysis. Dynamic mechanisms are more permissive than static mechanisms as they allow the execution of insecure programs as long as the current execution is secure. For example, a static mechanism would reject the program of Listing 1.2.2 because not *all* executions of the program are noninterferent.

```
if randomValue = 3.141592654 then
  send secretInfo to publicChannel (*leak, but happens rarely*)
else
  send 1 to publicChannel (*no leak*)
end
```

Dynamic mechanisms on the other hand would reject only executions where `randomValue` equals `3.141592654`.

Some of the earliest information-flow control mechanisms, such as the one proposed by Bell-Lapadula [13], were purely dynamic. However, as seen in Section 1.1.1, enforcing noninterference involves considering not only the outputs that occurred, but also those that could have occurred in another execution. Since dynamic mechanisms do not perform any static analysis,

they are only aware of what happens in the current execution. For this reason, like static mechanisms, they are forced to make approximations.

In this section, we present four dynamic mechanisms: monitoring, instrumentation, secure multi-execution and multi-faceted variables.

Monitoring Monitors (e.g., [14]) observe the commands that are executed and can halt the execution of a program in order to prevent a leak from happening. A simple way to implement a monitor is by integrating it into the operational semantics of a programming language (more information on operational semantics is available in [15]). For example the rule (ASSIGN) of Figure 1.2 states that when an assignment is executed, the value and security level of variable x ($level_x$) are updated in the memory m by the value and security level of expression e.

$$\text{(ASSIGN)}\ \frac{m(e) = r}{\langle x := e, m\rangle \longrightarrow \langle \textbf{stop}, m[x \mapsto r, level_x \mapsto level_e]\rangle}$$

$$\text{(SEND)}\ \frac{\neg(isPrivate(level_x) \land isPublic(level_c))}{\langle \textbf{send}\ x\ \textbf{to}\ c, m\rangle \longrightarrow \langle \textbf{stop}, m\rangle}$$

Figure 1.2: Example of operational semantics rules

Using this information, noninterference can then be enforced: in Figure 1.2, the rule (SEND) uses the levels to make sure that no leak occurs when sending information on a channel.

Instrumentation *Instrumentation* is similar to monitoring but, instead of monitoring the execution of the program, commands are inserted into the programs to make the additional verifications that a monitor would have done at runtime. In other words, the monitor is inlined in the program's code.

For example, Listing 1.12 presents an instrumented version of a program in which a condition has been inserted to make sure that the output is only executed when there is no leak of information.

```
if level_x = private && level_y = public then
  fail
else
  send x to y
end
```

Listing 1.12: Possible instrumentation of a send command

The main advantage of instrumentation is that the runtime environment of the program does not need to be modified.

Secure multi-execution* *Secure multi-execution* [16] is a fairly new enforcement technique. The idea is to execute the program multiple times, once for each security level. For example, the program of the listings below would be executed twice: once to produce the public outputs (Listing 1.13) and once to produce the private outputs (Listing 1.14). To avoid leaking information when producing the public outputs, private information is replaced with default values. When producing the private outputs, the actual values will be used (as indicated in the comments).

```
(*privateInfo = 42*)
x := privateInfo; (*x's value is replaced with a default value*)
send 1 to publicChannel; (*outputs 1 to public observers*)
send x to publicChannel; (*outputs the default value*)
send x to privateChannel (*not executed, it is not a public output*)
```

Listing 1.13: Producing the public outputs

```
(*privateInfo = 42*)
x := privateInfo; (*x's value is 42*)
send 1 to publicChannel; (*not executed, it is not a private output*)
send x to publicChannel; (*not executed, it is not a private output*)
send x to privateChannel (*outputs 42 to private observers*)
```

Listing 1.14: Producing the private outputs

The main disadvantage of secure multi-execution is that it does not preserve the original semantics of the program (i.e., what the programmer wanted it to do) when a leak is present and can result in unexpected behaviors.

Multi-faceted variables* A similar idea to secure multi-execution is to allow the use of *multi-faceted variables* in programs [17]. A multi-faceted variable is a variable that is assigned a tuple of values, interpreted as the target of a map from security levels to actual values. For example, the variable `address` of Listing 1.15 is a multi-faceted variable which has different values for observers of level `public` and `private`.

```
address := ("London",  "221b Baker Street, London"); (*public value, private value*)
address := address + ", England"; (*concatenation is performed on all facets*)
send address to publicChannel; (*sends "London, England"*)
send address to privateChannel (*sends "221b Baker Street, London, England"*)
```

Listing 1.15: A program with multi-faceted variables

Each time an operation is made on a multi-faceted value (e.g., the string concatenation in Listing 1.15), it must be performed on all facets. When an input/output operation is done, the appropriate facet is used (e.g., the public value is used when outputting to a public channel). Though its performance is better than secure multi-execution (since not all commands are

executed multiple times), it requires existing programs to be rewritten and may become a burden to the programmer if there is a large number of security levels.

1.2.3 Hybrid mechanisms

Hybrid information-flow control mechanisms are mechanisms that use a combination of static and dynamic analysis (e.g., [18, 19, 20, 21, 22]). They are attractive as they offer the advantages of both static and dynamic analyses: the low runtime overhead of static approaches combined with the flexibility of dynamic mechanisms.

For example, a static analysis could be used to instrument only specific parts of the program (those that need it) instead of the whole program, or a dynamic analysis could be used to gather statically unknown runtime information.

1.3 Precision

A mechanism is said to be more precise than another if it rejects less programs while remaining sound. Usually, the more precise a mechanism is, the slower is its analysis. For this reason, enforcement mechanisms vary greatly in terms of precision. This difference in precision comes not only from the types of information flows that are taken into account (see Section 1.1.1), but also from the sensitivity of their analysis. In this section, we present the four types of sensitivities mentioned by Hammer et al. [23]: flow sensitive, context sensitive, object sensitive and path sensitive. We reuse a few of their examples.

Flow sensitive An analysis is said to be *flow sensitive* if it takes into consideration the order of statements. For example, a flow-sensitive approach would accept the program in Listing 1.16 as the value contained in variable `x` during the **send** command is public.

```
x := secret; (*x is private*)
x := 0; (*x is public*)
send x to publicChannel (*Flow insensitive:Leak, Flow sensitive:OK*)
```

Listing 1.16: Flow sensitive vs. flow insensitive

A flow-insensitive approach would reject this program because `x` contains, at one point, sensitive information. Though flow sensitivity increases precision, it complicates the enforcement of noninterference [24].

Context sensitive* An analysis is said to be *context sensitive* if "procedure calling context is taken into account, and separate information is computed for different calls of the same procedure; a context-insensitive analysis merges all call sites of a procedure" [23]. For example, Listing 1.17 presents a program where the function `f` is called twice: once with a private argument and once with a public argument.

```
int f(int x) { return x+42 }
secret := 1;
public := 2;
s := f(secret);
x := f(public);
send x to publicChannel
```

Listing 1.17: Example for context-sensitivity

A context-insensitive analysis would reject this program because one of the calls to function `f` returns a private value, even though variable `x` contains only public information.

Object sensitive* An analysis is said to be *object sensitive* if "different host objects for the same field (attribute) of an object are taken into account; object-insensitive analysis merges the information for a field over all objects of the same class." [23]. In other words, an analysis that is object sensitive is able to differentiate the different instances of a class. For example, Listing 1.18 presents a program with two objects of the class `File`: `x` and `y`. The field `path` contains private information in `x`, but public information in `y`.

```
class File {
  string path;
  File(path) { this.path = path }
}
(*x and y are both instances of class File*)
x := new File(secretPath); (*but x.path is private*)
y := new File(publicPath); (*and y.path is public*)
send y.path to publicChannel
```

Listing 1.18: Example for object-sensitivity

An object-insensitive analysis would not differentiate between the different instances of class `File`, and so, would mistakenly report that a leak occurs at the **send** of Listing 1.18 even though the contents of `y.path` is public.

Path sensitive* An analysis is said to be *path sensitive* if it takes into account the conditions necessary for an information flow to occur. For example in Listing 1.19, the value of variable `secret` will never be sent on `publicChannel`. Indeed, for a leak to occur, the following condition would have to be true : $(\exists i \in \mathbb{Z} \mid (i\%2 = 0) \wedge (i = 3))$ (which is impossible).

```
a[3] := secret;
if (i % 2 = 0) then
  send a[i] to publicChannel
end
```

Listing 1.19: Example for path-sensitivity

Constraint solvers can be used to verify if this flow is possible.

1.4 Conclusion

We have presented in this chapter the basics of information-flow control: the notion of noninterference, the different types of information flows, mechanisms and sensitivities. The information-flow control mechanisms presented in the following chapters all enforce variants of noninterference, take into account explicit and implicit flows and are hybrid.

1.5 Bibliography

[1] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, 1982, pp. 11–20. Available: https://doi.org/10.1109/SP.1982.10014

[2] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976. Available: http://doi.acm.org/10.1145/360051.360056

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, 2008, pp. 333–348. Available: https://doi.org/10.1007/978-3-540-88313-5_22

[4] D. M. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, 1997, pp. 156–169. Available: https://doi.org/10.1109/CSFW.1997.596807

[5] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006), 5-7 July 2006, Venice, Italy*, 2006, pp. 190–201. Available: https://doi.org/10.1109/CSFW.2006.16

[6] G. Smith and D. M. Volpano, "Secure information flow in a multi-threaded imperative language," in *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, 1998, pp. 355–364. Available: http://doi.acm.org/10.1145/268946.268975

[7] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *the ACM Conference on Computer and Communications Security,*

CCS'12, Raleigh, NC, USA, October 16-18, 2012, 2012, pp. 881–893. Available: http://doi.acm.org/10.1145/2382196.2382289

[8] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, 2012, pp. 99–110. Available: http://doi.acm.org/10.1145/2254064.2254078

[9] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003. Available: https://doi.org/10.1109/JSAC.2002.806121

[10] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996. Available: https://doi.org/10.3233/JCS-1996-42-304

[11] L. Cardelli, "Type systems," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 263–264, 1996. Available: http://doi.acm.org/10.1145/234313.234418

[12] C. Hammer, "Information flow control for java: a comprehensive approach based on path conditions in dependence graphs," Ph.D. dissertation, Karlsruhe Institute of Technology, 2009. Available: http://d-nb.info/996983112

[13] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE CORP BEDFORD MA, Tech. Rep., 1973.

[14] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, 2009, pp. 113–124. Available: http://doi.acm.org/10.1145/1554339.1554353

[15] G. D. Plotkin, "A structural approach to operational semantics," *J. Log. Algebr. Program.*, vol. 60-61, pp. 17–139, 2004.

[16] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 2010, pp. 109–124. Available: https://doi.org/10.1109/SP.2010.15

[17] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama, "Faceted execution of policy-agnostic programs," in *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*, 2013, pp. 15–26. Available: http://doi.acm.org/10.1145/2465106.2465121

[18] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *Proceedings of the 22nd IEEE Computer Security Foundations*

Symposium, CSF 2009, Port Jefferson, New York, USA, July 8-10, 2009, 2009, pp. 43–59. Available: https://doi.org/10.1109/CSF.2009.22

[19] F. Besson, N. Bielova, and T. P. Jensen, "Hybrid information flow monitoring against web tracking," in *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, 2013, pp. 240–254. Available: https://doi.org/10.1109/CSF.2013.23

[20] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your ifcexception are belong to us," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, 2013, pp. 3–17. Available: https://doi.org/10.1109/SP.2013.10

[21] G. L. Guernic, A. Banerjee, T. P. Jensen, and D. A. Schmidt, "Automata-based confidentiality monitoring," in *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues, 11th Asian Computing Science Conference, Tokyo, Japan, December 6-8, 2006, Revised Selected Papers*, 2006, pp. 75–89. Available: https://doi.org/10.1007/978-3-540-77505-8_7

[22] J. Magazinius, A. Russo, and A. Sabelfeld, "On-the-fly inlining of dynamic security monitors," *Computers & Security*, vol. 31, no. 7, pp. 827–843, 2012. Available: https://doi.org/10.1016/j.cose.2011.10.002

[23] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *Int. J. Inf. Sec.*, vol. 8, no. 6, pp. 399–422, 2009. Available: https://doi.org/10.1007/s10207-009-0086-1

[24] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, 2010, pp. 186–199. Available: https://doi.org/10.1109/CSF.2010.20

Chapter 2

Enforcing Information Flow by Combining Static and Dynamic Analysis

Authors: Andrew Bedford, Josée Desharnais, Théophane G. Godonou and Nadia Tawbi

Conference: *International Symposium on Foundations & Practice of Security (FPS)*

Status: peer reviewed; published[1]; presented

Year: 2013

2.1 Résumé

Ce chapitre présente une approche pour appliquer des politiques de flots d'information en utilisant une analyse basée sur les types suivie d'une instrumentation lorsque nécéssaire. Notre approche vise à réduire les faux positifs générés par l'analyse statique et à réduire la surcharge d'exécution introduite en instrumentant seulement lorsque nécéssaire. L'idée clé de notre approche est d'identifier ce qui est inconnu statiquement. Au lieu de rejeter des programmes qui pourraient possiblement faire fuir de l'information, nous les instrumentons pour vérifier lors de l'exécution si une fuite se produit réellement. Deux des particularités de notre approche sont que nous utilisons quatre types de sécurité, et aussi que nous faisons la distinction entre les variables et canaux de communications. Cette distinction nous permet d'associer les niveaux de sécurité aux canaux plutôt qu'aux variables, dont le niveau de sécurité change en fonction de l'information qu'elles contiennent.

[1]The published version is available at Springer via https://doi.org/10.1007/978-3-319-05302-8_6

2.2 Abstract

This chapter presents an approach to enforce information flow policies using a multi-valued type-based analysis followed by an instrumentation when needed. Our approach aims at reducing false positives generated by static analysis, and at reducing execution overhead by instrumenting only when needed. False positives arise in the analysis of real computing systems when some information is missing at compile time, for example the name of a file, and consequently, its security level. The key idea of our approach is to distinguish between "negative" and "may" responses. Instead of rejecting the possibly faulty commands, they are identified and annotated for the second step of the analysis; the positive and negative responses are treated as is usually done. This work is a hybrid security enforcement mechanism: the *maybe-secure* points of the program detected by our type based analysis are instrumented with dynamic tests. The novelty of our approach is the handling of four security types, but we also treat variables and channels in a special way. Programs interact via communication channels. Secrecy levels are associated to channels rather than to variables whose security levels change according to the information they store, thus the analysis is flow-sensitive.

2.3 Introduction

In today's world, we depend on information systems in many aspects of our lives. Those systems are interconnected, rely on mobile components and are more and more complex. Security issues in this context are a major concern, especially when it comes to securing information flow. How can we be sure that a program using a credit card number will not leak this information to an unauthorized person? Or that one that verifies a secret password to authenticate a user will not write it in a file with public access? Those are examples of information flow breaches in a program that should be controlled. Secure information flow analysis is a technique used to prevent misuse of data. This is done by restricting how data are transmitted among variables or other entities in a program, according to their security classes.

Our objective is to take advantage of the combination of static and dynamic analysis. We design a multi-valued type system to statically check noninterference for a simple imperative programming language. To the usual two main security levels, public (or *Low*) and private (or *High*), we add two values, *Unknown* and *Blocked*. The former was introduced in [1] and captures the possibility that we may not know, before execution, whether some information is public or private. Standard two-valued analysis has no choice but to be pessimistic with uncertainty and hence generate false positive alarms. If uncertainty arises during the analysis, we tag the instruction in cause: in a second step, instrumentation at every such point together with dynamic analysis will allow us to head to a more precise result than purely static approaches. We get reduced false alarms, while introducing a light runtime overhead by

instrumenting only when there is a need for it. In this chapter, we add a fourth security type, *Blocked*, which is used to tag a public channel variable that must not receive any information, even public, because its value (the name of the channel) depends on private information. As long as no information is sent over such a channel, the program is considered secure.

The program on the left of Figure 2.1 shows how the blocking type results in fewer false positive alarms. The figure also exhibit our analysis of the program (which we will explain later) as well as the output given by our implementation. The identifiers *privateChannel*, *publicChannel*, *highValue* and *lowValue* in all the examples are predefined constants. The security types L, H, U, B represent *Low*, *High*, *Unknown* and *Blocked*, respectively, *pc* is the security type of the context, and $_instr = L$ to tell that there is no need for instrumentation. The first four lines of the program would be rejected by other analyses, including [1], because channel c is assigned a *Low* channel in the **then** branch, which depends on a private condition, *highValue*. In our work, c is just marked as blocked ("$c \mapsto B chan$") when it is assigned a public channel in a private context. However, in the last line, an information of low content is sent to c, which cannot be allowed, as it would reveal information on our confidential condition *highValue*. It is because of this last line that the program is rejected by our analysis: without it, c is just typed as B.

Input to analyzer	Inference analysis		
	Environment	pc	i
if *highValue*		$pc_{if} = H$	2
then $c :=$ *publicChannel*	$G(2) = [_instr \mapsto L, c \mapsto B\ chan]$	H	3
else $c :=$ *privateChannel*	$G(3) = [_instr \mapsto L, c \mapsto H\ chan]$	H	4
end;	$G(1) = [_instr \mapsto L, c \mapsto B\ chan]$	H	4
send *lowValue* **to** c	**fail** since $c \mapsto B\ chan$		
Output : Error (Send) : Cannot send *lowValue* to channel c because it is blocked.			

Figure 2.1: Analysis of a program where an implicit flow may lead to a leak of information

The goal of our security analysis is to ensure noninterference, that is, to prevent inadvertent information leaks from private channels to public channels. More precisely, in our case, the goal is to ensure that 1) a well-typed program satisfies noninterference, 2) a program not satisfying noninterference is rejected 3) a program that may satisfy noninterference is detected and sent to the instrumentation step. Furthermore, we consider that programs interact with an external environment through communication *channels*, i.e., objects through which a program can exchange information with users (printing screen, file, network, etc.). In contrast with the work of Volpano et al. [2], variables are not necessarily channels, they are local and hence their security type is allowed to change throughout the program. This is similar to flow-sensitive typing approaches like the one of Hunt and Sands, or Russo and Sabelfeld [3, 4]. Our approach distinguishes clearly communication channels, through which the program interacts and which have a priori security levels, from variables, used locally. Therefore, our definition of noninterference applies to communication channels: someone observing the final information contained in communication channels cannot deduce anything about the initial content of the

channels of higher security level.

We aim at protecting against two types of flows, as explained in [5]: *explicit flow* occurs when the content of a variable is directly transferred to another variable, whereas *implicit flow* happens when the content assigned to a variable depends on another variable, i.e., the guard of a conditional structure. Thus, the security requirements are:

- explicit flows from a variable to a channel of lower security are forbidden;
- implicit flows where the guard contains a variable of higher security than the variables assigned are forbidden.

Our static analysis is based on the typing system of [1]; our contributions are an improvement of the type system to allow fewer false positives, by the introduction of the blocked type, and the instrumentation algorithm that we have developed and implemented [6].

The rest of this chapter is organized as follows. After describing in Section 2.4 the programming language used, we present the type system ensuring that information will not be leaked improperly in Section 2.5. The inference algorithm is presented in Section 2.6. The instrumentation algorithm is presented in Section 2.7. Section 2.8 is dedicated to related work. We conclude in Section 2.9.

2.4 Programming Language

We illustrate our approach on a simple imperative programming language, introduced in [1], a variant of the one in [7], which was adapted to deal with the communication via channels.

2.4.1 Syntax

Let $\mathcal{V}ar$ be a set of identifiers for variables, and $\mathcal{C}$ a set of communication channel names. Throughout the chapter, we use generically the following notation: variables are $x \in \mathcal{V}ar$, and there are two types of constants: $n \in \mathbb{N}$ and $nch \in \mathcal{C}$. The syntax is as follows:

$$\begin{array}{lll}
\textit{(phrases)} & p ::= & e \mid c \\
\textit{(expressions)} & e ::= & x \mid n \mid nch \mid e_1 \textbf{ op } e_2 \\
\textit{(commands)} & c ::= & \textbf{skip} \mid x := e \mid c_1; c_2 \\
 & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \textbf{ end} \mid \textbf{while } e \textbf{ do } c \textbf{ end} \mid \\
 & & \textbf{receive}_\mathbf{c}\ x_1 \textbf{ from } x_2 \mid \\
 & & \textbf{receive}_\mathbf{n}\ x_1 \textbf{ from } x_2 \mid \\
 & & \textbf{send } x_1 \textbf{ to } x_2
\end{array}$$

Values are integers (we use zero for false and nonzero for true), or channel names. The symbol **op** stands for arithmetic or logic binary operators on integers and comparison operators on channel names. Commands are mostly the standard instructions of imperative programs.

We suppose that two programs can only communicate through channels (which can be, for example, files, network channels, keyboards, computer screens, etc.). We assume that the program has access to a pointer indicating the next element to be read in a channel and that the send to a channel would append an information in order for it to be read in a first-in-first-out order. When an information is read in a channel it does not disappear, only the read pointer is updated, the observable content of a channel remains as it was before. Our programming language is sequential; we do not claim to treat concurrency and communicating processes as it is treated in [8, 9]. We consider that external processes can only read and write to public channels. The instructions related to accessing channels deserve further explanations.

The instruction $\mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2$ stands for "receive content". It represents an instruction that reads a value from a channel with name x_2 and assigns its content to x_1. The instruction $\mathbf{receive_n}\ x_1\ \mathbf{from}\ x_2$ stands for "receive name". Instead of getting data from the channel, we receive another channel name, which might be used further in the program. This variable has to be treated like a channel. The instruction $\mathbf{send}\ x_1\ \mathbf{to}\ x_2$ is used to output on a channel with name x_2 the content of the variable x_1. The need for two different receive commands is a direct consequence of our choice to distinguish variables from channels. It will be clearer when we explain the typing of commands, but observe that this allows, for example, to receive a private name of channel through a public channel[2]: the information can have a security level different from its origin's. This is not possible when variables are observable.

2.4.2 Semantics

(ASSIGN)	$$\frac{\langle e, \mu\rangle \rightarrow_e v}{\langle \mathbf{x := e}, \mu\rangle \rightarrow \mu[x \mapsto v]}$$
(RECEIVE-CONTENT)	$$\frac{x_2 \in \mathrm{dom}(\mu) \qquad read(\mu(x_2)) = n}{\langle \mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2, \mu\rangle \rightarrow \mu[x_1 \mapsto n]}$$
(RECEIVE-NAME)	$$\frac{x_2 \in \mathrm{dom}(\mu) \qquad read(\mu(x_2)) = nch}{\langle \mathbf{receive_n}\ x_1\ \mathbf{from}\ x_2, \mu\rangle \rightarrow \mu[x_1 \mapsto nch]}$$
(SEND)	$$\frac{x_1 \in \mathrm{dom}(\mu)}{\langle \mathbf{send}\ x_1\ \mathbf{to}\ x_2, \mu\rangle \rightarrow \mu, update(\mu(x_2), \mu(x_1))}$$
(CONDITIONAL)	$$\frac{\langle e, \mu\rangle \rightarrow_e n \qquad n \neq 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end}, \mu\rangle \rightarrow \langle c_1, \mu\rangle}$$
	$$\frac{\langle e, \mu\rangle \rightarrow_e n \qquad n = 0}{\langle \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end}, \mu\rangle \rightarrow \langle c_2, \mu\rangle}$$

Figure 2.2: A few rules of the structural operational semantics

The behavior of programs follows a commonly used operational semantics [1]; we present a

[2]but not the converse, to avoid implicit flow leaks

few rules in Figure 2.2. An instruction p is executed under a memory map $\mu : \mathcal{V}ar \rightarrow \mathbb{N} \cup \mathcal{C}$. Hence the semantics specifies how *configurations* $\langle p, \mu \rangle$ evolve, either to a value, another configuration, or a memory. Evaluation of expressions under a memory involves no "side effects" that would change the state of memory. In contrast, the role of commands is to be executed and change the state. Thus we have two evaluation rules: $\langle e, \mu \rangle$ leads to a value resulting from the evaluation of expression e on memory μ; this transition is designated by $\rightarrow_e$. Finally, $\langle c, \mu \rangle$ leads to a memory produced by the execution of command c on memory μ; this transition is designated by $\rightarrow$.

We explain the rules that manipulate channels. The instructions $\mathbf{receive_c}$ x_1 **from** x_2 and $\mathbf{receive_n}$ x_1 **from** x_2 are semantically evaluated similarly. Information from the channel x_2 is read and assigned to the variable x_1. The distinctive feature of the rule RECEIVE-CONTENT is that the result of evaluation is an integer variable, while for the rule RECEIVE-NAME, the result is a channel name. Here, we introduce a generic function *read(channel)* that represents the action of getting information from a channel (eg. get a line from a file, input from the keyboard, etc.). The content of a channel remains the same after both kinds of receive.

The instruction **send** x_1 **to** x_2 updates the channel x_2 with the value of the variable x_1. This is done by the generic function *update(channel, information)*, which represents the action of updating the channel with some information. Note that the content of the variable x_2, that is, the name of the channel, does not change; hence μ stays the same. The content of the channel is updated after a **send**.

2.5 Security Type System

We now present the security type system that we use to check whether a program of the language described above, either satisfies noninterference, may satisfy it or does not satisfy it. It is an improvement of the one introduced in [1]: we add a security level, B, to tag a channel that should be blocked.

The security types are defined as follows:

$$\begin{array}{ll} \textit{(data types)} & \tau ::= L \mid U \mid H \mid B \\ \textit{(phrase types)} & \rho ::= \tau\ val \mid \tau\ chan \mid \tau\ cmd \end{array}$$

We consider a set of four security levels $SL = \{L, U, H, B\}$. This set is extended to a lattice $(SL, \sqsubseteq)$ using the following order: $L \sqsubseteq U \sqsubseteq H \sqsubseteq B$ (we use freely the usual symbols $\sqsupseteq$ and $\sqsupset$). It is with respect to this order that the supremum $\sqcup$ and infimum $\sqcap$ over security types are defined. We lift this order to phrase types in the trivial way, and assume this returns $\perp$ when applied to phrases of different types, e.g., $H\ chan \sqcup H\ val = \perp$. When typing a program, security types are assigned to variables, channels and commands, hence phrase types – and to the context of execution. The meaning of types is as follows. A variable of type $\tau\ val$ has a content of security type τ; a channel of type $\tau\ chan$ can store information of type τ or lower

(indeed, a private channel must have the possibility to contain or receive both private and public information). The security typing of commands is standard, but has a slightly different meaning: a command of type τ *cmd* is guaranteed to only allow flows into channels whose security types are τ or higher. Hence, if a command is of type L *cmd* then it may contain a flow to a channel of type L *chan*. Type B will only be assigned to channels, to indicate that they were of type L *chan* but must be blocked, to avoid an implicit flow.

Our type system satisfies two natural properties: *simple security*, applying to expressions and *confinement*, applying to commands [7]. *Simple security* says that an expression e of type τ *val* or τ *chan* contains only variables of level τ or lower. Simple security ensures that the type of a variable is consistent with the principle stated in the precedent paragraph. *Confinement* says that a command c of type τ *cmd* executed under a context of type *pc* allows flows only to channels of level $\tau \sqcup pc$ or higher, in order to avoid a flow from a channel to another of lower security (H to L for example). Those two properties can be used to prove noninterference (see [10]).

Our typing rules are shown in Figure 2.3. They are the same as in [1] except for the three rules that deal with channels. A *typing judgment* has the form $\Gamma, pc \vdash p : \rho, \Gamma'$, where Γ and Γ' are typing environments, mapping variables to a type of the form τ *val* or τ *chan* that represents their security level; *pc* is the security type of the context. The program is typed with a context of type L; according to the security types of conditions, some blocks of instructions are typed with a higher context, as will be explained later. The typing judgment can be read as: within an initial typing environment Γ and a security type context *pc*, the command p has type ρ, yielding a final environment Γ'. When the typing environment stays unchanged, Γ' is omitted. Since the type of channels is constant, there is a particular typing environment for channel constants, named *TypeOf_ Channel* that is given before the analysis. In the rules, α stands for either the label *val* or *chan*, depending on the context.

We use, as in [1], a special variable *_ instr*, whose type (maintained in the typing environment map) tells whether or not the program needs instrumentation. The initial value of *_ instr* is L; if the inference algorithm detects a need for instrumentation, its value is changed to U, H or B, depending on the rule applied, most of the time depending on the type of a channel. When it is updated, the supremum operator is always involved to make sure that the need for instrumentation is recorded until the end.

We need to define three operators, two of which on typing environments: $\Gamma \dagger [x \mapsto \rho]$ and $\Gamma \sqcup \Gamma'$. The former is a standard update, where the image of x is set to ρ, no matter if x is in the original domain of Γ or not. For the conditional rule in particular, we need a union of environments where common value variables must be given, as security type, the supremum of the two types, and where channel variables are given type U if they differ and none of them is blocked.

(CHAN_S) $\dfrac{TypeOf_Channel(nch) = \tau}{\Gamma, pc \vdash nch : \tau\ chan}$ (INT_S) $\Gamma, pc \vdash n : L\ val$

(OP_S) $\dfrac{\Gamma, pc \vdash e_1 : \tau_1\ \alpha, \qquad \Gamma, pc \vdash e_2 : \tau_2\ \alpha}{\Gamma, pc \vdash e_1\ \mathbf{op}\ e_2 : (\tau_1 \sqcup \tau_2) val}$ (VAR_S) $\dfrac{\Gamma(x) = \tau\ \alpha}{\Gamma, pc \vdash x : \tau\ \alpha}$

(SKIP_S) $\Gamma, pc \vdash\ \mathbf{skip}\ :\ H\ \ cmd$

(ASSIGN-VAL_S)
$$\frac{\Gamma, pc \vdash e : \tau\ val}{\Gamma, pc \vdash x := e : (\tau \sqcup pc)\ cmd, \Gamma \dagger [x \mapsto (\tau \sqcup pc) val]}$$

(ASSIGN-CHAN_S)
$$\frac{\Gamma, pc \vdash e : \tau\ chan}{\Gamma, pc \vdash x := e : \tau\ cmd, \Gamma \sqcup [_\ instr \mapsto HL_L^L(pc, \tau)] \dagger [x \mapsto HL_\tau^B(pc, \tau) chan]}$$

(RECEIVE-CONTENT_S)
$$\frac{\Gamma(x_2) = \tau\ chan}{\Gamma, pc \vdash \mathbf{receive_c}\ x_1\ \mathbf{from}\ x_2 : (\tau \sqcup pc)\ cmd, \Gamma \dagger [x_1 \mapsto (\tau \sqcup pc) val]}$$

(RECEIVE-NAME_S)
$$\frac{\Gamma(x_2) = \tau\ chan}{\Gamma, pc \vdash \mathbf{receive_n}\ x_1\ \mathbf{from}\ x_2 : \tau\ cmd, \Gamma \sqcup [_\ instr \mapsto HL_\tau^L(pc, \tau)] \dagger [x_1 \mapsto HL_{U \sqcup \tau}^B(pc, \tau) chan]}$$

(SEND_S)
$$\frac{\Gamma(x_1) = \tau_1\ \alpha \qquad \Gamma(x_2) = \tau\ chan \qquad \neg((\tau_1 \sqcup pc) = H \wedge \tau = L) \qquad \tau \neq B}{\Gamma, pc \vdash \mathbf{send}\ x_1\ \mathbf{to}\ x_2 : \tau\ \ cmd, \Gamma \sqcup [_\ instr \mapsto HL_L^U(\tau_1 \sqcup pc, \tau)]}$$

(CONDITIONAL_S)
$$\frac{\Gamma, (pc \sqcup \tau_0) \vdash c_1 : \tau_1\ cmd, \Gamma' \qquad \Gamma, pc \vdash e : \tau_0 val \qquad \Gamma, (pc \sqcup \tau_0) \vdash c_2 : \tau_2\ cmd, \Gamma'' \qquad \Gamma' \sqcup \Gamma'' \sqsupset \bot}{\Gamma, pc \vdash\ \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2\ \mathbf{end} : (\tau_1 \sqcap \tau_2)\ cmd, \Gamma' \sqcup \Gamma''}$$

(LOOP1_S)
$$\frac{\Gamma, pc \vdash e : \tau_0 val \qquad \Gamma, (pc \sqcup \tau_0) \vdash c : \tau\ \ cmd, \Gamma' \qquad \Gamma = \Gamma \sqcup \Gamma' \sqsupset \bot}{\Gamma, pc \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} : \tau\ \ cmd, \Gamma \sqcup \Gamma'}$$

(LOOP2_S)
$$\frac{\Gamma, (pc \sqcup \tau_0) \vdash c : \tau\ \ cmd, \Gamma' \quad \Gamma \neq \Gamma \sqcup \Gamma' \sqsupset \bot \qquad \Gamma, pc \vdash e : \tau_0 val \quad \Gamma \sqcup \Gamma', (pc \sqcup \tau_0) \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} : \tau'\ cmd, \Gamma''}{\Gamma, pc \vdash \mathbf{while}\ e\ \mathbf{do}\ c\ \mathbf{end} : \tau'\ cmd,\ \ \Gamma''}$$

(SEQUENCE_S)
$$\frac{\Gamma, pc \vdash c_1 : \tau_1\ cmd, \Gamma' \qquad \Gamma', pc \vdash c_2 : \tau_2\ cmd, \Gamma''}{\Gamma, pc \vdash c_1; c_2 : (\tau_1 \sqcap \tau_2)\ cmd, \Gamma''}$$

Figure 2.3: Typing rules

Definition 2.1. *The supremum of two environments is given as* $dom(\Gamma \sqcup \Gamma') = dom(\Gamma) \cup dom(\Gamma')$*, and*

$$\Gamma \sqcup \Gamma'(x) = \begin{cases} \Gamma(x) & \text{if } x \in dom(\Gamma) \setminus dom(\Gamma') \\ \Gamma'(x) & \text{if } x \in dom(\Gamma') \setminus dom(\Gamma) \\ U\ chan & \text{if } B chan \neq \Gamma(x) = \tau\ chan \neq \tau' chan = \Gamma'(x) \neq B chan \\ \Gamma(x) \sqcup \Gamma'(x) & \text{otherwise.} \end{cases}$$

Note that $\Gamma \sqcup \Gamma'(x)$ can return $\bot$ if Γ and Γ' are incompatible on variable x, for example if $\Gamma(x)$ is a value, and $\Gamma'(x)$ is a channel (this can only happen if Γ and Γ' come from different branches of an **if** command).

In the three rules that modify a channel, ASSIGN-CHAN_S, RECEIVE-NAME_S et SEND_S, the following operator is also used.

Definition 2.2. *The function HL computes the security level of _instr and channel variables in the three typing rules where a channel is modified.*

$$HL^{\psi}_{\nu}(pc,\tau) = \begin{cases} \psi & \text{if } (pc,\tau) = (H,L) \\ U & \text{if } (pc,\tau) \in \{(U,L),(U,U),(H,U)\} \\ \nu & \text{otherwise.} \end{cases} \quad \text{where } \psi,\nu,pc,\tau \in SL.$$

The notation *HL* refers to a downward flow "H to L" because this (handy and maybe tricky) function encodes (with ψ and ν), in particular, how such a flow from pc to τ should be handled. When it is clear that there is a downward flow, from H to L, then *HL* returns type ψ. When we are considering the security type of a channel variable, ψ is either U or B. Such a flow may not lead to a rejection of the program, nor to an instrumentation: when a variable is blocked, there is no need to instrument. For other flows, the analysis distinguishes between safe flows and uncertain ones. For example, flows from U to H are secure, no matter what the types of uncertain variables actually are at runtime (L or H). In these cases, $HL^{\psi}_{\nu}(pc,\tau)$ returns ν. However, depending on the actual type of the U variable at runtime, a flow U to L, from U to U or from H to U may be secure or not. A conservative analysis would reject a program with such flows but ours will tag the program as needing instrumentation and will carry on the type analysis. Hence, in these cases, *HL* will return U.

In related work, there are *subtyping judgements* of the form $\rho_1 \subseteq \rho_2$ or $\rho_1 \leq \rho_2$ [7, 2]. For instance, given two security types τ and τ', if $\tau \subseteq \tau'$ then any data of type τ can be treated as data of type τ'. Similarly, if a command assigns contents only to variables of level H or higher then, *a fortiori*, it assigns only to variables L or higher; thus we would have $H\ cmd \subseteq L\ cmd$. In our work, we integrated those requirements directly in the typing rules. Instead of using type coercions, we assign a fixed type to the instruction according to the more general type. For two expressions e_1 and e_2 of type τ_1 and τ_2 respectively, e_1 **op** e_2 is typed with $\tau_1 \sqcup \tau_2$. For two commands c and c' typed τ and τ', the composition through sequencing or conditionals is typed with $\tau \sqcap \tau'$.

We now comment the typing rules that are modified with respect to [1]. ASSIGN-CHAN_S and RECEIVE-NAME_S both modify a channel variable and they make use of the function HL^{ψ}_{ν}. The usual condition for the modification of a channel would be to avoid downward flow by imposing $pc \sqsubseteq \tau$ or, as in [1], $pc \preceq \tau$; the latter is a weakening of the former, that returns false only if $pc = H$ and $\tau = L$. In this chapter, we chose to only reject a program if an unauthorized **send** is performed. If we detect an implicit flow in ASSIGN-CHAN_S or RECEIVE-NAME_S, that is, $pc = H$ and $\tau = L$, we rather block the assigned channel (by $\psi = B$ in HL^{B}_{-}), as in the program of Figure 2.1; if the channel is never used, a false positive has been avoided. If the channel is blocked, there is no need for instrumentation, hence $\psi = L$ in HL^{L}_{-} for both rules. In RECEIVE-NAME_S, we must call instrumentation when τ is U

or H to prevent a downward flow from x_2 to x_1. In that case, the channel variable obtains security type $U \sqcup \tau$ because its type is unknown: we could receive the name of a private channel on a public one (but could not read on in). In ASSIGN-CHAN_S, this type is τ, the type of the assigned expression.

The rule for SEND_S states that the typing fails in two situations where the leak of information is clear: either the channel to which x_1 is sent is blocked ($\tau = B$), or it is of type L and either the context or the variable sent has type H ($(\tau_1 \sqcup pc) = H$). An example where $\tau = B$ was just discussed above. If the typing does not fail, the instrumentation will be called in each case where there is a possibility, at runtime, that $\tau_1 \sqcup pc$ be H while the channel has type L; those are the cases $(\tau_1 \sqcup pc, \tau) \in \{((U, L), (U, U), (H, U)\}$. The "$\psi$ branch" in the definition of HL^{ψ}_{-} is useless, as it is a case where the typing rejects the program.

The rule CONDITIONAL_S requires to type the branches c_1 and c_2 under the type context $pc \sqcup \tau_0$, to prevent downward flows from the guard to the branches.

We now explain why $\sqcup$ is defined differently on channel variables and value variables. If Γ and Γ', the environments associated to the two branches of the **if** command, differ on a value variable, we choose to be pessimistic, and assign the supremum of the two security types. A user who prefers to obtain fewer false positive could assign type U to this variable, and leave the final decision to dynamic analysis. In the case of channel variables, we do not have the choice: different unblocked channels must obtain the type $U\,chan$. The program on the left of Figure 2.7 illustrates why. The last line of the program would require that c be typed as $L\,chan$ so that the program be rejected. However, since the **else** branch makes c a private information, a command like **send** c **to** *publicChannel* should also be rejected, and hence in this case we would like that c had been typed $H\,chan$. Hence we must type c as $U\,chan$, justifying the definition of $\sqcup$. Interestingly, this also shows that in our setting, the uncertain typing is necessary.

We conclude this section by discussing occurrences of false positive alarms. A rejection can only happen from the application of the rule SEND_S: either the channel to which x_1 is sent is blocked, or it is of type L and the context, or the variable sent, is of type H. According to our rules, type L can only be assigned if it is the true type of the variable, but H can be the result of a supremum taken in rule CONDITIONAL_S or LOOP_S. False positive can consequently occur from typing an **if** or **while** command whose guard always prevent a "bad" branch to be taken. This is unavoidable in static analysis, unless we want to instrument any uncertainty arising from the values of guards. Nevertheless, our inference typing rules prevent more false positives than previous work through the blocking of channels and the unknown types U.

2.6 Inference Algorithm

The inference algorithm implements the specification given by the type system together with some refinements we adopted in order to prepare for the instrumentation step. The refinements consist in keeping track of the command line number and of the generated environment for this command. Although it may seem overloading, this strategy lightens the dynamic step since it avoids type inference computation whenever it is already done. The algorithm is implemented as the function **`Infer`** which is applied to the current typing environment, $g_e : \mathcal{V}ar \to \{L, H, U, B\}$, a number identifying the current command to be analyzed, the command line i, the security level of the current context, pc, and the actual command to be analyzed, c . Along the way, **`Infer`** returns a typing environment representing the environment valid after the execution of the command c and an integer representing the number identifying the next command to be analyzed. **`Infer`** updates $G : int \to (\mathcal{V}ar \to \{L, H, U, B\})$ as a side effect; G associates to each command number a typing environment valid after its execution. Recall that the environment associates to a specific variable $_instr$ a security level. After the application of the inference algorithm, if the program is not rejected and the resulting environment associates U, H or B to $_instr$ then the program needs instrumentation, otherwise it is safe w.r.t. noninterference.

To analyze a program P, **`Infer`** is invoked with $g_e = [_instr \ \mapsto L]$, $i = 0$, $pc = L$ and $c = P$. The inference algorithm uses a set of utility functions that implement some operators, functions and definitions appearing in the typing system. Their meaning and their implementation are straightforward. Here is the list of these functions. The set $SecType$ stands for $\{\tau \ \ v : \tau \in \{L, U, H, B\}, v \in \{val, chan\}\}$, t and t_i ranges over $SecType$, and g_i ranges over Env, **`lessOrEqual`** implements $\sqsubseteq$, **`inf`** and **`sup`** implement respectively the infimum and the supremum of two security levels. **`supEnv`** implements the supremum of two environments, as in Definition 2.1. **`infV`** : $SecType \times SecType \to SecType \cup \{\bot_T\}$ returns $\bot_T$ if the two security types do not have the same nature. If the nature is the same, then it returns a security type where the security level is the infimum of the two security types given as argument, **`supV`** : $SecType \times SecType \to SecType \cup \{\bot_T\}$ behaves the same way as **`infV`** except that it returns a security type where the security level is the supremum of the two security types given as argument, **`incBottomEnv`** : $Env \to bool$ returns true if at least one variable is associated to $\bot_T$ in its parameter, **`updateEnv`** : $Env \times \mathcal{V}ar \times SecType \to Env$ implements $\Gamma \dagger [x \mapsto \rho]$, **`eqEnv`** : $Env \times Env \to bool$ checks if two environments are equal. It returns true if the two environments have the same domain and all their variables have the same security type. It returns false otherwise, **`evalN`** : $SecType \to \{val, chan\}$, extracts the nature of a security type (val or $chan$), **`evalT`** : $SecType \to \{L, U, H, B\}$, extracts the level of a security type, **`inferE`** : $Env \times Exp \to SecType$ returns the highest security type of the variables present in the expression to which it is applied, and **`HL`** : $\{L, U, H, B\}^4 \to \{L, U, H, B\}$ implements the function HL as in Definition 2.2.

```
Infer: Env × int × Sec × cmd → Env × int

Infer(g_e, i, pc, c) =
    case c of
        skip : G(i) = g_e
            return (G(i), i + 1)
        x := e :
            τ = evalT(inferE(g_e, e))
            case evalN(inferE(g_e, e) ) of
                val: G(i) = updateEnv(g_e, x, sup(pc, τ)  val)
                    return (G(i), i + 1)
                chan: _instr_t = HL(L, L, pc, τ)
                    x_t = HL(B, τ, pc, τ)
                    _instr_t2 = g_e(_instr)
                    G(i) = updateEnv(updateEnv(g_e, _instr, sup(_instr_t, _instr_t2)), x, x_t  chan)
                    return (G(i), i + 1)
        receive_c x_1 from x_2 :
            τ = evalT(g_e(x_2))
            G(i) = updateEnv(g_e, x_1, sup(pc, τ)  val)
                return (G(i), i + 1)
        receive_n x_1 from x_2 :
            τ = evalT(g_e(x_2))
            _instr_t = HL(L, τ, pc, τ)
            _instr_t2 = g_e(_instr)
            x1_t = HL(B, sup(U, τ), pc, τ)
            G(i) = updateEnv(updateEnv(g_e, _instr, sup(_instr_t, _instr_t2)), x_1, x1_t  chan)
            return (G(i), i + 1)
        send x_1 to x_2 :
            τ_1 = evalT(g_e(x_1))
            τ = evalT(g_e(x_2))
            _instr_t = HL(U, L, sup(τ_1, pc), τ)
            _instr_t2 = g_e(_instr)
            if((τ ≠ B) and ¬(sup(τ_1, pc) = H and τ = L)))
                then G(i) = updateEnv(g_e, _instr, sup(_instr_t, _instr_t2))
                else fail
            return (G(i), i + 1)
        c_1; c_2 :
            (g_1, j) = Infer(g_e, i, pc, c_1)
            (g_2, k) = Infer(g_1, j, pc, c_2)
            return (g_2, k)
        if e then c_1  else c_2 end:
            t = evalT(inferE(g_e, e))
            pc_if = sup(pc, t)
            (g_1, j) = Infer(g_e, i + 1, pc_if, c_1)
            (g_2, k) = Infer(g_e, j, pc_if, c_2)
            if(¬incBottomEnv(supEnv(g_1, g_2))) then G(i) = supEnv(g_1, g_2)
                else fail
            return (G(i), k)
        while e do c end:
            t = evalT( inferE(g_e, e))
            pc_while = sup(pc, t)
            (g_e', j) = Infer(g_e, i + 1, pc_while, c)
            if (eqEnv(g_e, supEnv(g_e, g_e'))) and (¬incBottomEnv(supEnv(g_e, g_e'))))
                then g_res = supEnv(g_e, g_e')
                else if (¬eqEnv(g_e, supEnv(g_e, g_e'))) and (¬incBottomEnv(supEnv(g_e, g_e'))))
                    then (g_res, j) = Infer(supEnv(g_e, g_e'), i, pc_while, while e do c end)
                    else fail
            G(i) = supEnv(G(i), gres)
            return (g_res, j)
```

Figure 2.4: Inference Algorithm

The inference algorithm **Infer** is presented in Figure 2.4. Some examples of its output are presented in the following section.

2.7 Instrumentation

```
Instrument: cmd * int → int

Instrument(c, i) =    case c of
    skip : IC ∧ " skip; "
     return (i + 1)
     x := e :
       τ = evalT(inferE(G(i), e))
       case evalN(inferE(G(i), e)) of
         val: IC = IC∧ "x := e ; "
               if (τ = U) then
                    IC = IC∧"updateEnv(G(i), x, sup(evalT(TypeOf_Expression(e)), top(pc))val); "
               end ;
               IC = IC∧ " updateEnv(g_M, x, G(i)(x)) ; "
           return (i + 1)
         chan: IC = IC∧ "x := e ; "
                if  (τ = U) then
                     IC = IC∧" updateEnv(G(i), x, TypeOf_Channel(e)); "
                end
                IC = IC∧ " updateEnv(g_M, x, G(i)(x)); "
           return (i + 1)
     receive_c x1 from x2 :
       IC = IC ∧ "receive_c x1 from x2; "
       if  (G(i)(x1) = Uval)  then
            IC = IC∧ " updateEnv(G(i), x1, sup(evalT(TypeOf_Expression(x2)), top(pc))val) ; "
       end
       IC = IC∧ "updateEnv(g_M, x1, G(i)(x1)) ; "
       return (i + 1)
     receive_n x1 from  x2 :
       IC = IC∧ "receive_n x1 from x2 ";
       if  (G(i)(x2) != L chan)
         then  IC = IC∧ " if TypeOf_Channel(x1) = L chan and TypeOf_Channel(x2) = H chan
            then updateEnv(G(i), x1, B chan)
            else updateEnv(G(i), x1, TypeOf_Channel(x1))
           end "
         else IC = IC∧ "updateEnv(G(i), x1, TypeOf_Channel(x1))"
         end
       IC = IC∧ " updateEnv(g_M, x1, G(i)(x1)); "
     return (i + 1)
     c1; c2 :
       j = Instrument(c1, i); k = Instrument(c2, j)
       return k
     send x1 to x2 :
            IC = IC∧ " tau = TypeOf_Expression(x2) ; tau1 = TypeOf_Expression(x1);
            if(((tau = L chan) and (sup(evalT(tau1), top(pc)) = H)) or (tau = B chan))
            then fail else send x1 to x2 end; "
     return (i + 1)
     if e then c1 else c2 end :
       IC = IC∧ "push(sup(top(pc), evalT(TypeOf_Expression(e))), pc) ;
       if e then "
       j = Instrument(c1, i + 1)
      IC = IC∧ "else "
       k = Instrument(c2, j)
       IC = IC∧ "end;
       pop (pc); "
     return k
     while e to c end:
     IC = IC∧
       "push(sup(top(pc), evalT(TypeOf_Expression(e))), pc);
       while e do "
     j = Instrument(c, i + 1)
     IC = IC∧ "end ;
       pop (pc) ; "
     return j
```

Figure 2.5: Instrumentation algorithm.

Our instrumentation is based on the inference algorithm. It is a new contribution w.r.t. [1]. It inserts commands in the program so that, during execution, the program will update the security level of variables which were unknown (U) statically. Each instruction is treated with its corresponding line number and its context security level. Instructions may be inserted to prevent unsecure executions. The instrumentation algorithm is shown in Figure 2.5; it is given a command *cmd* to instrument and the number of this command. The algorithm updates $IC : String$ as a side effect, which is the instrumented program; it uses the matrix of typing environments G produced by the inference algorithm, which is a global variable. *G(i)* refers to the typing environment of instruction i, and hence *G(i)(x)* is the security type of variable x at instruction i.

Commands are inserted so that the instrumented program will keep a table g_M of the security levels of variables, picking the already known types in G. This table is also a global variable. g_M offers two advantages, it keeps track of the most recent values of the variables. No further analysis is necessary to find which instruction was the last to modify the variables. It is also easier to read the value from a table than from the matrix G. The usefulness of g_M can be shown with the following example.

$\mathbf{receive_n}$ c **from** *publicChannel*;
$\mathbf{receive_c}$ a **from** *publicChannel*;
if $(a \mod 2 \neq 0)$ **then**
    $\mathbf{receive_c}$ a **from** c
end;
send a **to** *publicChannel*

The inference algorithm determines after the first instruction that the type of c is $U\ chan$. Variable a, before the **if** command, has the type $L\ val$. The static analysis will conclude that the type of a after executing the **if** command is $U\ val$. If the instrumented program updates the variables immediately in G, the type of a would be $H\ val$. The following **send** would be considered unsecure no matter what the dynamic value of a is. Our instrumentation will insert instructions that put in g_M the last type of a when it was read. So depending on whether the execution of the instrumented program enters the **then** branch or not, a will take either the security level of c, or it will keep the security level of *publicChannel*. This will allow the instrumented program to be rejected during execution only if it is actually unsecure.

A set of utility functions are predefined in the target language and used by the instrumented programs. Function TypeOf_Channel serves as a register for the constant channels defined prior to the execution. Function TypeOf_Expression returns the actual type of an expression: it uses the information of g_M for variables, TypeOf_Channel for actual channels and takes the supV of these values when the expression is $e_1 \mathbf{op}\ e_2$. TypeOf_Expression and TypeOf_Channel

are commands executed by the instrumented program. To prevent implicit flows, commands are inserted so that the instrumented program will keep a stack of contexts *pc*. Each time the execution branches on an expression, whether it is in an **if** or a **while** command, the context is pushed onto the stack. The context is the supremum of the type of expression e and the context in which the actual command is executed. The last context is popped from the stack everytime the execution of a branching command finishes. The stack *pc* is initially empty and the result of reading an empty stack is always L. The functions `push` and `pop` are used to manipulate the stack of contexts during the execution of the instrumented program. The remaining functions are an implementation of their counter part in the algorithm **`Infer`**.

The analysis and instrumentation algorithms have been implemented. The implementation is divided into two parts : an analyzer and an interface. The analyzer is written in OCaml. It uses OCamllex and OCamlyacc to generate the lexer and parser. In order to maximize the portability of our application, we use OCaml-Java to compile our OCaml code into bytecode so that it may run on a Java Virtual Machine. As for the interface, it is written in Java and uses the standard Swing library. If an error is detected while analyzing, whether it is a lexical, syntactic, semantic or flow error, the analyzer stops and displays a message explaining the cause of the error. If the analyzer infers that the code needs to be instrumented, it automatically generates and displays the instrumented code. If no error occurs and there is no need for instrumentation, then a message of correctness is displayed.

Examples A few examples of the whole approach are presented in the following figures. The figures show the returned environment G, the returned command number i as well as the input *pc*, the security level of the context. Recall that the identifiers *privateChannel*, *publicChannel*, *highValue* and *lowValue* are predefined constants. The result of the analysis, including instrumentation when necessary, is shown in the lower part of the figures.

The program of Figure 2.6 is rejected. The security level of the value variable x is H because its value is assigned inside the context of *highValue*, which is of type H. There is an attempt to send x on a public channel, which make the program be rejected.

Input to analyzer	Inference analysis		
	Environment	pc	i
if *highValue*		$pc_{if} = H$	2
then x := *lowValue*	$G(2) = [_instr \mapsto L, x \mapsto H\ val]$	H	3
else skip	$G(3) = [_instr \mapsto L]$	H	4
end;	$G(1) = [_instr \mapsto L, x \mapsto H\ val]$	H	4
send x **to** *publicChannel*	**fail** since $H \not\sqsubseteq L$		
Output : Error (Send) : Cannot send x (H) to publicChannel (L).			

Figure 2.6: Implicit flow

The program in Figure 2.7 is similar to the one in Figure 2.1 except that the context in which c is defined is now L instead of H. For this reason, it is not necessary to block channel c.

Since c can either be a public or private channel (depending on the value of *lowValue*), it is marked as unknown. A call for instrumentation results from the first send, to ensure that *highValue* is only sent to a private channel.

<table>
<tr><th>Input to analyzer</th><th colspan="3">Inference analysis</th></tr>
<tr><td rowspan="6">if lowValue
  then $c := publicChannel$
  else $c := privateChannel$
end;
send highValue to c</td><td>Environment</td><td>pc</td><td>i</td></tr>
<tr><td>$G(1) = [_\ instr \mapsto L]$</td><td>$pc_{if} = L$</td><td>2</td></tr>
<tr><td>$G(2) = [_\ instr \mapsto L, c \mapsto L\ chan]$</td><td>$L$</td><td>3</td></tr>
<tr><td>$G(3) = [_\ instr \mapsto L, c \mapsto H\ chan]$</td><td>$L$</td><td>4</td></tr>
<tr><td>$G(1) = [_\ instr \mapsto L, c \mapsto U\ chan]$</td><td>$L$</td><td>4</td></tr>
<tr><td>$G(4) = [_\ instr \mapsto U, c \mapsto U\ chan]$</td><td>$L$</td><td>5</td></tr>
<tr><td colspan="4">

```
Output : push(sup(top(pc), evalT(TypeOf_Expression(lowValue))), pc);
         if lowValue then
            c := publicChannel;
            updateEnv(g_M, c, G(2)(c));
         else
            c := privateChannel;
            updateEnv(g_M, c, G(3)(c));
         end;
         pop(pc);
         tau = TypeOf_Expression(c);
         tau1 = TypeOf_Expression(highValue);
         if(((tau = Lchan) and (sup(evalT(tau1), top(pc)) = H)) or (tau = B chan))
         then fail;
         else send highValue to c;
```

</td></tr>
</table>

Figure 2.7: The send of a high value on an unknown channel calls for instrumentation

The example presented in Figure 2.8 shows how the instrumentation algorithm works. The inference algorithm determines that the program needs instrumentation. The program is shown on the upper left corner of the figure. The instrumentation result is shown in the lower part of the figure. The third instruction receives a channel name on another one. The instrumentation is necessary to obtain the real type of this channel. In the sixth instruction of the instrumented program, the update of $G(3)$ is due to the fact that the inference algorithm marks the channel c as unknown on that line. The fourth instruction is a **send** command. A check is inserted in the instrumented code to ensure that a secret information is neither sent on a public channel (the type of c being unknown statically) nor on a blocked one (B).

A more "realistic" example is described in [11]: one may want to "prohibit a personal finance program from transmitting credit card information over the Internet even though the program needs Internet access to download stock market reports. To prevent the finance program from illicitly transmitting the private information (perhaps cleverly encoded), the compiler checks that the information flows in the program are admissible." This could be translated into the code of Figure 2.9 where all the channels, except *internet*, are private.

2.8 Related Work

Securing flow information has been the focus of active research since the seventies. Dynamic techniques were the first methods as in [12]. Denning and Denning [13] introduce for the

Input to analyzer	Inference analysis		
	Environment	pc	i

receive$_{\mathbf{c}}$ v **from** $privateChannel$;
if $lowValue$ **then**
  receive$_{\mathbf{n}}$ c **from** $publicChannel$;
  send v **to** c
else skip
end
	$G(1) = [_\ instr \mapsto L, v \mapsto H\ val]$	L	2

		$pc_{if} = L$	3
	$G(3) = [_\ instr \mapsto L, v \mapsto H\ val, c \mapsto U\ chan]$	L	4
	$G(4) = [_\ instr \mapsto U, v \mapsto H\ val, c \mapsto U\ chan]$	L	5
	$G(5) = [_\ instr \mapsto L, v \mapsto H\ val]$	L	6
	$G(2) = [_\ instr \mapsto U, v \mapsto H\ val, c \mapsto U\ chan]$	L	6

Output :	**receive$_{\mathbf{c}}$** v **from** $privateChannel$;
	updateEnv($g_M, v, G(1)(v)$);
	push(sup(top(pc), evalT(TypeOf_Expression($lowValue$))), pc);
	if $lowValue$ **then**
	  receive$_{\mathbf{n}}$ c **from** $publicChannel$;
	  updateEnv($G(3), c$, TypeOf_Channel(c));
	  updateEnv($g_M, c, G(3)(c)$);
	  tau = TypeOf_Expression(c);
	  $tau1$ = TypeOf_Expression(v);
	  if((($tau = L\ chan$) and (sup(evalT($tau1$), top(pc)) $= H$))
	   or ($tau = B\ chan$))
	   then fail;
	   else send v **to** c;
	else skip;
	end;
	pop(pc);		

Figure 2.8: The **send** of a high value on an unknown channel calls for instrumentation

Input	**receive$_{\mathbf{c}}$** $stockMarketReports$ **from** $internet$; **send** $stockMarketReports$ **to** $screen$; **receive$_{\mathbf{c}}$** $creditCardNumber$ **from** $settings$; **send** $creditCardNumber$ **to** $secureLinkToBank$; **receive$_{\mathbf{c}}$** $latestTransactions$ **from** $secureLinkToBank$; **send** $latestTransactions$ **to** $screen$; $cleverlyEncodedCreditCardNumber := creditCardNumber * 3 + 2121311611218191$; **send** $cleverlyEncodedCreditCardNumber$ **to** $internet$
Output	Error (Send) : Cannot send $cleverlyEncodedCreditCardNumber$ (H) to $internet$ (L).

Figure 2.9: A more realistic example

first time, secure information-flow by static analysis, based on control and data flow analysis. Subsequently, many approaches have been devised using type systems. They vary in the type of language, its expressivness and the property being enforced. Volpano and Smith in [2] introduce a type based analysis for an imperative language. Pottier and Simonet in [14] analyse the functional language ML, with references, exceptions and polymorphism. Banerjee and Naumann devise a type based analysis for a Java-like language. Their analysis however has some trade-offs like low security guards for conditionals that involve recursive calls. In [15], Myers statically enforces information policies in JFlow, an extension of Java that adds security levels annotations to variables. Barthe et al. [16] investigate logical-formulation of noninterference, enabling the use of theorem proving or model-checking techniques. Nevertheless, purely static approaches are too conservative and suffer from a large number of false positive. In fact some information need to take an accurate decision are often only available during execution. This has cause a revival of interest for dynamic analysis. Russo and

Sabelfeld in [17], prove that dynamic analyses could enforce the same security policy enforced by most static analyses, termination-insensitive noninterference and even be more permissive (with less false-positive). This is true for flow insensitive analyses but not for flow sensitive ones. In [4], Russo and Sabelfeld show the impossibility for a sound purely dynamic monitor to accept the same set of programs accepted by the classic flow sensitive analysis [18] of Hunt and Sands. Russo and Sabelfeld in [4] present a monitor that uses static analysis during execution. In [19], the authors present an interesting approach to noninterference based on abstract interpretation.

Our approach is flow sensitive, similarly to [18]. However, it distinguishes between variables in live memory and channels. We argue that our approach lead to less false positive and to lighter executions than existing approaches.

2.9 Conclusion

Ensuring secure information flow within sensitive systems has been studied extensively. In general, the key idea in type-based approaches is that if a program is well typed, then it is secure according to the given security properties.

We define a type system that captures lack of information in a program at compile-time. Our type system is flow sensitive, variables are assigned the security levels of their stored values. We clearly distinguish between variables and channels through which the program communicates, which is more realistic.

Our main contribution is the handling of a multi-valued security typing. The program is considered well typed, ill typed or uncertain. In the first case, the program can safely be executed, in the second case the program is rejected and need modifications, while in the third case instrumentation is to be used in order to guarantee the satisfaction of noninterference. This approach allows to eliminate false positives due to conservative static analysis approximations and to introduce run-time overhead only when it is necessary. We obtain fewer false positives than purely static approaches because we send some usually rejected programs to instrumentation.

2.10 Bibliography

[1] J. Desharnais, E. P. Kanyabwero, and N. Tawbi, "Enforcing information flow policies by a three-valued analysis," in *Proceedings of the 6th international conference on Mathematical Methods, Models and Architectures for Computer Network Security: computer network security*, ser. MMM-ACNS'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 114–129. Available: http://dx.doi.org/10.1007/978-3-642-33704-8_11

[2] D. Volpano, C. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2-3, pp. 167–187, Jan. 1996. Available: http://dl.acm.org/citation.cfm?id=353629.353648

[3] S. Hunt and D. Sands, "On flow-sensitive security types," in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 79–90. Available: http://doi.acm.org/10.1145/1111037.1111045

[4] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *Proceedings of the IEEE Computer Security Foundations Symposium*, 2010.

[5] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, pp. 236–243, May 1976. Available: http://doi.acm.org/10.1145/360051.360056

[6] A. Bedford, J. Desharnais, T. G. Godonou, and N. Tawbi, "Hybrid flow analysis implementation," https://github.com/andrew-bedford/ulsifa, 2013, last Accessed on October 2018.

[7] G. Smith, "Principles of secure information flow analysis," in *Malware Detection*. Springer, 2007, vol. 27, pp. 291–307.

[8] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *Proceedings of the IEEE Computer Security Foundations Workshop*, Jul. 2006.

[9] N. Kobayashi, "Type-based information flow analysis for the pi-calculus," *Acta Informatica*, vol. 42, no. 4-5, pp. 291–347, 2005.

[10] J. Desharnais, E. P. Kanyabwero, and N. Tawbi, "Enforcing information flow policies by a three-valued analysis, long version," http://www.ift.ulaval.ca/fileadmin/ift/Nadia_Tawbi/PDF/MMMACNS2012articleLong.pdf, 2012, last Accessed on October 2018.

[11] S. Zdancewic and A. C. Myers, "Secure information flow via linear continuations," *Higher Order and Symbolic Computation*, vol. 15, p. 2002, 2002.

[12] J. S. Fenton, "Memoryless subsystems," *Comput. J.*, vol. 17, no. 2, pp. 143–147, 1974.

[13] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, pp. 504–513, July 1977. Available: http://doi.acm.org/10.1145/359636.359712

[14] F. Pottier and V. Simonet, "Information flow inference for ML," in *Proceedings of the The ACM Symposium on Principles of Programming Languages*, 2002.

[15] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the ACM Symposium on Principles of Programming Languages*, 1999.

[16] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Proceedings of the IEEE workshop on Computer Security Foundations*, 2004. Available: http://portal.acm.org/citation.cfm?id=1009380.1009669

[17] A. Sabelfeld and A. Russo, "From dynamic to static and back: Riding the roller coaster of information-flow control research," in *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, 2009, pp. 352–365. Available: https://doi.org/10.1007/978-3-642-11486-1_30

[18] S. Hunt and D. Sands, "On flow-sensitive security types," *SIGPLAN Not.*, vol. 41, no. 1, pp. 79–90, Jan. 2006. Available: http://doi.acm.org/10.1145/1111320.1111045

[19] R. Giacobazzi and I. Mastroeni, "Abstract non-interference: parameterizing non-interference by abstract interpretation," *SIGPLAN Not.*, vol. 39, no. 1, pp. 186–197, Jan. 2004. Available: http://doi.acm.org/10.1145/982962.964017

Chapter 3

A Progress-Sensitive Flow-Sensitive Inlined Information-Flow Control Monitor

Authors: Andrew Bedford, Stephen Chong, Josée Desharnais, Elisavet Kozyri and Nadia Tawbi

Journal: *Computers & Security*

Status: peer reviewed; published[1]; presented

Year: 2017

3.1 Résumé

Nous présentons dans ce chapitre un moniteur hybride de flots d'information qui est sensible au progrès (c'est-à-dire, les fuites par progrès sont détectées) et au flot (c'est-à-dire, l'ordre des commandes est pris en compte). La méthode utilisée par notre moniteur hybride consiste en trois étapes : (1) à l'aide d'analyse statique, vérifier que le programme ne contient pas de fuites d'information évidentes; (2) instrumenter l'application (c'est-à-dire, insérer des commandes) pour prévenir les fuites d'information confidentielles à l'exécution; (3) évaluer partiellement le programme pour diminuer l'impact de l'instrumentation sur le temps d'exécution. Une particularité de notre moniteur est qu'il utilise des ensembles de niveaux comme étiquettes afin de traquer les niveaux d'information qui peuvent avoir influencés la valeur d'une variable. Nous illustrons notre approche sur un langage impératif simple.

[1]The published version is available at ScienceDirect via https://doi.org/10.1016/j.cose.2017.04.001

3.2 Abstract

We present a novel progress-sensitive, flow-sensitive hybrid information-flow control monitor for an imperative interactive language. Progress-sensitive information-flow control is a strong information security guarantee which ensures that a program's progress (or lack thereof) does not leak information. Flow-sensitivity means that this strong security guarantee is enforced fairly precisely: our monitor tracks information flow per variable and per program point. We illustrate our approach on an imperative interactive language.

Our hybrid monitor is inlined: source programs are translated, by a type-based analysis, into a target language that supports dynamic security levels. A key benefit of this is that the resulting monitored program is amenable to standard optimization techniques such as partial evaluation. One of the distinguishing features of our hybrid monitor is that it uses sets of levels to track the different *possible* security types of variables. This feature allows us to distinguish outputs that *never* leak information from those that *may* leak information.

3.3 Introduction

We increasingly rely on computer systems to safeguard confidential information, and maintain the integrity of critical data and operations. But in our highly interconnected world, these trusted systems often need to communicate with untrusted parties. Trusted systems risk leaking confidential information to untrusted parties, or allowing input from untrusted parties to corrupt data or the operation of the trusted system.

Information-flow control is a promising approach to enable trusted systems to interact with untrusted parties, providing fine-grained application-specific control of confidential and untrusted information. Static mechanisms for information-flow control (such as security type systems [1, 2]) analyze a program before execution to determine whether the program's execution will satisfy the appropriate information flow requirements. This has low runtime overhead, but can generate many false positives. Hybrid mechanisms ([3]) have been proposed to eliminate some of these false positives. They combine static analysis with runtime monitoring in order to increase the precision of the analysis at the expense of higher runtime overhead. Compared to hybrid mechanisms, purely dynamic mechanisms ([4]), which enforce information flow policies only by monitoring execution and without using any static analysis, are either unsound or less permissive [5].

In this work, we enforce confidentiality policies using a novel hybrid information-flow control monitor for an imperative interactive language. It is an extension of the work presented at IFIP SEC 2016 [6]. The key features of our monitor are as follows.

Our monitor is *progress-sensitive* [7]: it prevents confidential information from being leaked via progress channels. Information leaks through a progress channel when a program's progress

(or lack thereof) depends on confidential information and is observable by an adversary. It is a generalization of termination-sensitive information security to interactive systems (i.e., systems that interact with an external environment at runtime).

Our monitor is *flow-sensitive*: the security level associated with program variables may change during the execution. Flow sensitivity increases the precision of the monitor, meaning that it is able to accept more programs.

Our language has *channel-valued variables*: communication channels are constants that can be assigned to program variables. This language feature allows realistic communication scenarios to be modelled in our language (e.g., where the same code may communicate with users of arbitrary security levels). Most previous work on language-based information-flow control require that the channel used for an input or output operation be statically known, and allow only a single communication channel per security level.

Our monitor is *inlined*: source programs are translated into a target language that supports dynamic security levels [8]. The type-based translation inserts commands to track the security levels of program variables and contexts, and to control information flow. A key benefit of inlining the monitor is that the resulting monitored program in the target language is amenable to standard optimization techniques such as partial evaluation [9].

Our monitor is *hybrid*: it uses both dynamic and static enforcement techniques. The translation to the target language performs a static analysis. If the program is statically determined to be insecure, then the program is statically rejected. Otherwise, the translation of the program dynamically tracks the statically unknown security levels of variables, and ensures that no leak occurs at runtime.

Our main contributions are as follows.

- We present an extended version of the hybrid monitor first presented in our previous article [6]. The extension consists in generalizing the flow- and progress-sensitive enforcement to general lattices.
- We use sets of levels during static analysis to represent the possible levels that can be associated with a variable during execution. When general lattices are enforced, sets of levels are a more accurate representation than the *unknown level* U introduced in our previous article [6]. Thus, the distinction that our static analysis makes between outputs that *never* leak information and outputs that *may* leak information is more precise when using sets of levels. This more precise distinction leads to fewer inlined commands, and thus to less runtime overhead and increased permissiveness of the monitor.
- Previous work treats channels and regular variables differently, by associating distinct type structures during analysis. Our approach demonstrates that modeling the sensitivity of the information conveyed by these two entities leads to a striking similar type

structure.

- We present two additional ideas to increase the precision of the static analysis and the permissiveness of the dynamic analysis: propagating constraints on the set of possible security levels and using conditional updates. We plan on integrating these ideas into our future mechanisms.
- We prove that our inlined monitor is sound and that the semantics of the original program is preserved, as long as the program is secure.

3.3.1 Examples

We present several examples of programs in our source language, to both provide background on information-flow control, and highlight some of the features of our hybrid monitor. For simplicity, we assume that the variables `lowValue`, `medValue`, `highValue`, `lowChannel`, `medChannel` and `highChannel` exist, have arbitrary values and have the suggested security levels (L for low, M for medium and H for high).

Explicit and implicit flows An *insecure explicit information flow* occurs when a confidential value is output to a public channel. An *insecure implicit information flow* [10] occurs when the decision to perform output on a public channel depends on confidential information. This violates security because an observer of the public channel will see whether the output occurred, and might thus learn confidential information. Techniques for tracking and controlling implicit and explicit information flow at the language level are well known [1, 2, 5], and are used in this work. The following program exhibits both insecure explicit and insecure implicit information flow. Our approach will reject this program statically.

```
(* insecure explicit flow *)
send highValue to lowChannel;
if highValue > 0 then
  (* insecure implicit flow *)
  send 42 to lowChannel
end
```

Unknown security levels Our source language supports variables whose security level could be statically unknown. Consider the following program, where the output may or may not be secure, depending on the value of `lowValue`.

```
if lowValue > 0
 then c := highChannel
 else c := lowChannel
end;
send highValue to c
```

Listing 3.1: Statically uncertain channel level

Purely static mechanisms would reject this program entirely, and indeed, to the best of our knowledge, all previous work either cannot express this program or statically reject it. This is because it would be unsound to statically treat `c` as being a private channel, since that might incorrectly allow private values to be sent to public channel `lowChannel`. Similarly, it would be unsound to statically treat `c` as a public channel, since that might incorrectly allow values read from private channel `highChannel` to be treated as public values. By contrast, our hybrid approach recognizes that the security of this program depends on the runtime value of `c`, and instruments it to track whether `c` refers to a high-security channel or a low-security channel, in order to intervene only in the latter case. We use sets of levels in our security types in order to track the different possibilities during our static analysis (e.g., `c` is {L,H} after the conditional).

To push the permissiveness a little more, we treat value variables similarly as channel variables. Take Listing 3.2 for example.

```
if lowValue > 0
  then x := lowValue  (*x is L*)
  else x := highValue (*x is H*)
end;                  (*x is {L,H}*)
send x to lowChannel
```

Listing 3.2: Statically uncertain variable level

Traditionally, the level of variable `x` after the **if** command would be statically approximated as being high. Such an approximation would cause the program to be statically rejected at the **send** command. By tracking the possible levels of variables, we can detect that a safe executions can happen and choose to instrument the program instead.

Progress channels The progress of a program, which is observable through its outputs, can also reveal information. For example, in the following program, whether or not the output on the low-security channel occurs depends on whether the preceding loop terminates, which in turn depends on confidential information.

```
while highValue > 0 do
  skip
end; (*loop may diverge*)
send 42 to lowChannel
```

Listing 3.3: Progress leak

Although this example leaks only 1 bit of information, progress channels can be used to leak a significant amount of information [7]. The most common way to prevent leaks through progress channels is to forbid loops whose execution depends on confidential information [11, 12], but it leads to the rejection of many secure programs, such as the following.

```
while highValue > 0 do
  highValue := highValue - 1
end; (*loop always terminates*)
send 42 to lowChannel
```

Listing 3.4: Loop that always terminates

To accept such programs, we follow Moore et al. [13] and use an oracle (conservative and assumed correct) to statically determine the termination behavior of loops. If the oracle determines that a loop always terminates (like the one in Listing 3.4), then we know that no following output could leak information through progress channels. On the other hand, if the oracle says that it may diverge (like the one in Listing 3.3), then we must take into account the fact that an output following the loop's execution could leak information.

In our approach, the oracle is a parameter and is based on termination analysis methods brought from the literature such as the one described in Cook et al. [14]

The inlined monitor itself may introduce leaks of confidential values through progress channels. The decision that a monitor makes to allow or not an execution, may depend on confidential values. Thus, outputs following this decision could leak these confidential values. Consider the following example:

```
if highValue > 0
   then skip
   else send highValue to c
end;
send 42 to lowChannel
```

Listing 3.5: Leaking through monitor decision

If during execution `c` is `lowChannel`, then the decision to allow this **send** leaks the value of the condition, `highValue > 0`, to `lowChannel`.

The danger of a monitor to leak confidential values through progress channels becomes greater when the monitor enforces confidentiality policies on a multilevel lattice. Consider the following example, taken from Kozyri et al. [15].

```
if medValue > 0
   then x := highValue  (*x is H *)
   else x := lowValue   (*x is L *)
end;
send x to c
send 1 to lowChannel
```

Listing 3.6: Variable level sensitivity

Suppose that the unknown channel `c` of the first **send** command happens to be of medium security level at runtime. If the monitor does not halt the execution of the first **send** command, then the second **send** command would be reached. Then, if the output to `lowChannel` occurred, it would leak information about `medValue`.

To prevent such a leak, we combine our previous work [6] with the dynamic mechanism of Kozyri et al. [15] and keep track of the level of information that could be leaked through progress channels. More specifically, for Listing 3.5, we detect that one of the branch of the **if** contains a **send** command that may be blocked during runtime and update the level of information that could be leaked through progress channels to H (i.e., the level of the condition), hence preventing future outputs to channels that are lower than H. Similarly, for Listing 3.6, we detect that the first **send** command may be blocked during execution and that the value of its parameters (`x` and `c`) depends of information of level M. For this reason, the level of information that could be leaked through progress channels is updated to M after that first output, hence preventing future outputs to channels that are lower than M.

3.3.2 Structure

In Section 3.4, we present the imperative language used to illustrate our approach. Section 3.5 defines the noninterference property. Section 3.6 describes our typed-based instrumentation mechanism, explains the type system, and presents the target language in which the instrumented programs are written; it is an extension of the source language with dynamic security levels. Section 3.7 proves that the instrumented programs are noninterferent. Section 3.8 presents two ways to increase the precision and permissiveness of our monitor. Section 3.9 is a summary of related work. Finally, we conclude in Section 3.10.

3.4 Source Language

Source programs are written in a simple imperative language with commands for receiving and sending information.

3.4.1 Syntax

Let $\mathcal{V}$ be a set of identifiers for variables, and $\mathcal{C}$ a set of predefined communication channels. The syntax is as follows.

$$
\begin{array}{llll}
\textit{(variables)} & x & \in & \mathcal{V} \cup \mathcal{C} \\
\textit{(integer constants)} & n & \in & \mathbb{Z} \\
\textit{(expressions)} & e & ::= & x \mid n \mid e_1 \textbf{ op } e_2 \mid \textbf{read } x \\
\textit{(commands)} & cmd & ::= & \\
\multicolumn{4}{l}{\quad \textbf{skip} \mid x := e \mid \textbf{if } e \textbf{ then } cmd_1 \textbf{ else } cmd_2 \textbf{ end} \mid} \\
\multicolumn{4}{l}{\quad \textbf{while } e \textbf{ do } cmd \textbf{ end} \mid cmd_1; cmd_2 \mid \textbf{send } x_1 \textbf{ to } x_2}
\end{array}
$$

Values are integers (we use zero for false and nonzero for true), or channel names. The symbol **op** stands for arithmetic or logic binary operators on integers.

We suppose that the interaction of a program with its environment (which can be a user or another program) is done through channels. Channels can be, for example, files, users, network channels, keyboards, computer screens, etc. Without loss of generality, we consider that each channel consists of one value. The expression **read** x returns the current value in channel x (without modifying the channel's content). Command **send** x_1 **to** x_2 sends the value of variable x_1 to channel x_2 and overwrites the current value in the channel. In other words, it outputs the value of x_1 to channel x_2. The security levels of these channels are designated in advance by some security administrator.

Note that in order to keep the syntax simple, we chose to enforce certain constraints in the semantics and type system rather than in the syntax. For example, it is the semantics and type system that verify that x_1 is an integer variable and x_2 is a channel during the execution of a **send** command.

3.4.2 Semantics

A memory $m : \mathcal{V} \uplus \mathcal{C} \to \mathbb{Z} \uplus \mathcal{C}$ is a partial map from variables and channels to values, where the value of a channel is the last value sent to this channel. More precisely a memory is the disjoint union of two (partial) maps of the following form:

$$m_v : \mathcal{V} \to \mathbb{Z} \uplus \mathcal{C}, \qquad m_c : \mathcal{C} \to \mathbb{Z},$$

where $\uplus$ stands for the disjoint union operator. We omit the subscript whenever the context is clear. We write $m(e) = r$ to indicate that the evaluation of expression e under memory m returns r.

The semantics of the source language is mostly standard and is illustrated in Figure 3.1. Program configurations are tuples $\langle cmd, m, o \rangle$ where cmd is the command to be evaluated, m is the current memory and o is the current output trace. A transition between two configurations is denoted by the $\longrightarrow$ symbol. We write $\longrightarrow^*$ for the reflexive transitive closure of the $\longrightarrow$ relation.

We write $v :: vs$ for sequences where v is the first element of the sequence, and vs is the rest of the sequence. We write ϵ for the empty sequence. An output trace is a sequence of output events: it is of the form $o = (v_0, ch_0) :: (v_1, ch_1) :: \ldots$ where $v_k \in \mathbb{Z}$ is an integer value, and $ch_k \in \mathcal{C}$ is a channel, $k \in \mathbb{N}$. The rule for sending a value appends a new output event to the end of the trace. We abuse notation and write $o :: (v, ch)$ to indicate event (v, ch) appended to trace o.

We write $\langle cmd, m, \epsilon \rangle \downarrow o$ if execution of configuration $\langle cmd, m, \epsilon \rangle$ can produce trace o, where o may be finite or infinite. For finite o, $\langle cmd, m, \epsilon \rangle \downarrow o$ holds if there is a configuration

$$\text{(Skip)}\ \langle \mathbf{skip}, m, o\rangle \longrightarrow \langle \mathbf{stop}, m, o\rangle$$

$$\text{(Assign-Ch)}\ \frac{e \in \mathcal{C}}{\langle x := e, m, o\rangle \longrightarrow \langle \mathbf{stop}, m[x \mapsto e], o\rangle}$$

$$\text{(Assign)}\ \frac{e \notin \mathcal{C} \qquad m(e) = r}{\langle x := e, m, o\rangle \longrightarrow \langle \mathbf{stop}, m[x \mapsto r], o\rangle}$$

$$\text{(Send)}\ \frac{m(x_1) = v \in \mathbb{Z} \qquad (ch = x_2 \in \mathcal{C} \vee ch = m(x_2) \in \mathcal{C})}{\langle \mathbf{send}\ x_1\ \mathbf{to}\ x_2, m, o\rangle \longrightarrow \langle \mathbf{stop}, m[ch \mapsto v], o :: (v, ch)\rangle}$$

$$\text{(Seq1)}\ \frac{\langle cmd_1, m, o\rangle \longrightarrow \langle \mathbf{stop}, m', o'\rangle}{\langle cmd_1; cmd_2, m, o\rangle \longrightarrow \langle cmd_2, m', o'\rangle}$$

$$\text{(Seq2)}\ \frac{\langle cmd_1, m, o\rangle \longrightarrow \langle cmd_1', m', o'\rangle \qquad cmd_1' \neq \mathbf{stop}}{\langle cmd_1; cmd_2, m, o\rangle \longrightarrow \langle cmd_1'; cmd_2, m', o'\rangle}$$

$$\text{(If)}\ \frac{m(e) \neq 0 \Longrightarrow i = 1 \qquad m(e) = 0 \Longrightarrow i = 2}{\langle \mathbf{if}\ e\ \mathbf{then}\ cmd_1\ \mathbf{else}\ cmd_2\ \mathbf{end}, m, o\rangle \longrightarrow \langle cmd_i, m, o\rangle}$$

$$\text{(Loop1)}\ \frac{m(e) \neq 0}{\langle \mathbf{while}\, e\, \mathbf{do}\, cmd\, \mathbf{end}, m, o\rangle \longrightarrow \langle cmd; \mathbf{while}\, e\, \mathbf{do}\, cmd\, \mathbf{end}, m, o\rangle}$$

$$\text{(Loop2)}\ \frac{m(e) = 0}{\langle \mathbf{while}\ e\ \mathbf{do}\ cmd\ \mathbf{end}, m, o\rangle \longrightarrow \langle \mathbf{stop}, m, o\rangle}$$

Figure 3.1: Semantics of the source language

$\langle cmd', m', o\rangle$ such that $\langle cmd, m, \epsilon\rangle \longrightarrow^* \langle cmd', m', o\rangle$. For infinite o, $\langle cmd, m, \epsilon\rangle \downarrow o$ holds if for all traces o' such that o' is a finite prefix of o, we have $\langle cmd, m, \epsilon\rangle \downarrow o'$.

3.5 Security

For our purposes, we assume a finite lattice of security levels $(\mathcal{L}, \sqsubseteq)$ which contains at least two elements: L for the bottom of the lattice and H for the top of the lattice, i.e. $\forall \ell \in \mathcal{L}, L \sqsubseteq \ell \wedge \ell \sqsubseteq H$. We define an execution as ℓ-secure if the outputs to channels of level ℓ or lower do not reveal any information about the inputs of channels that are not lower than or equal to ℓ. This is a standard form of noninterference (e.g., [1, 2]) adapted for our particular language model.

Before formally defining noninterference, we first introduce some helpful technical concepts. The *projection of trace o to security level ℓ*, written $o \upharpoonright \ell$, is its restriction to output events whose channels' security levels are less than or equal to ℓ. Formally,

$$\epsilon \upharpoonright \ell = \epsilon$$

$$((v, ch) :: o) \upharpoonright \ell = \begin{cases} (v, ch) :: (o \upharpoonright \ell) & \text{if } \mathit{levelOfChan}(ch) \sqsubseteq \ell \\ o \upharpoonright \ell & \text{otherwise} \end{cases}$$

where $\mathit{levelOfChan}(ch)$ denotes the security level of channel ch (typically specified by the administrator).

We say that two memories m and m' are ℓ-equivalent if they agree on the content of variables (including channel variables) whose security levels are ℓ or lower.

Definition 3.1 (Progress-sensitive noninterference)**.** *We say that a program p satisfies* progress-sensitive noninterference *if for any $\ell \in \mathcal{L}$, and for any two memories m and m' that are ℓ-equivalent, and for any trace o such that $\langle p, m, \epsilon \rangle \downarrow o$, then there is some trace o', such that $\langle p, m', \epsilon \rangle \downarrow o'$ and $o \upharpoonright \ell = o' \upharpoonright \ell$.*

This definition of noninterference is progress-sensitive in that it assumes that an observer can distinguish an execution that will not produce any additional observable output (due to termination or divergence) from an execution that will make progress and produce additional observable output. Progress-insensitive definitions of noninterference typically weaken the requirement that $o \upharpoonright \ell = o' \upharpoonright \ell$ to instead require that $o \upharpoonright \ell$ is a prefix of $o' \upharpoonright \ell$, or vice versa.

Among the previously presented examples, only Listing 3.4 satisfies progress-sensitive noninterference. Nevertheless, we statically accept the programs of Listings 3.1, 3.2, 3.5 and 3.6, since we transform them into programs that satisfy progress-sensitive noninterference.

3.6 Type-Based Instrumentation

We enforce noninterference by translating a source program to a target program that tracks the security levels of its variables and checks the security of output commands. The translation performs a type-based static analysis of the source program, and rejects programs that contain outputs whose executions will always be blocked by our monitor (i.e., the translation fails).

In this section, we first present the security types for the source language followed by the description of the target language, which extends the source language with runtime representation of security levels. We then present the translation from the source language to the target language.

3.6.1 Source Language Types

Source language types are defined according to the following grammar.

$$\begin{array}{ll} \textit{(security labels, Lab)} & \ell ::= \mathcal{P}(\mathcal{L}) \setminus \{\emptyset\} \\ \textit{(value types, ValT)} & \sigma ::= \mathit{int}_\ell \mid \mathit{int}_\ell\ \mathit{chan} \\ \textit{(variable types, VarT)} & \tau ::= \sigma_\ell \end{array}$$

Security labels are non-empty sets of security levels. They represent the possible security levels of a variable at runtime. If a security label contains more than one element, it means that its security level is statically unknown (see Listings 3.1 and 3.2 on pages 44 and 45 for examples).

Value types are the types of integers and channels. Type int_ℓ is the type of integers whose values are of security level ℓ, and type $\mathit{int}_\ell\ \mathit{chan}$ is the type of a channel whose values are of security level ℓ.

Variable types associate a security level with a value type. Intuitively, $\sigma_{\ell'}$ represents the type of a variable whose value type is σ, and whose variable type is ℓ'. The latter is an upper bound of the information level influencing the value of the variable. When a variable type ℓ' is associated with a value type ℓ for a channel, it means that the sensitivity of the content of the channel is ℓ, and the sensitivity of the channel itself is ℓ'. This is the same approach that was used by Bedford et al. [6].

For example, consider the program in Listing 3.1, page 44. Immediately following the conditional command, the type system gives variable `c` the type $(\mathit{int}_{\{L,H\}}\ \mathit{chan})_{\{L\}}$. This type reflects that only low information determines which channel is assigned to `c` (i.e., variable `lowValue` determines `c`'s value), and whether `c` is a low channel or a high channel is statically unknown.

Similarly, when a variable type ℓ' is associated with a value type ℓ for an integer, it means that the sensitivity of the integer is ℓ, and the sensitivity of ℓ is ℓ'. This is the same approach that was used by Kozyri et al. [15] to prevent leakage through guarded sends.

For example, consider the program in Listing 3.6, page 46. Immediately following the conditional command, the type system gives variable `x` the type $(\mathit{int}_{\{L,H\}})_{\{M\}}$. This type reflects that an information of medium level determines which value is assigned to `x` (i.e., variable `medValue` determines `x`'s value), and that the information contained within `x` is either of low security level or of high security level. It is necessary to keep track of the context level in which `x` has been assigned its value. This information is used to halt the execution of the second **send** and prevent observers from deducing that `medValue` is less or equal to 0.

3.6.2 Sets of Levels

Note that we use sets of levels not only to increase the precision of the analysis, but also because we *have to* due to our use of channel variables. Indeed, one of the issues that we encountered is the fact that we cannot conservatively approximate the level of a channel variable, due to the fact that we both read and write on channels. Listing 3.7 illustrates why.

```
if lowValue > 0 then
  c := lowChannel
else
  c := highChannel
end
send highValue to c (*Pessimist: c is L*)
x := read c (*Pessimist: c is H*)
send x to lowChannel
```

Listing 3.7: We cannot be pessimistic about channel variables

After the conditionals, `c` has type $(int_{\{L,H\}} chan)_{\{L\}}$ because it contains either a low or high channel and its value is assigned in a context of level L. Our typing system accepts this program, but makes sure that a runtime checks are inserted. If the condition `lowValue > 0` happens to be true at runtime, then the execution of the program will be stopped at the first send command thanks to the inserted runtime checks. Similarly, if the condition is false, then the execution of the program will be stopped at the second send. The uncertainty is unavoidable in the presence of flow sensitivity and channel variables. Indeed, we point out that we cannot be pessimistic about the level of `c` in this program. The output command suggests that a safe approximation for `c` would be a low security level. Yet, the input command suggests that a safe approximation for `c` would be a *high* security level, which contradicts the previous observation.

Consequently, in order to accept the program in Listing 3.7, we chose to use sets of security levels. As a consequence, we will obtain fewer false positives as we do not consider the worst possible case in our analysis, we leave it to the execution to check whether the information flow turns out to be secure or not.

We derive two relations that allow us to compare the sets of possible levels.

Definition 3.2. *The relations* $\sqsubseteq_s$, surely less than, *and* $\sqsubseteq_m$, maybe less than, *are defined as follows*

$$\begin{aligned} \ell_1 \sqsubseteq_s \ell_2 &= (\forall e_1 \in \ell_1, e_2 \in \ell_2.\ e_1 \sqsubseteq e_2). \\ \ell_1 \sqsubseteq_m \ell_2 &= (\exists e_1 \in \ell_1, e_2 \in \ell_2.\ e_1 \sqsubseteq e_2). \end{aligned}$$

Intuitively, we have $\ell \sqsubseteq_s \ell'$ when we can be sure statically that $\ell \sqsubseteq \ell'$ will be true at runtime, and we have $\ell \sqsubseteq_m \ell'$ when it is possible that $\ell \sqsubseteq \ell'$ at runtime.

Definition 3.3. *The supremum on sets of levels is defined as follows*

$$\ell_1 \sqcup \ell_2 = \{e_1 \sqcup e_2 \mid e_1 \in \ell_1, e_2 \in \ell_2\}.$$

Recall that sets of levels represent the possible security levels that can occur. Hence a supremum between specific levels must be mimicked, between sets of levels, by the set of all possible results of such suprema. Consequently, we call it a supremum even if the corresponding preorder relation is not used here.

We instrument source programs to track at runtime the security levels that are statically unknown. In order to track these security levels, our target language allows their runtime representation.

3.6.3 Syntax and Semantics of the Target Language

Our target language is inspired by the work of Zheng and Myers [8], which introduced a language with first-class security levels, and a type system that soundly enforces noninterference in this language. The syntax of our target language is defined as follows:

(variables)	x	$\in$	$\mathcal{V} \cup \mathcal{C}$
(level variables)	$\tilde{x}$	$\in$	$\mathcal{V}_{level}$
(integer constants)	n	$\in$	$\mathbb{Z}$
(basic levels)	k	$\in$	$\mathcal{L}$
(level expressions)	l	$::=$	$k \mid \tilde{x} \mid l_1 \sqcup l_2$
(integer expressions)	exp	$::=$	$x \mid n \mid exp_1\ \mathbf{op}\ exp_2 \mid \mathbf{read}\ x$
(expressions)	e	$::=$	$exp \mid l$
(commands)	cmd	$::=$	
			$\mathbf{skip} \mid (x_1, \ldots, x_n) := (e_1, \ldots, e_n) \mid$
			$\mathbf{if}\ e\ \mathbf{then}\ cmd_1\ \mathbf{else}\ cmd_2\ \mathbf{end} \mid cmd_1; cmd_2 \mid$
			$\mathbf{while}\ e\ \mathbf{do}\ cmd\ \mathbf{end} \mid \mathbf{send}\ x_1\ \mathbf{to}\ x_2 \mid$
			$\mathbf{if}\ l_1 \sqsubseteq l_2\ \mathbf{then}\ (\mathbf{send}\ x_1\ \mathbf{to}\ x_2)\ \mathbf{else}\ \mathbf{fail}\ \mathbf{end}$

The main difference between our source language and target language is that it adds support for *level variables*, a runtime representation of security levels. These level variables will allow target programs to verify that certain conditions are met before sending the contents of a variable to a channel. This is the goal of the new send command, nested in a conditional – we call it a *guarded send*.

For simplicity, we assume that security levels can be stored only in a restricted set of variables $\mathcal{V}_{level} \subseteq \mathcal{V}$. Thus, the variable part m_v of a memory m now has the following type

$$m_v : (\mathcal{V}_{level} \to \mathcal{L}) \uplus (\mathcal{V} \setminus \mathcal{V}_{level} \to \mathbb{Z} \uplus \mathcal{C})$$

Furthermore we assume that $\mathcal{V}_{level}$ contains variables _pc and _hc, and two level variables x_{val} and x_{ctx} associated with each variable $x \in \mathcal{V} \setminus \mathcal{V}_{level}$. Variables _pc and _hc hold the

security levels of the context and halting context respectively, they represent the security level in which a command is executed. We will discuss in more details what they represent in Section 3.6.4. Variables x_{val} and x_{ctx} are used to track the security levels of variables at runtime. For example, if x is a channel variable of security type $(int_\ell\ chan)_{\ell'}$, then the values of these variables should be $x_{\mathsf{val}} = \ell$ and $x_{\mathsf{ctx}} = \ell'$ (this will be ensured by our instrumentation).

The simultaneous assignment $(x_1, \ldots, x_n) := (e_1, \ldots, e_n)$ is introduced for the sake of clarity. Any assignment implies an immediate update of the concerned level variables. For all common commands, the semantics of the target language is the same as in the source language.

3.6.4 Instrumentation as a Type System

Our instrumentation algorithm is specified as a type system in Figure 3.2. Its goal is to inline monitor actions in the program under analysis, thereby generating a safe version of the program, or to reject the program when it contains obvious leaks of information. The inlined actions are essentially updates of level variables and checks on these variables in order to control the execution of potentially leaking **send** commands. After a check, a **send** command is either executed if it is safe or its execution is prevented and the program is aborted.

The typing rules of expressions have judgements of the form $\Gamma \vdash e : \sigma_\ell$, stating that σ_ℓ is the type of e under the typing environment $\Gamma : \mathcal{V} \uplus \mathcal{C} \to VarT$. The instrumentation judgements are of the form

$$\Gamma, pc, hc \vdash cmd : t, h, \Gamma', [\![cmd]\!]$$

where Γ, Γ' are typing environments (initially empty) and *cmd* is the command under analysis. The program context, $pc \in Lab$, is used to keep track of the security level in which a command is executed, in order to detect implicit flows. The halting context, $hc \in Lab$, is used to detect progress channels leaks. It represents the level of information that could have caused the program to *halt* (due to a failed guarded send command) or diverge (due to an infinite loop). In other words, it is the level of information that could be leaked through progress channels by an output. Variable $h \in Lab$ corresponds to the halting context after executing command *cmd*. The termination type $t \in \mathcal{T} = \{T, D, M_\ell\}$ of a command is used to keep the halting context up to date. We distinguish three termination types: T means that a command terminates for all memories, D means that a command diverges for all memories, M_ℓ means that a command's termination is unknown statically; the subscript is used to indicate on which level(s) the termination depends. For example, the termination of the loop in Listing 3.3 is $M_{\{H\}}$ because it can either terminate or diverge at runtime, and this depends on information of level H. The loop in Listing 3.4 on the other hand is of termination type T because, no matter what the value of variable h is, it will always eventually terminate. Similarly, a loop whose condition is always true will have termination type D since it always diverges. Recall that, of course, the precision of this analysis depends on the precision of the oracle. Variable $[\![cmd]\!]$ is the

$$\text{(S-CHAN)}\ \frac{levelOfChan(ch) = \ell}{\Gamma \vdash ch : (int_{\{\ell\}}\ chan)_{\{L\}}} \qquad \text{(S-INT)}\ \Gamma \vdash n : (int_{\{L\}})_{\{L\}} \qquad \text{(S-VAR)}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

$$\text{(S-READ)}\ \frac{\Gamma \vdash c : int_\ell\ chan_{\ell_c}}{\Gamma \vdash \textbf{read}\, c : (int_\ell)_{\ell_c}} \qquad \text{(S-OP)}\ \frac{\Gamma \vdash e_1 : (int_{\ell_1})_{\ell'_1} \qquad \Gamma \vdash e_2 : (int_{\ell_2})_{\ell'_2}}{\Gamma \vdash e_1\ \textbf{op}\ e_2 : (int_{\ell_1 \sqcup \ell_2})_{\ell'_1 \sqcup \ell'_2}}$$

$$\text{(S-SKIP)}\ \Gamma, pc, hc \vdash\ \textbf{skip} :\ T, hc, \Gamma, \mathsf{skip}$$

$$\text{(S-ASSIGN)}\ \frac{\Gamma \vdash e : \sigma_{\ell'_e}}{\Gamma, pc, hc \vdash x := e :\ T, hc, \Gamma[x \mapsto \sigma_{pc \sqcup \ell'_e}],\ \mathsf{genassign}}$$

$$\text{(S-SEND)}\ \frac{\Gamma(x) = (int_{\ell_x})_{\ell'_x} \qquad \Gamma(c) = (int_{\ell_c}\ chan)_{\ell'_c} \qquad (pc \sqcup hc \sqcup \ell_x \sqcup \ell'_x \sqcup \ell'_c) \sqsubseteq_s \ell_c}{\Gamma, pc, hc \vdash \textbf{send}\ x\ \textbf{to}\ c : T, hc, \Gamma, \textbf{send}\ x\ \textbf{to}\ c}$$

$$\text{(S-GSEND)}\ \frac{\begin{array}{c}\Gamma(x) = (int_{\ell_x})_{\ell'_x} \qquad \Gamma(c) = (int_{\ell_c}\ chan)_{\ell'_c} \\ (pc \sqcup hc \sqcup \ell_x \sqcup \ell'_x \sqcup \ell'_c) \not\sqsubseteq_s \ell_c \qquad (pc \sqcup hc \sqcup \ell_x \sqcup \ell'_x \sqcup \ell'_c) \sqsubseteq_m \ell_c\end{array}}{\Gamma, pc, hc \vdash \textbf{send}\ x\ \textbf{to}\ c : T, pc \sqcup hc \sqcup \ell'_x \sqcup \ell'_c, \Gamma,\ \mathsf{gengsend}}$$

$$\text{(S-IF)}\ \frac{\begin{array}{c}\Gamma \vdash e : (int_{\ell_e})_{\ell'_e} \qquad pc' = pc \sqcup \ell_e \sqcup \ell'_e \\ h_d = hasGSend(\{cmd_1, cmd_2\}, pc') \qquad h = \bigcup_{j \in \{1,2\}} (h_j \sqcup h_d \sqcup level(t_1 \oplus_{pc'} t_2)) \\ \Gamma, pc', hc \vdash cmd_j : t_j, h_j, \Gamma_j, [\![cmd_j]\!] \quad j \in \{1,2\}\end{array}}{\Gamma, pc, hc \vdash\ \textbf{if}\ e\ \textbf{then}\ cmd_1\ \textbf{else}\ cmd_2\ \textbf{end} : t_1 \oplus_{pc'} t_2, h, \Gamma_1 \sqcup_{pc'} \Gamma_2, \mathsf{genif}}$$

$$\text{(S-LOOP1)}\ \frac{\begin{array}{c}\Gamma = \Gamma \sqcup_{pc'} \Gamma' \qquad \Gamma \vdash e : (int_{\ell_e})_{\ell'_e} \\ O(e, cmd, \Gamma) = t_o \qquad pc' = pc \cup (pc \sqcup \ell_e \sqcup \ell'_e) \qquad hc' = hc \cup (hc \sqcup level(t') \sqcup h') \\ h_d = hasGSend(\{cmd\}, pc') \qquad \Gamma, pc', hc' \vdash cmd : t', h', \Gamma', [\![cmd]\!]\end{array}}{\Gamma, pc, hc \vdash \textbf{while}\ e\ \textbf{do}\ cmd\ \textbf{end} : t_o, h_d \sqcup h' \sqcup level(t_o), \Gamma, \mathsf{genwhile}}$$

$$\text{(S-LOOP2)}\ \frac{\begin{array}{c}\Gamma \neq \Gamma \sqcup_{pc'} \Gamma' \qquad \Gamma \vdash e : (int_{\ell_e})_{\ell'_e} \qquad \Gamma, pc \sqcup \ell_e \sqcup \ell'_e, hc \vdash cmd : t', h', \Gamma', [\![cmd]\!] \\ pc' = pc \cup (pc \sqcup \ell_e \sqcup \ell'_e) \qquad hc' = hc \cup (hc \sqcup level(t') \sqcup h') \\ \Gamma \sqcup_{pc'} \Gamma', pc', hc' \vdash \textbf{while}\ e\ \textbf{do}\ cmd\ \textbf{end} : t'', h'', \Gamma'', \mathsf{instCode}\end{array}}{\Gamma, pc, hc \vdash \textbf{while}\ e\ \textbf{do}\ cmd\ \textbf{end} : t'', h'', \Gamma'', \mathsf{instCode}}$$

$$\text{(S-SEQ1)}\ \frac{\Gamma, pc, hc \vdash cmd_1 : D, h, \Gamma_1, [\![cmd_1]\!]}{\Gamma, pc, hc \vdash cmd_1; cmd_2 : D, h, \Gamma_1, [\![cmd_1]\!]}$$

$$\text{(S-SEQ2)}\ \frac{\begin{array}{c}t_1 \neq D \qquad \Gamma, pc, hc \vdash cmd_1 : t_1, h_1, \Gamma_1, [\![cmd_1]\!] \\ \Gamma_1, pc, h_1 \vdash cmd_2 : t_2, h_2, \Gamma_2, [\![cmd_2]\!]\end{array}}{\Gamma, pc, hc \vdash cmd_1; cmd_2 : t_1 \fatsemi t_2, h_2, \Gamma_2, [\![cmd_1]\!]; [\![cmd_2]\!]}$$

Figure 3.2: Instrumentation and typing rules for the source language

generated instrumented version of command *cmd* and is often presented using a macro whose name starts with *gen*.

The instrumentation of a program p begins by inserting commands to initialize a few level variables: $_\mathsf{pc}$, $_\mathsf{hc}$ are initialized to L, as well as the level variables x_{ctx} and x_{val} for each variable $x \in \mathcal{V}$ appearing in p. Similarly, level variables c_{ctx} and c_{val} associated with each channel c used in p are also initialized, but the latter rather gets initialized to $\mathit{levelOfChan}(c)$, which is an input parameter for the analysis. After the initialization, the instrumentation is given by the rules of Figure 3.2. We now explain these rules.

Rules (S-CHAN) and (S-INT) specify the type of channel and integer constants respectively.

Rule (S-VAR) infers the type of a variable from the environment Γ.

Rule (S-OP) infers the type of an expression from the types of its operands and it reflects the fact that operations on channels are not allowed.

Rule (S-READ) returns the security type of the current value in channel c. This rule highlights the striking similarity between type $\mathit{int}_{\ell}\ \mathit{chan}_{\ell_c}$ of a channel and type $(\mathit{int}_{\ell})_{\ell_c}$ of a variable that may subsequently store expression **read** c. Here, the security label ℓ of channel's value type becomes the security label of variable's value type. Similarly for the security label ℓ_c of channel's variable type. To the best of our knowledge, this is the first work that tries to use similar type structures for channels and ordinary variables in an enforcement mechanism.

Rule (S-ASSIGN) updates the type of x in the environment with the type of expression e. The supremum with pc is used to prevent implicit flows. Its instrumentation is given by the following macro, which represents a simultaneous assignment $x := e, x_{\mathsf{val}} := e_{\mathsf{val}}$, and $x_{\mathsf{ctx}} := e_{\mathsf{ctx}} \sqcup _\mathsf{pc}$.

$$\begin{aligned}&\mathsf{genassign} = \\ &\qquad (x, x_{\mathsf{val}}, x_{\mathsf{ctx}}) := (e, e_{\mathsf{val}}, e_{\mathsf{ctx}} \sqcup _\mathsf{pc})\end{aligned}$$

We write e_{ctx} to represent the expression made of the supremum of variables appearing in expression e. For example if $e = x + \textbf{read}\ c$, then $e_{\mathsf{ctx}} = x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}$. If $e = x + y$ then $e_{\mathsf{ctx}} = x_{\mathsf{ctx}} \sqcup y_{\mathsf{ctx}}$. The idea is the same for e_{val}.

Rule (S-SEND) checks whether a send command is statically safe by requiring $(pc \sqcup hc \sqcup \ell_x \sqcup \ell'_x \sqcup \ell'_c) \sqsubseteq_s \ell_c$ (i.e., all possible values of the left-hand side are always lower or equal to the right-hand side). The variables on the left-hand side correspond to the level of information that can be revealed by the output to c. If so, the instrumentation inserts the send command as it is.

Rule (S-GSend) checks whether a send command may be safe, by requiring $(pc \sqcup hc \sqcup \ell_x \sqcup \ell'_x \sqcup \ell'_c) \sqsubseteq_m \ell_c$. The instrumentation then transforms it into a guarded send, as follows

$$
\begin{aligned}
&\mathsf{gengsend} = \\
&\quad \textbf{if } _\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{val}} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}} \sqsubseteq c_{\mathsf{val}} \textbf{ then} \\
&\quad\quad (\textbf{send } x \textbf{ to } c) \\
&\quad \textbf{else} \\
&\quad\quad \textbf{fail} \\
&\quad \textbf{end}; \\
&\quad _\mathsf{hc} := _\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}};
\end{aligned}
$$

The halting context must record the possible failure of a guarded send at runtime, and hence, it is updated with the level of information that influences its success/failure. Particularly, the halting context is updated with the sensitivity of the context and of the two variables involved, the channel variable [6] and the regular variable [15]. For example, Listing 3.8 shows why c_{ctx} must be included in this update.

```
if x > 0 (*H at runtime *)
  then c := lowChannel
  else c := highChannel
end;
send highValue to c (*guarded send*)
send lowValue to lowChannel
```

Listing 3.8: Dangerous runtime halting

Assume that x is high and false at runtime. Then the first guarded send is accepted, but allowing an output on a low security channel subsequently would leak information about x. This is because, had x been true, then the first send would have failed. However, c has level $int_{\{L,H\}} chan_{\{H\}}$ after the conditional. Updating _hc with c_{ctx} will affect the check of all subsequent guarded send and prevent such leaks.

If none of the send rules can be applied, then the program is statically rejected.

Before explaining the rules for composed commands, we first need to define a few functions and operators. For the conditional rules, we need a supremum of environments.

Definition 3.4. *The supremum of two environments is given as* $dom(\Gamma_1 \sqcup_{pc} \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2)$*, and*

$$(\Gamma_1 \sqcup_{pc} \Gamma_2)(x) =$$
$$\begin{cases} \Gamma_i(x) & \text{if } x \in dom(\Gamma_i) \backslash dom(\Gamma_j), \\ & \{i,j\} = \{1,2\} \vee \Gamma_1(x) = \Gamma_2(x) \\ (int_{\ell_1 \cup \ell_2}\, chan)_{(\ell'_1 \cup \ell'_2) \sqcup pc} & \text{if } \Gamma_1(x) = (int_{\ell_1}\; chan)_{\ell'_1} \wedge \\ & \Gamma_2(x) = (int_{\ell_2}\; chan)_{\ell'_2} \wedge \\ & \Gamma_1(x) \neq \Gamma_2(x) \\ (int_{\ell_1 \cup \ell_2})_{(\ell'_1 \cup \ell'_2) \sqcup pc} & \text{if } \Gamma_1(x) = (int_{\ell_1})_{\ell'_1} \wedge \\ & \Gamma_2(x) = (int_{\ell_2})_{\ell'_2} \wedge \\ & \Gamma_1(x) \neq \Gamma_2(x) \\ Error & \text{otherwise.} \end{cases}$$

When a typing inconsistency occurs, e.g., when a variable is used as an integer in one branch and as a channel in another, the analysis stops and an error is returned, causing the program to be statically rejected.

The function $level : \mathcal{T} \rightarrow \mathcal{P}(\mathcal{L}) \setminus \{\emptyset\}$ returns the *termination level* (i.e., the level that the termination depends on) and is defined as:

$$level(t) = \begin{cases} \{L\} & \text{if } t \in \{T, D\} \\ \ell & \text{if } t = M_\ell \end{cases}$$

Two operators are used to compose terminations types, $\oplus$, used in the typing of conditionals, and $⨟$, used in the typing of sequences. They are defined as follows.

$$t_1 \oplus_{pc} t_2 = \begin{cases} t_1 & \text{if } t_1 = t_2 \in \{T, D\} \\ M_{pc \sqcup (\ell_1 \cup \ell_2)} & \text{otherwise,} \\ & \ell_1 = level(t_1), \ell_2 = level(t_2) \end{cases}$$

$$t_1 ⨟ t_2 = \begin{cases} M_{\ell_1 \sqcup \ell_2} & \text{if } t_1 = M_{\ell_1} \text{ and } t_2 = M_{\ell_2} \\ t_i & \text{if } t_j = T, \{i,j\} = \{1,2\} \\ D & \text{otherwise} \end{cases}$$

The following example illustrates the use of these operators:

```
if highValue then
  while 1 do skip end (* D *)
else
  skip (* T *)
end
```

The termination type of an **if** command is computed using the $\oplus$ operator, from three parameters: the termination types of each of the two branches and the security level of the guard condition. Hence, in this example, we obtain $D \oplus_H T = M_{\{H\}}$.

One more function needs to be defined. Its motivation is given by the following example. Assume that in Listing 3.9 `c` turns out to be a low channel at runtime. If the last send command is reached and executed it would leak information about `highValue`. The same leak would happen if instead of the guarded send we had a diverging loop.

```
if highValue > 0 then
 if ℓ ⊑ ℓ′
   then send highValue to c
   else fail end;
end;
send lowValue to lowChannel
```

Listing 3.9: A guarded send can generate a progress leak

To prevent this kind of leak, we verify if the branches of a conditional contains guarded send commands using the function

$$hasGSend : \mathcal{P}(Cmd) \times Lab \rightarrow Lab,$$

where Cmd is the set of commands and Lab the set of security labels. If one of the commands given in parameter contains a guarded send, then it returns the security label given in parameter, otherwise L is returned. This function is used to update the halting context to pc when there is a risk that one of the branch halts the execution. Without this update, hc could leak information about the condition in a subsequent send.

Rule (S-IF) specifies the typing of an **if** command. When typing an **if** command, we type the two branches under pc', which is the supremum of the conditional's guard expression and current context. The resulting typing environments, Γ_1 and Γ_2, then contain the security levels that variables may have after executing the first or second branch. The typing environment returned by the **if** is the join of those two, defined in Definition 3.4, so that it contains the possibilities of both branches. Similarly, variable h is used to calculate the possible values that the halting context may have after the conditional, hence the union.

Its instrumentation is given by the following macro:

```
genif =
        _oldpcν := _pc;
        if e then
            _pc:=_pc ⊔ e_val ⊔ e_ctx;
            ⟦cmd1⟧;
            update(mv2)
        else
            _pc:=_pc ⊔ e_val ⊔ e_ctx;
            ⟦cmd2⟧;
            update(mv1)
        end;
        _pc := _oldpc
```

where update is defined as follows

```
update(mv) =
        if mv = ∅  skip;
        else
          for each x ∈ mv   x_ctx := x_ctx ⊔ _pc;
          if _hc ∈ mv   _hc := _hc ⊔ _pc;
```

and where t_j is the termination type of cmd_j, mv_j is the set of modified variables in cmd_j (we include _hc in this set if the termination of the two branches can differ i.e. if $\neg(t_1 = t_2 \in \{T, D\})$ or if at least one guarded send occurs in the other branch), and $e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$ is the guard condition's level expression.

The instrumented code starts by saving the current context to $_\mathsf{oldpc}^{\nu}$ (the symbol ν indicates that it is a fresh variable). The program context is updated with the security level of the guard condition. The **if** itself is then generated.

The function update generates the command skip; if the parameter set mv is empty otherwise it generates updates of the context level of each modified variable in the other branch as well as the update of _hc if necessary. The underlying reason is to ensure that the value of these level variables is at least pc.

In a situation like the following listing, this function allows to update y's level, to protect x.

```
y := 0;
if x > 0 then (*H at runtime*)
  y := 1
else skip end;
send x to lowChannel
```

Listing 3.10: Modified variables

Here, even if the else branch is taken at runtime, the level of variable y must be updated. Otherwise, information about x would be revealed by the send command (even with a guarded send).

Rules (S-Loop1), (S-Loop2) specify the **while** command typing. They involve computing a least fixed point to derive the right typing environment. This is necessary because of the flow sensitivity feature. Typing rule (S-Loop2) is applied recursively until a least fixpoint is found, at which point (S-Loop1) is applied and its result is returned. Note that, since the calculation of the fixpoint is defined in a constructive way, the rule will not accept any other fixpoints than a least fixpoint and it will always compute the same least fixpoint. The union operator is used to update the pc' and hc' variables so that we keep track of all their possible values. Due to our use of finite lattices, and the monotonicity of the union and supremum on levels, it is easy to show that this computation converges, the proof is given in 3.13, Lemma 3.4. The typing relies on an oracle O that returns the termination type of the loop (t_o). It is worth noting that the call to the oracle is performed statically. Calling it dynamically would enhance precision, but increase significantly the overhead. If the loop contains guarded send commands, which could fail and reveal information about the condition of the loop, then we update the halting context to prevent this leak. The presence of at least one guarded send command is detected using the function *hasGSend*().

$$
\begin{aligned}
\mathsf{genwhile} = \; & \\
& \mathsf{_oldpc}^{\nu} := \mathsf{_pc}; \\
& \textbf{while } e \textbf{ do} \\
& \quad \mathsf{_pc} := \mathsf{_pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}; \\
& \quad [\![cmd]\!]; \\
& \textbf{end}; \\
& \mathsf{_pc} := \mathsf{_pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}; \\
& \mathsf{update}(mv); \\
& \mathsf{_pc} := \mathsf{_oldpc}
\end{aligned}
$$

The inserted commands are similar to those of the **if** command. The level variables and halting context are updated after the loop in case an execution does not enter the loop. The context needs to be updated at the begining of each iteration as the value, and hence level, of expression e may change.

Rule (S-Seq1) is applied if cmd_1 always diverges; we then ignore cmd_2, as it will never be executed. Otherwise, rule (S-Seq2) is applied. The halting context returned is h_2 instead of $h_1 \sqcup h_2$ because h_2 already takes into account h_1.

Examples of instrumented programs are available in 3.12.

3.7 Soundness

In order to prove that the instrumented program generated by Figure 3.2 correctly enforces noninterference, we need to adapt the definitions of noninterference and ℓ-equivalent memories to our target language, because of level variables. Recall that a memory for the target language is the union of two maps of the following form:

$$m_v : (\mathcal{V}_{level} \to \mathcal{L}) \uplus (\mathcal{V} \setminus \mathcal{V}_{level} \to \mathbb{Z} \uplus \mathcal{C}),$$
$$m_c : \mathcal{C} \to \mathbb{Z}.$$

We write $\text{dom}_l(m) := \text{dom}(m_v) \cap \mathcal{V}_{level} = m_v^{-1}(\mathcal{L})$. A memory m is called *complete* for a program p if

- $\{x_{\mathsf{val}}, x_{\mathsf{ctx}}\} \subseteq \text{dom}_l(m)$ for any $x \in \mathcal{C} \cup \text{dom}(m) \setminus \mathcal{V}_{level}$ that appears in p.
- $_\mathsf{pc}$ and $_\mathsf{hc}$ are in $\text{dom}_l(m)$
- if $c \in \mathcal{C}$ appears in p, then $m_v(c_{\mathsf{val}}) = \mathit{levelOfChan}(c)$ and $m_v(c_{\mathsf{ctx}}) = L$
- if $m(x) \in \mathcal{C}$ then $m_v(x_{\mathsf{val}}) = \mathit{levelOfChan}(m(x))$.

The first two conditions ensure that level variables exist in the domain of the memory, whereas the last ones makes sure that it is compliant with the security policy for channels.

The definition of ℓ-equivalent memories, which is based on [15], must handle level variables. Whenever the level variable x_{ctx} corresponding to a variable x is such that $m(x_{\mathsf{ctx}}) \sqsubseteq \ell$ in one memory, then it must have the same value in both memories, otherwise a leak can happen.

Definition 3.5. *ℓ-equivalent memories. We say that two complete memories of the target language m_1 and m_2 are* ℓ-equivalent*, written $m_1 \equiv_\ell m_2$, iff they satisfy the following properties*

1. *if $m_i(_\mathsf{pc}) \sqsubseteq \ell$ for some $i \in \{1, 2\}$, then $m_1(_\mathsf{pc}) = m_2(_\mathsf{pc})$. The same property holds for $_\mathsf{hc}$.*
2. *$x \in m_{i,v}^{-1}(\mathbb{Z} \cup \mathcal{C}), \wedge m_i(x_{\mathsf{ctx}}) \sqsubseteq \ell$, for some $i = 1, 2$, then*
   - *$m_1(x_{\mathsf{ctx}}) = m_2(x_{\mathsf{ctx}})$*
   - *$m_1(x_{\mathsf{val}}) = m_2(x_{\mathsf{val}})$*
   - *if $m_1(x_{\mathsf{val}}) \sqsubseteq \ell$ then $m_1(x) = m_2(x)$*
3. *$c \in \mathcal{C} \wedge \mathit{levelOfChan}(c) \sqsubseteq \ell \Rightarrow m_{1_c}(c) = m_{2_c}(c)$*

It may seem surprinsing that the memories may differ on a level variable such as x_{val}; this is because they may differ on the value of high variables. Too see this, here is a variation of Listing 3.1.

```
if highValue > 0
 then c := highChannel
 else c := lowChannel
end;
```

Listing 3.11: The security type of c's content is sensible

In this example, the content of c depends on a private condition, and hence its level variable $\mathsf{c_{ctx}}$ should be H; moreover, variable $\mathsf{c_{val}}$, containing the level of the content of c, may have different values in two ℓ-equivalent memories. Another example, for non-channel variables, is Listing 3.6, reproduced below, where medValue has security level M, with $L \sqsubseteq M \sqsubseteq H$.

```
if medValue > 0
 then x := highValue  (*x is H *)
 else x := lowValue   (*x is L *)
end;
send x to c
send 1 to lowChannel
```

In this example, variable x_{val} contains information of level M. If c turns out to be of level M at execution then reaching or not the last send command will depend on x_{val}. The program will be blocked or not, and this will reveal which branch was taken in the preceding conditional. Interestingly, this problem does not arise when the lattice is restricted to only two levels, $\{L, H\}$, as argued in Kozyri et al. [15].

Here is the definition of noninterference for the target language. The difference from Definition 3.1 is the requirement of the memories to be complete and the use of Definition 3.5.

Definition 3.6. *Progress-sensitive noninterference. We say that a program p satisfies* progress-sensitive noninterference *if for any $\ell \in \mathcal{L}$, and for any two complete memories m and m' that are ℓ-equivalent, and for any trace o such that $\langle p, m, \epsilon \rangle \downarrow o$, then there is some trace o', such that $\langle p, m', \epsilon \rangle \downarrow o'$ and $o \upharpoonright \ell = o' \upharpoonright \ell$.*

Using these updated definitions, we prove that the instrumented programs are noninterferent.

Theorem 3.1 (Soundness of enforcement)**.** *If a program p is well typed according to the type system of Figure 3.2, then the generated program $[\![p]\!]$ satisfies progress sensitive noninterference.*

We also show that the instrumentation preserves the semantics of the original program. That is, the instrumentation of a program p, written $[\![p]\!]$, produces exactly the same output as p as

long as it is allowed to continue; it may be stopped at some point to prevent a leak. If m is a memory for the target language, we write $\hat{m}$ for the restriction of m to $\mathcal{V} \setminus \mathcal{V}_{level}$.

Theorem 3.2 (Semantics preservation)**.** *Let p be a program, m a memory, and o, o' output traces. Then*

$$\langle [\![p]\!], m, \epsilon \rangle \downarrow o \Rightarrow \ \langle p, \hat{m}, \epsilon \rangle \downarrow o$$
$$(\langle p, \hat{m}, \epsilon \rangle \downarrow o \wedge \langle [\![p]\!], m, \epsilon \rangle \downarrow o') \Rightarrow \ o \preceq o' \vee o' \preceq o$$

where $o' \preceq o$ means that o' is a prefix of o.

The proofs are available in Section 3.13.

3.8 Increasing Precision and Permissiveness

During the course of this work, we thought of two ways to improve the precision of our static analysis and permissiveness of our dynamic analysis. While we chose not to use them into this work (to keep things as simple as possible), we think they are worth pointing out.

3.8.1 Security Type Constraints

In Listing 3.13, only executions where `c` is a high channel will get past the first guarded send. For this reason, we can consider, for the rest of the analysis, that its type is $int_{\{H\}} chan_{\{L\}}$ instead of $int_{\{L,H\}} chan_{\{L\}}$.

```
if lowValue > 0 then
  c:= lowChannel
else
  c:= highChannel
end;
send highValue to c; (*will be transformed into a guarded send*)
(*to reach here, c must be {H}*)
x := read c; (*so x is {H}*)
send x to lowChannel (*always leaks, so statically rejected*)
```

Listing 3.13: Constraint on the security type of a channel variable

The same idea applies to integer variables. For example, we know that the instructions after the first **send** of Listing 3.14 will only be reached if variable `x` is low. For this reason, we can consider that `x`'s type after the **send** will be $(int_L)_{\{L\}}$ instead of $(int_{\{L,H\}})_{\{L\}}$.

```
if lowValue > 0 then
  c := lowChannel
else
  c := highChannel
end;
x := read c; (*x is {L,H}*)
send x to lowChannel; (*will be transformed into a guarded send*)
(*to reach here, x must be {L}*)
send x to lowChannel (*no need to transform into a guarded send*)
```

Listing 3.14: Constraint on the security type of an integer variable

Using these constraints in our static analysis would lead to the insertion of fewer guarded sends, and thus, less runtime overhead.

3.8.2 Conditional Updates of the Halting Context

While we chose to always update the halting context _hc after a guarded send with

$$_\mathsf{hc} := _\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}},$$

there are cases where we can be more precise. One such case is illustrated in Listing 3.15.

```
if medValue then
  x := read highChannel
else
  x := read highChannel2
end;
send x to c; (*guarded send*)
send lowValue to lowChannel
```

Listing 3.15: Example where _hc does not need to be updated with $\mathsf{x}_{\mathsf{ctx}}$

In this example, while x's value may differ, its type is constant and equal to $(int_{\{H\}})_{\{M\}}$. Since the value of medValue does not affect its type, it means that it has no influence on the guarded send's decision to block or allow the output. Hence, in this case, the update to _hc variable after the guarded send does not need to include variable $\mathsf{x}_{\mathsf{ctx}}$.

Similarly, when a channel c whose type is constant is used in a guarded send command, the update to _hc does not need to include variable $\mathsf{c}_{\mathsf{ctx}}$.

Hence, using conditional updates would allow the dynamic analysis to be more permissive as the updates to _hc are less conservative.

3.9 Related Work

There has been much research into language-based techniques for controlling information flow over the last two decades. In this section, we focus on hybrid techniques for information-flow control. Hybrid techniques are attractive as the combination of static and dynamic analyses offers potentially multiple advantages such as low runtime overhead, increased precision and flexibility.

Le Guernic et al. [3] present the first hybrid information-flow control monitor. The enforcement is based on a monitor that is able to perform static checks during the execution. The enforcement is not flow-sensitive. Le Guernic, in [16], extends this work to concurrent programs. Russo and Sabelfeld [5] generalize their work, presenting a series of hybrid monitors that differ on the action to perform in the event of a security violation. They also state that purely dynamic enforcements are more permissive than purely static enforcements but they cannot be used in case of flow-sensitivity. They propose a hybrid flow-sensitive enforcement based on calling static analysis during the execution. This enforcement is not progress sensitive.

Kozyri et al. [15] show that it is not trivial to design dynamic enforcement mechanisms that support general lattices, are flow-sensitive, and do not leak information through termination enforced by the monitor. Their mechanism supports all these three features by using *metalabels*, which are labels on labels. A metalabel represents the sensitivity of the corresponding label, in the same way that ℓ represents the sensitivity of σ in a variable type σ_ℓ.

Bedford et al. [17] generate instrumented code, enforcing information flow based on static analysis (i.e., an information-flow monitor is inlined). The approach supports channel variables and is flow sensitive, but does not take into account leaks due to progress. Also, the inlined monitor does not use dynamic security levels, but employs a heavy-handed approach which is not as amenable to standard optimization techniques as the present one. The target language is not formally defined and no soundness proof of the instrumented code is provided.

Moore et al. [13] consider precise enforcement of flow-insensitive progress-sensitive security. Progress sensitivity is also based on an oracle's analysis, but they call upon it dynamically while we do it statically. We have also introduced additional termination types to increase the permissiveness of the monitor.

Chudnov and Naumann [18] inline a flow-sensitive hybrid monitor (based on a monitor of Russo et al. [5]) and extend it to Javascript [19]. They prove its soundness by showing that the execution of the inlined monitor is bisimilar to the execution of a non-inlined monitor. We inline a flow-sensitive progress-sensitive hybrid monitor and, as we did not have already have a non-inlined monitor, we proved its soundness by showing that the output traces produced by two ℓ-equivalent executions will always be the same.

Magazinius et al. [20] present on-the-fly inlining of a dynamic information security monitor. We speculate that we could extend their ideas to allow on–the fly instrumentation.

Assaf et al. [21] enforce termination-insensitive noninterference on a simple imperative language that support pointers using hybrid monitor. To deal with pointer-aliasing, they use a set of memory locations, which is similar to our sets of levels. One difference is that we use sets to distinguish outputs that never leak information from those that may leak information. Since we only insert runtime checks for the outputs that may leak information, our monitor introduces less overhead at runtime.

Askarov and Sabelfeld [22] use hybrid monitors to enforce information security in dynamic languages. In this setting, dynamic evaluation of programs (e.g., eval statements in JavaScript) requires on-the-fly static analysis of programs. They provide a model to define noninterference that is suitable to progress-sensitivity and they quantify information leaks due to termination [7].

Hritcu et al. [23] introduces an error-handling mechanism that allows all errors (even those caused by an information-flow control violation) to be safely recoverable. They support dynamic levels. To help prevent leaks through covert channels, they provide a discretionary access control mechanism called clearance that allows them to put an upper bound on the *pc*. Contrarily to our approach, the detection (and prevention) of leaks through progress channels is not done automatically.

Askarov et al. [24] introduce a hybrid monitoring framework capable of handling concurrent programs. They illustrate their approach on a simple imperative language similar to ours, but it does not support channel variables. In their framework, each thread is guarded by its own local monitor (progress- and flow-sensitive). There is also a single global monitor that synchronizes the threads. Like us, they make use of an oracle to approximate the termination behaviour of branches. This oracle is called upon at runtime (making it a kind of on-the-fly static analysis), whereas ours is called only statically. The main difference between their approach and ours, exluding the concurrency of course, is the fact that our monitor is inlined whereas theirs is not.

3.10 Conclusion

We have presented a hybrid information flow enforcement mechanism, which detects and prevents leaks that may occur through the data-flow or the progress of a program. It uses information inferred during a phase of static analysis to instrument the program; this helps to reduce the number of false positives during the execution. The instrumented program uses level variables, a simple yet powerful way, to perform its dynamic analysis. This instrumented code can then be partially evaluated in order to reduce the amount of added commands.

Our main contributions are the following.

(a) We present an extended version of the hybrid monitor first presented in our previous article [6]. It is capable of enforcing flow- and progress-sensitive information security on general lattices. It is more precise and introduces less overhead than currently available solutions (e.g., [11, 13]) for two reasons: it makes use of a static termination oracle and does not approximate the level of a variable at the join of a conditional. Since our monitor is inlined, it can be easily optimized using classical partial evaluation techniques, [9].

(b) We prove the soundness of our inlined monitor and that the semantics of the original program is preserved, as long as it is secure.

(c) We show that, thanks to the use of sets of levels, it is possible to distinguish outputs that *never* leak information from outputs that *may* leak information.

(d) We present two ideas to increase the precision of the static analysis and the permissiveness of the dynamic analysis: propagating constraints on the set of possible security levels and using conditional updates. We chose not to use them in this work in order to increase readability.

Future Work Future work includes extensions to concurrency, declassification and information leakage due to timing. We would like to scale up the approach to deal with real world languages and to test it on elaborate programs. The use of abstract interpretation [25] to enhance the static analysis is also to be considered in future work.

Acknowledgments We would like to thank Fred B. Schneider and the reviewers for their valuable comments. Andrew Bedford, Josée Desharnais, and Nadia Tawbi are supported by grants from Laval University and NSERC. Elisavet Kozyri is supported by AFOSR grants F9550-16-0250 and grants from Microsoft.

3.11 Bibliography

[1] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, 2003.

[2] D. Volpano, C. Irvine, and G. Smith, “A sound type system for secure flow analysis,” *Journal of computer security*, vol. 4, no. 2, pp. 167–187, 1996.

[3] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt, “Automata-based confidentiality monitoring,” *Asian Computing Science Conference*, 2006.

[4] T. H. Austin and C. Flanagan, “Efficient purely-dynamic information flow analysis,” in *Proceedings of the Workshop on Programming Languages and Analysis for Security*, 2009.

[5] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *CSF*, 2010, pp. 186–199.

[6] A. Bedford, S. Chong, J. Desharnais, and N. Tawbi, "A progress-sensitive flow-sensitive inlined information-flow control monitor," in *ICT Systems Security and Privacy Protection - 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30 - June 1, 2016, Proceedings*, 2016, pp. 352–366.

[7] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Proceedings of the European Symp. on Research in Computer Security: Computer Security*, 2008.

[8] L. Zheng and A. C. Myers, "Dynamic security labels and noninterference," in *Formal Aspects in Security and Trust*. Springer, 2005, pp. 27–40.

[9] N. D. Jones, C. K. Gomard, and P. Sestoft, *Partial evaluation and automatic program generation*. Prentice Hall, 1993.

[10] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, pp. 236–243, May 1976.

[11] K. R. O'Neill, M. R. Clarkson, and S. Chong, "Information-flow security for interactive programs," in *CSFW*. IEEE, 2006.

[12] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *POPL*, 1998.

[13] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *CCS 2012*, 2012.

[14] B. Cook, A. Podelski, and A. Rybalchenko, "Proving program termination," *Commun. ACM*, vol. 54, no. 5, pp. 88–98, May 2011.

[15] E. Kozyri, J. Desharnais, and N. Tawbi, "Block-safe information flow control," Department of Computer Science, Cornell University, Tech. Rep., Aug. 2016.

[16] G. L. Guernic, "Automaton-based confidentiality monitoring of concurrent programs," in *CSF*, 2007.

[17] A. Bedford, J. Desharnais, T. G. Godonou, and N. Tawbi, "Enforcing information flow by combining static and dynamic analysis," in *Proceedings of the International Symposium on Foundations & Practice of Security*, 2013.

[18] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *Proceedings of the 23rd IEEE Security Foundations Symposium*, 2010.

[19] ——, "Inlined information flow monitoring for javascript," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 629–643. Available: http://doi.acm.org/10.1145/2810103.2813684

[20] J. Magazinius, A. Russo, and A. Sabelfeld, "On-the-fly inlining of dynamic security monitors," *Computers & Security*, vol. 31, no. 7, pp. 827–843, 2012.

[21] M. Assaf, J. Signoles, F. Tronel, and É. Totel, "Program transformation for non-interference verification on programs with pointers," in *IFIP International Information Security Conference*. Springer, 2013, pp. 231–244.

[22] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *CSF*, 2009.

[23] C. Hritcu, M. Greenberg, B. Karel, B. C. Pierce, and G. Morrisett, "All your ifcexception are belong to us," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 3–17.

[24] A. Askarov, S. Chong, and H. Mantel, "Hybrid monitors for concurrent noninterference," in *Computer Security Foundations Symposium*, 2015.

[25] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, 1977, pp. 238–252.

3.12 Appendix: Examples

In order to simplify the examples, we assume that the variables `lVal`, `mVal`, `hVal` and channels `lChan`, `mChan`, `hChan` already exist, have arbitrary values and have the suggested security levels. We also assume that all other level variables have been initialized to L. Boolean values are represented as integer where 0 means false and any other integer means true.

Guarded send

The following example illustrates a situation where the guarded send is used to prevent a possible leak of information.

```
if  mVal > 0  then
  c := mChan
else
  c := hChan
end;
send hVal to c; (*may leak information*)
```

Here is its instrumentation :

```
(*if*)
_oldpc1 := _pc;
if lVal > 0 then
  _pc := _pc ⊔ lVal_val ⊔ lVal_ctx ⊔ L;
  (c, c_val, c_ctx) := (lChan, lChan_val, lChan_ctx ⊔ _pc); (*assign*)
  (c, c_ctx) := (c, c_ctx ⊔ _pc); (*update*)
else
  _pc := _pc ⊔ lVal_val ⊔ lVal_ctx ⊔ L;
  (c, c_val, c_ctx) := (hChan, hChan_val, hChan_ctx ⊔ _pc); (*assign*)
  (c, c_ctx) := (c, c_ctx ⊔ _pc); (*update*)
end
_pc := _oldpc1;

(*guarded send*)
if _hc ⊔ _pc ⊔ hVal_val ⊔ hVal_ctx ⊔ c_ctx ⊑ c_val then
  (send hVal to c)
else fail end;
_hc := _pc ⊔ _hc ⊔ hVal_ctx ⊔ c_ctx;
```

Divergence

Commands after a loop that always diverges are ignored. Hence, the following program is statically safe.

```
while 1 do
  skip
end;
send hVal to lChan
```

Even if it is statically safe, it is instrumented :

```
(*while*)
_oldpc1 := _pc;
while 1 do
  _pc := _pc ⊔ L ⊔ L;
  (*skip*)
  skip;
end;
_pc := _pc ⊔ L ⊔ L;
_pc := _oldpc1
```

After partial evaluation, it results in the following program:

```
while 1 do
  skip
end
```

Notice that because the code is statically safe, the partial evaluation is able to get rid of the instructions added by the instrumentation algorithm. This is because, if the code is statically safe, then the conditions of the guarded send commands are all true. If guarded send commands are not needed, then level variables are also not needed.

3.13 Appendix: Proofs

We prove that the type system of Figure 3.2 generates noninterferent programs (i.e., we prove its soundness).

Theorem 3.1 (Soundness of enforcement) *If a program p is well typed according to the type system of Figure 3.2, then the generated program $[\![p]\!]$ satisfies progress-sensitive noninterference.*

The theorem is a consequence of the following results.

Lemma 3.1. *If two memories are ℓ-equivalent, then they agree on every expressions involving level variables that were generated by the instrumentation and whose value is $\sqsubseteq \ell$; these expressions include $_\mathsf{pc}, _\mathsf{hc}, e_{\mathsf{val}}, e_{\mathsf{ctx}}$, etc.*

Sketch. The proof is by induction. The idea is to show that all such expressions either only use level variables of the form x_{ctx} which are $\sqsubseteq \ell$, on which ℓ-equivalent memories agree by definition, or that whenever these expressions use some x_{val}, then they *protect* this potentially dangerous variable by x_{ctx}. The program context, $_\mathsf{pc}$, is modified in the conditional and the loop, where the supremum is taken from the level of the condition. □

Notation If $\Gamma(x) = \sigma_{\ell'_x}$, we write $\Gamma_{\mathsf{ctx}}(x)$ to mean ℓ'_x. If $\Gamma(x) = (int_{\ell_x}\ chan)_{\ell'_x}$ or $(int_{\ell_x})_{\ell'_x}$, then we write $\Gamma_{\mathsf{val}}(x)$ to mean ℓ_x.

The following lemma states that security types of variables calculated by the typing system of Figure 3.2 include all possible runtime values.

Lemma 3.2. *For any execution starting with a complete memory, if the runtime memory is m when an instrumented command $[\![cmd]\!]$ is executed, and m' is the memory after the execution, assuming the typing generated by Figure 3.2 is*

$$\Gamma, pc, hc \vdash cmd : t, hc', \Gamma', [\![cmd]\!],$$

we have

(1) $m(_\mathsf{pc}) \in pc$ *and* $m(_\mathsf{pc}) = m'(_\mathsf{pc})$

and the following invariants in the program execution

(2) $m(_\mathsf{hc}) \in hc$

(3) $m(x_{\mathsf{val}}) \in \Gamma_{\mathsf{val}}(x)$

(4) $m(x_{\mathsf{ctx}}) \in \Gamma_{\mathsf{ctx}}(x)$

Proof. The proof is by structural induction.

Base Cases:

Initially, we have $pc = hc = \{L\}$, $m(_\mathsf{pc}) = m(_\mathsf{hc}) = L$, similarly for x_{val} and x_{ctx}, so the conditions are true at the initial state.

Case $[\![$**skip**$]\!]$ is trivial.

Case $[\![x := e]\!] = (x, x_{\mathsf{val}}, x_{\mathsf{ctx}}) := (e, e_{\mathsf{val}}, e_{\mathsf{ctx}} \sqcup _\mathsf{pc})$

Since the execution of this code does not modify $_\mathsf{pc}$ or $_\mathsf{hc}$, we have (1) and (2).

For every variable t_{val} in expression e_{val}, we have that $m(t_{\mathsf{val}}) \in \Gamma_{\mathsf{val}}(t)$. Similarly, for every variable t_{ctx} in expression e_{ctx}, we have that $m(t_{\mathsf{ctx}}) \in \Gamma_{\mathsf{ctx}}(t)$. Hence, by the definition of the supremum on sets (Definition 3.3) and (S-Op), we have (3) (i.e., $m'(x_{\mathsf{val}}) \in \Gamma'_{\mathsf{val}}(x)$). We also have (4) due to the updated environment returned (S-Assign).

Case $[\![$**send** x **to** $c]\!]$

Subcase $[\![$**send** x **to** $c]\!]$ = **send** x **to** c

Trivial since no variable type is modified.

Subcase $[\![$**send** x **to** $c]\!]$ =

```
if _pc ⊔ _hc ⊔ x_val ⊔ x_ctx ⊔ c_ctx ⊑ c_val
then (send x to c)
else fail
end;
_hc := _pc ⊔ _hc ⊔ x_ctx ⊔ c_ctx;
```

Initially we have that $m(_\mathsf{pc}) \in pc, m(_\mathsf{hc}) \in hc, m(x_{\mathsf{ctx}}) \in \Gamma_{\mathsf{ctx}}(x)$ and $m(c_{\mathsf{ctx}}) \in \Gamma_{\mathsf{ctx}}(c)$. Only $_\mathsf{hc}$ is modified by this command, hence we have (1), (3) and (4). We also have (2) by the supremum on sets and the hc' returned by (S-GSend).

Induction Cases:

Case $[\![cmd_1; cmd_2]\!]$
We can use induction on $[\![cmd_1]\!]$, with the following hypothesis:

- $\Gamma, pc, hc \vdash cmd_1 : t_1, hc_1, \Gamma_1, [\![cmd_1]\!]$.

Induction hypothesis gives us that m'' (the memory after executing $[\![cmd_1]\!]$) and Γ_1 satisfy (1)-(4). If $t_1 = D$ then $[\![cmd_1; cmd_2]\!]$ is simply $[\![cmd_1]\!]$, and we are done since all executions diverge. If $t_1 \neq D$, then we have for $[\![cmd_2]\!]$ that:

- $\Gamma_1, pc, h_1 \vdash cmd_2 : t_2, h_2, \Gamma_2, [\![cmd_2]\!]$

Since the same program context as for cmd_1 is fed to the typing rule, then by the induction hypothesis and the fact that m'' and Γ_1 satisfy (1)-(4), we have the result for m' and Γ_2.

Case $[\![$**if** e **then** cmd_1 **else** cmd_2 **end**$]\!] =$

```
_oldpc^ν := _pc;
if e then
  _pc:=_pc ⊔ e_val ⊔ e_ctx;
  [[cmd_1]];
  update(mv_2)
else
  _pc := _pc ⊔ e_val ⊔ e_ctx;
  [[cmd_2]];
  update(mv_1)
end;
_pc := _oldpc
```

For (1) we observe that _pc has the same value before and after the instrumented code. Moreover, before cmd_i, $i = 1, 2$, the value of _pc is in pc', the context used to type these commands. For (3), by induction, we have that the invariants are true after the execution of $[\![cmd_1]\!]$ or $[\![cmd_2]\!]$. Hence we have (3). The update function in each branch inserts commands that, for every modified variable x, updates its x_{ctx} with _pc. This corresponds to what is done in the supremum of environments (Definition 3.4), so we have (4). Finally, since the update function updates the _hc with _pc when the termination behavior of the two branches can differ, and that the _hc is updated by update if one of the branch contains a guarded send, we have (2) as this corresponds to the union done in (S-If).

Case $[\![$**while** e **do** cmd **end**$]\!] =$

```
_oldpc^ν := _pc;
while e do
  _pc := _pc ⊔ e_val ⊔ e_ctx;
  [[cmd]];
end;
_pc := _pc ⊔ e_val ⊔ e_ctx;
update(mv);
_pc := _oldpc
```

The case is similar to the conditional. Variable _pc is updated at the beginning of each iteration, it belongs to $pc \sqcup \ell_e \sqcup \ell'_e$, as wanted for the induction step involving $[\![cmd]\!]$, and we get (1), as pc' contains all the possible values of _pc when $[\![cmd]\!]$ is executed. By induction, we have that the invariants are true after executing $[\![cmd]\!]$. Hence we have (3). We also have (2) and (4) using the same arguments as in the conditional.

□

The following proposition shows that any step of two executions performed from ℓ-equivalent memories results in ℓ-equivalent outputs. Theorem 3.1 follows as a corollary from it and from the next lemma.

Proposition 3.3 *Let m_i, $i = 1, 2$ be two ℓ-equivalent complete memories, o_i be a trace and $[\![cmd]\!]$ be a command generated by Figure 3.2, that is*

$$\Gamma, pc, hc \vdash cmd : t, h, \Gamma', [\![cmd]\!].$$

Then if both $[\![cmd]\!], m_i$ terminate, that is, we have maximal executions $\langle [\![cmd]\!], m_i, o_i \rangle \longrightarrow^* \langle \mathbf{stop}, m'_i, o'_i \rangle$, then the following statements are invariants:

(a) the memories are ℓ-equivalent (Def 3.5)

(b) ℓ-projections of observations are equal

Proof. The proof is by structural induction.
Base Cases:
Initially, we have m_i, $i = 1, 2$ two ℓ-equivalent memories where the level variables have been initialized, so (a) and (b) are straightforward.

Case $[\![$**skip**$]\!]$ is trivial.

Case $[\![x := e]\!] = (x, x_{\mathsf{val}}, x_{\mathsf{ctx}}) := (e, e_{\mathsf{val}}, e_{\mathsf{ctx}} \sqcup _\mathsf{pc})$

We have to prove (a), as (b) is trivial since this command does not modify the output traces. For ℓ-equivalence, since $_\mathsf{pc}$, $_\mathsf{hc}$ and the contents of channels are not modified, we obtain conditions (1) and (3) of Definition 3.5. For condition (2): since m_1 and m_2 are ℓ-equivalent, if $m'_1(x, x_{\mathsf{val}}, x_{\mathsf{ctx}}) \neq m'_2(x, x_{\mathsf{val}}, x_{\mathsf{ctx}})$, then either $m_i(x_{\mathsf{val}}) \not\sqsubseteq \ell$ (in which case m'_1 and m'_2 are still ℓ-equivalent) or $m'_i(x_{\mathsf{ctx}}) = m_i(_\mathsf{pc} \sqcup e_{\mathsf{ctx}}) \not\sqsubseteq \ell$ (by Lemma 3.1). Meaning that there is at least one variable t_{ctx} in e_{ctx} such that $m_i(t_{\mathsf{ctx}}) \not\sqsubseteq \ell$, or that $_\mathsf{pc} \not\sqsubseteq \ell$. Hence, m'_1 and m'_2 are still ℓ-equivalent. If $m'_1(x, x_{\mathsf{val}}, x_{\mathsf{ctx}}) = m'_2(x, x_{\mathsf{val}}, x_{\mathsf{ctx}})$, then m'_1 and m'_2 are ℓ-equivalent.

Case $[\![$**send** x **to** $c]\!]$

Subcase $[\![$**send** x **to** $c]\!] =$ **send** x **to** c

By the typing rule, we have

$$pc \sqcup hc \sqcup \ell_x \sqcup \ell'_x \sqcup \ell'_c \sqsubseteq_s \ell_c$$

using the property of supremum, combining with Lemma 3.2 and converting the notation, we obtain

$$\begin{aligned} m_i(_\mathsf{pc}) &\in pc \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \\ m_i(_\mathsf{hc}) &\in hc \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \\ m_i(x_{\mathsf{val}}) &\in \Gamma_{\mathsf{val}}(x) \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \\ m_i(x_{\mathsf{ctx}}) &\in \Gamma_{\mathsf{ctx}}(x) \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \\ m_i(c_{\mathsf{ctx}}) &\in \Gamma_{\mathsf{ctx}}(c) \sqsubseteq_s \Gamma_{\mathsf{val}}(c) \end{aligned}$$

By the definition of $\sqsubseteq_s$, this implies $m_i(_\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{val}} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \sqsubseteq m_i(c_{\mathsf{val}})$, which means that the sending of x on c is safe. Hence we obtain (a) and (b) by ℓ-equivalence of the memories.

Subcase $[\![$**send** x **to** $c]\!] =$

```
if _pc ⊔ _hc ⊔ x_val ⊔ x_ctx ⊔ c_ctx ⊑ c_val
  then (send x to c)
  else fail
end;
_hc := _pc ⊔ _hc ⊔ x_ctx ⊔ c_ctx;
```

We have to prove (a) and (b). For (a), we only have to prove conditions (1) and (3) as only the value of $_\mathsf{hc}$ and the content of the channel may change.

If $m_i(_\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \sqsubseteq \ell$ for one memory $i \in \{1, 2\}$ then, by ℓ-equivalence, it is also the case for the other memory and we get (1). It also implies that the two memory agree on x_{val} and c_{val} as they are ℓ-equivalent. This means that they either both fail, in which case

there is nothing more to prove, or the **send** command is executed in both. If it is executed, then $m_i(_\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{val}} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \sqsubseteq m_i(c_{\mathsf{val}})$, for $i = 1, 2$, and by ℓ-equivalence again, $o_i \upharpoonright \ell = o'_i \upharpoonright \ell$ in both executions and (b) is true, as well as (3).

If $m'_i(_\mathsf{hc}) = m_i(_\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \not\sqsubseteq \ell$ for $i = 1, 2$, then we get (1) and we have that the **send** command could succeed in m_1 and fail in m_2. But since $m_i(_\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}) \not\sqsubseteq \ell$, we know that it cannot succeed on a channel whose level is $\sqsubseteq \ell$ and we get (b) and (3).

Induction Cases:

Case $[\![cmd_1; cmd_2]\!]$
The hypothesis gives that m_1 and m_2 satisfy (a) and (b). Then we can use induction on $[\![cmd_1]\!]$, with the following hypotheses:

- $\Gamma, pc, hc \vdash cmd_1 : t_1, h_1, \Gamma_1, [\![cmd_1]\!]$.
- $\langle [\![cmd_1]\!], m_i, o_i \rangle \longrightarrow^* \langle \mathbf{stop}, m''_i, o''_i \rangle$

Induction hypothesis gives us that m''_1 and m''_2 satisfy (a) and (b). We have for $[\![cmd_2]\!]$ that:

- $\Gamma_1, pc, h_1 \vdash cmd_2 : t_2, h_2, \Gamma_2, [\![cmd_2]\!]$
- $\langle [\![cmd_2]\!], m''_i, o''_i \rangle \longrightarrow^* \langle \mathbf{stop}, m'_i, o'_i \rangle$

By the induction hypothesis and the fact that m''_1 and m''_2 satisfy (a) and (b), we obtain that m'_1 and m'_2 satisfy (a) and (b).

Case $[\![$**if** e **then** cmd_1 **else** cmd_2 **end**$]\!] =$

```
_oldpc^ν := _pc;
if e then
    _pc := _pc ⊔ e_val ⊔ e_ctx;
    [[cmd1]];
    update(mv2)
else
    _pc := _pc ⊔ e_val ⊔ e_ctx;
    [[cmd2]];
    update(mv1)
end;
_pc := _oldpc
```

Assume that m_1 and m_2 satisfy (a) and that $o_1 \upharpoonright \ell = o_2 \upharpoonright \ell$. If $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) = \sqcup_{t \in Var(e)} m_i(t_{\mathsf{val}} \sqcup t_{\mathsf{ctx}}) \sqsubseteq \ell$, $i = 1, 2$ (symmetry is given by ℓ-equivalence), we have the result by induction since both memories take the same branch, say i.

Now assume that $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \not\sqsubseteq \ell$, $i = 1, 2$. If $m_1(e) = m_2(e)$ we are in the same situation as above. So assume w.l.o.g. that $m_1(e)$ is true but $m_2(e)$ is false; hence under memory m_i, code $[\![cmd_i]\!]$ will be executed.

If both commands terminate, that is:

- $\langle [\![cmd_1]\!], m_1, o_1 \rangle \longrightarrow^* \langle \mathbf{stop}, m_1', o_1' \rangle$
- $\langle [\![cmd_2]\!], m_2, o_2 \rangle \longrightarrow^* \langle \mathbf{stop}, m_2', o_2' \rangle$

Then we know, by Lemma 3.3(c), that

$$o_1' \upharpoonright \ell = o_1 \upharpoonright \ell = o_2 \upharpoonright \ell = o_2' \upharpoonright \ell.$$

because before executing cmd_i, variable _pc is updated with the value of $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}})$ which is $\not\sqsubseteq \ell$. Hence, the execution of $[\![cmd_i]\!]$ will produce no output on channels of level $\sqsubseteq \ell$. Hence, (b) is proven for the induction step.

This also proves (a) on channels, but we need to prove it on variables (level and non level).

Let $x \in \mathcal{V}$ and $mv = mv_1 \cup mv_2$, where mv_i the set of variables that may be modified in cmd_i, $i = 1, 2$. By definition, variable _hc is also included in this set if the termination of the branches may differ or if one of them contains a guarded send. If $x \notin mv$ and the m_i's agree on x, x_{ctx} and x_{val}, then the m_i''s also agree on x, x_{ctx} and x_{val}.

If $x \in mv_i$, we have that $x_{\mathsf{ctx}} \sqsupseteq _\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}$ due to the commands inserted by the update function. Thus we have (a).

Case $[\![$while e do cmd end$]\!]$ =

```
_oldpc^ν := _pc;
while e do
  _pc := _pc ⊔ e_val ⊔ e_ctx;
  [[cmd]];
end;
_pc := _pc ⊔ e_val ⊔ e_ctx;
update(mv);
_pc := _oldpc
```

As a loop is essentially a (possibly infinite) sequence of **if**, the case is similar to the conditional. Assume that m_1 and m_2 satisfy (a) and (b). If $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) = \sqcup_{t \in Var(e)} m_i(t_{\mathsf{val}} \sqcup t_{\mathsf{ctx}}) \not\sqsubseteq \ell$, then we know that the execution of $[\![cmd]\!]$ will produce no output on channels of level $\sqsubseteq \ell$ since variable $_\mathsf{pc}$ is updated before entering the loop, and that every variable that is or could have been modified by $[\![cmd]\!]$ will have a t_{ctx} that is $\not\sqsubseteq \ell$ due to the update function. Hence, we have (a) and (b).

If instead we have that $m_i(e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \sqsubseteq \ell$, then we have the result by induction since both memories will always take the same branch.

□

Lemma 3.3. *With the premises of the previous theorem, if one step of $[\![cmd]\!]$ fails or diverges for one memory then it also fails or diverges for the other or no more output on channels of level ℓ or lower will be performed on that execution. More precisely, for $i = \{1, 2\}$*

(a) $m_i(_\mathsf{hc}) \sqsubseteq m'_i(_\mathsf{hc})$

(b) if only one execution fails or diverges, say m_1, then $o_2 \upharpoonright \ell = o'_2 \upharpoonright \ell$ and $m'_2(_\mathsf{hc}) \not\sqsubseteq \ell$.

(c) if $m_i(_\mathsf{pc}) \not\sqsubseteq \ell$ or $m_i(_\mathsf{hc}) \not\sqsubseteq \ell$ then $o_i \upharpoonright \ell = o'_i \upharpoonright \ell$

(d) if both executions fail or both diverge, then the ℓ-projections of observations are equal

Proof. (a) is straightforward since $_\mathsf{hc}$ is always included in the right-hand side when updating $_\mathsf{hc}$.

(b) is proven by induction, following the lines of Proposition 3.3. The interesting cases are the guarded send subcase and the inductive cases.

Case $[\![$send x to $c]\!]$
Let's assume that the send command is transformed into a guarded send. The only case where only one execution fails, say m_1, is one where $m'_2(_\mathsf{hc}) \not\sqsubseteq \ell$, and where no observation is made on a channel of level ℓ or lower, as wanted.

Case $[\![cmd_1; cmd_2]\!]$
Let m''_i and o''_i be the memories and output traces after executing cmd_1. Let m'_i and o'_i be the memories and output traces after executing $cmd_1; cmd_2$. If one execution fails or diverges, say m_1 on cmd_1, then by induction, $m''_2(_\mathsf{hc}) \not\sqsubseteq \ell$, and by (c), we obtain $o_2 \upharpoonright \ell = o''_2 \upharpoonright \ell$. By (a) and (c) and induction, we then get $o_2 \upharpoonright \ell = o'_2 \upharpoonright \ell$, as wanted. By (a), and (c) again, we also obtain $m'_2(_\mathsf{hc}) \not\sqsubseteq \ell$. If m_1 fails or diverges on cmd_2 instead, the argument is similar.

Case $[\![$if e then cmd_1 else cmd_2 end$]\!]$
There are two situations in which only one execution, say m_1, can fail: (1) only one execution executes a guarded send or (2) they both execute the same guarded send but the result is different. In case (1), we have that $m_i(_\mathsf{pc}) \not\sqsubseteq \ell$ (a consequence of Proposition 3.3 and Lemma 3.1), and that at least one of the **if**'s branch contains a guarded send, which is always followed by an update to variable $_\mathsf{hc}$. Since one of the branch modifies variable $_\mathsf{hc}$, it also means that the update function updates $_\mathsf{hc}$ to $_\mathsf{pc}$, which is $\not\sqsubseteq \ell$. Hence, we have (b) in this case. In case (2), the only way that the result of a guarded send can be different is if $m_i(_\mathsf{pc} \sqcup _\mathsf{hc} \sqcup \mathsf{x}_{\mathsf{ctx}} \sqcup \mathsf{c}_{\mathsf{ctx}}) \not\sqsubseteq \ell$, where x is the variable sent and c the channel on which the send occurs. In this case, the update to $_\mathsf{hc}$ that immediately follows the guarded send will ensure that $m_2(_\mathsf{hc}) \not\sqsubseteq \ell$. Hence we also have (b) in this case.

Similarly, for only only one execution to diverge, two things must be true: (1) $m_i(_\mathsf{pc}) \not\sqsubseteq \ell$, and (2) the termination type of the **if** command is M. Since the termination type is M, we have that the update function inserted after the conditional updates the $_\mathsf{hc}$ to $_\mathsf{pc}$, which is $\not\sqsubseteq \ell$. Hence, we also have (b) in this case.

Case $[\![$while e do cmd end$]\!]$
The argument is similar to the **if** command.

For (c), there are two cases where there are observations, the send case and the guarded send case. For the latter, the condition is taken care of by the guard. For the former, if $m_i(_\mathsf{pc}) \not\sqsubseteq \ell$ or $m_i(_\mathsf{hc}) \not\sqsubseteq \ell$ then, by Lemma 3.2 (1) and (2), one of the sets pc or hc contains a security level $\ell' \not\sqsubseteq \ell$. By the typing rule, this implies that $\ell' \sqsubseteq_s \Gamma_{\mathsf{val}}(c)$. By definition of $\sqsubseteq_s$, all elements of $\Gamma_{\mathsf{val}}(c)$ are greater than or equal to ℓ' and hence, again by Lemma 3.2 (3), $\ell' \sqsubseteq_s m_i(c_{\mathsf{val}})$ and $\ell \not\sqsubseteq_s m_i(c_{\mathsf{val}})$, thus $o_i \upharpoonright \ell = o'_i \upharpoonright \ell$.

Finally, for (d), we have three cases: both executions fail, both executions diverge and $m_i(_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \sqsubseteq \ell$ or both executions diverge and $m_i(_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \not\sqsubseteq \ell$, where e is the guard expression of the loop that diverges. If both executions fail, then there will be no more outputs and so the ℓ-projections remain equivalent. If both executions diverge and $m_i(_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \sqsubseteq \ell$, then by Lemma 3.1 we have that the ℓ-projections of observations are equal. If both executions diverge and $m_i(_\mathsf{pc} \sqcup e_{\mathsf{val}} \sqcup e_{\mathsf{ctx}}) \not\sqsubseteq \ell$, then the update to the $_\mathsf{pc}$ inside the body of the loop ensures that $m_i(_\mathsf{pc}) \not\sqsubseteq \ell$ before executing cmd, and so that there will be no more outputs on channels of level lower or equal to ℓ.

□

Lemma 3.4 (Fixed-point)**.** *The fixed-point computed by typing rules* (S-Loop1) *and* (S-Loop2) *always converges to a value.*

Proof. Let Γ, Γ' be typing environments and pc, pc' be sets of levels. We say that $(\Gamma, pc) \sqsubseteq (\Gamma', pc')$ if:

$$\begin{aligned}
&\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma') \wedge \\
&pc \subseteq pc' \wedge \\
&(\forall x \in \mathrm{dom}(\Gamma). \\
&\quad \Gamma_{\mathsf{val}}(x) \subseteq \Gamma'_{\mathsf{val}}(x) \wedge \Gamma_{\mathsf{ctx}}(x) \subseteq \Gamma'_{\mathsf{ctx}}(x))
\end{aligned}$$

Since this relation is a partial order (i.e., reflexive, transitive, antisymmetric), to prove that the computation of the fixpoint in (S-Loop2) always converges, we only need to prove that applying the rule iteratively builds a chain of environments that satisfy the relation. More precisely, we prove that the sequence $(\Gamma_0 \sqcup_{pc'_0} \Gamma'_0, pc'_0) \sqsubseteq (\Gamma_1 \sqcup_{pc'_1} \Gamma'_1, pc'_1) \sqsubseteq \ldots$ converges, where Γ_0 is the initial environment, Γ'_0 is the environment obtained after the first analysis of the loop body cmd, pc'_0, is the context in which the first analysis of cmd is performed. The result of analyzing cmd, the loop body, under Γ_i is Γ'_i. Note that $\Gamma_i = (\Gamma_{i-1} \sqcup_{pc'_{i-1}} \Gamma'_{i-1})$, is the environment under which the loop is recursively analyzed at step i.

By Definition 3.4, we have

$$\mathrm{dom}(\Gamma_{i-1} \sqcup_{pc'_{i-1}\Gamma'_{i-1}}) \subseteq \mathrm{dom}(\Gamma_i \sqcup_{pc'_i} \Gamma'_i).$$

Since $pc'_i = pc'_{i-1} \cup (pc'_{i-1} \sqcup \ell_{e_{i-1}} \sqcup \ell'_{e_{i-1}})$, where $\ell_{e_{i-1}}$ and $\ell'_{e_{i-1}}$ are the levels associated to the conditional expression of the loop at step $i-1$, we have $pc'_{i-1} \subseteq pc'_i$. Let $\Gamma = (\Gamma_{i-1} \sqcup_{pc'_{i-1}} \Gamma'_{i-1})$ and $\Gamma' = (\Gamma_i \sqcup_{pc'_i} \Gamma'_i)$. Let $x \in \mathrm{dom}(\Gamma)$.

Case x is not modified in the body of the loop :
We have that $\Gamma_{\mathsf{val}}(x) = \Gamma'_{\mathsf{val}}(x)$ and $\Gamma_{\mathsf{ctx}}(x) = \Gamma'_{\mathsf{ctx}}(x)$ by Definition 3.4.

Case x is modified in the body of the loop:
We have that $\Gamma_{\mathsf{val}}(x) \subseteq \Gamma'_{\mathsf{val}}(x)$ by Definition 3.4. Since Γ is calculated using pc'_{i-1}, we have that $\Gamma_{\mathsf{ctx}}(x) \sqcup pc'_{i-1} = \Gamma_{\mathsf{ctx}}(x)$ and since Γ' is calculated using pc'_i, we have that $\Gamma'_{\mathsf{ctx}}(x) \sqcup pc'_i = \Gamma'_{\mathsf{ctx}}(x)$. Hence, we have that $\Gamma_{\mathsf{ctx}}(x) \subseteq \Gamma'_{\mathsf{ctx}}(x)$ by $pc'_{i-1} \subseteq pc'_i$ and by Definition 3.4. Hence, a fixpoint will always be reached since $\mathcal{L}$ is finite. □

Theorem 3.2 *(Semantics preservation.) Let p be a program, m a memory, and o an output trace. Then*

$$\langle [\![p]\!], m, \epsilon\rangle \downarrow o \Rightarrow \ \langle p, \hat{m}, \epsilon\rangle \downarrow o$$
$$(\langle p, \hat{m}, \epsilon\rangle \downarrow o \wedge \langle [\![p]\!], m, \epsilon\rangle \downarrow o') \Rightarrow \ o \preceq o' \vee o' \preceq o$$

where $o' \preceq o$ means o' is a prefix of o.

Note that $\hat{m}$ is a memory for the source language, and hence it is of type : $\mathcal{V} \uplus \mathcal{C} \to \mathbb{Z} \uplus \mathcal{C}$ whereas m can, in addition, map variables to levels.

Sketch. By structural induction. The program generated by our instrumentation contains the same commands as the original program, in the same order. The only difference being the additional assignments on level variables and checks. For this reason, the only non-trivial case is the send command, since it modifies the output trace, or halts the program. Hence assume that cmd = **send** x **to** c. There are two cases: (1) $[\![cmd]\!] = cmd$ and (2) $[\![cmd]\!]$ = **if** $_\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{val}} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}} \sqsubseteq_s c_{\mathsf{val}}$ **then** (**send** x_1 **to** x_2) **else fail end**; $_\mathsf{hc} := _\mathsf{pc} \sqcup _\mathsf{hc} \sqcup x_{\mathsf{ctx}} \sqcup c_{\mathsf{ctx}}$.

In the first case, the claim is trivial. In the second case, the send is guarded by a condition. If this condition is true, the sending will happen and the output trace will be updated with $o :: (m(x_1), m(x_2))$, as would have been done by cmd. Otherwise, the program $[\![cmd]\!]$ will be stopped, and hence, no more output will happen, although cmd could produce other outputs. □

Chapter 4

Andrana: Quick and Accurate Malware Detection for Android

Authors: Andrew Bedford, Sébastien Garvin, Josée Desharnais, Nadia Tawbi, Hana Ajakan, Frédéric Audet and Bernard Lebel

Conference: *Foundations and Practice of Security*

Status: peer reviewed; published[1]; presented

Year: 2016

4.1 Résumé

Pour aider les utilisateurs à identifier les applications qui sont les plus susceptibles de faire fuir de l'information confidentielle (c.à.d., les applications malicieuses), nous présentons dans ce chapitre un outil de détection de maliciel pour Android appelé *Andrana*. Andrana exploite des techniques d'analyse statique et d'apprentissage automatique pour déterminer, avec une précision de 94.90%, si une application est malicieuse. Son analyse peut être faite directement sur un appareil mobile en moins d'une seconde et en utilisant seulement 12 Mo de mémoire. L'avantage principal d'Andrana comparativement aux antivirus est qu'il est capable d'identifier des applications malicieuses connues et inconnues, alors que les antivirus ne peuvent en général détecter que des applications malicieuses connues.

4.2 Abstract

In order to identify applications that are most likely to leak a user's sensitive information (i.e., malicious applications), we present in this chapter a malware detection tool for Android

[1]The published version is available at Springer via https://doi.org/10.1007/978-3-319-51966-1_2

called *Andrana*. Instead of relying on the more frequently used dynamic analysis, it leverages static analysis and machine learning techniques to determine, with an accuracy of 94.90%, if an application is malicious. Its analysis can be performed directly on a mobile device in less than a second and using only 12 MB of memory. Compared to antiviruses, which mostly use pattern matching algorithms to identify known malware (i.e., they look for specific sequences of instructions), Andrana's main advantage is that it can not only detect known malware and their variations, but also unknown malware.

4.3 Introduction

Android's domination of the mobile operating system market [1] has attracted the attention of malware authors and researchers alike. In addition to its large user base, what makes Android attractive to malware authors is that, contrarily to iOS users, Android users can install applications from a wide variety of sources such as first and third-party application markets (e.g., Google Play Store, Samsung Apps), torrents and direct downloads. Malware on mobile devices can be damaging due to the large amounts of sensitive information that they contain (e.g., emails, photos, banking information, location).

In order to protect users and their information, researchers have begun to develop malware detection tools specifically for Android. Traditional approaches, such as the signature-based and heuristics-based detection of antiviruses can only detect previously known attacks and hence suffer from a low detection rate. One possible solution is to use Machine Learning algorithms to determine which combinations of features (i.e. characteristics and properties of an application) are typically present in malware. These algorithms learn to detect malware by analyzing datasets of applications known to be malicious or benign.

The features used in Machine Learning are typically dynamically detected by executing the application in a sandbox (an isolated environment where applications can be safely monitored) where events are simulated [2, 3, 4]. This approach has two major problems, the first being the time needed. Analyzing each malware takes between 10 and 15 minutes (depending on the number of events sent to the simulator). The infrastructure required to keep such a tool up-to-date needs to be of considerable size, as more than 60 000 applications are added to Google's Play Store each month [5]. The second problem is that this approach cannot take into account all possible executions of the application, only those that happen in the time allocated. Furthermore, sophisticated malwares can exploit this fact by stopping their malicious behavior when they detect that the current execution is in a sandbox.

To address these issues, we built a new malware detection tool for Android called Andrana. It uses static analysis to detect features, and Machine Learning algorithms to determine if these features are sufficient to classify an application as a malware. Static analysis can be performed quickly and directly on a mobile device. This means that no sandbox and no exter-

nal infrastructure is required. Also, because static analysis considers all possible executions, it can detect attempts to evade analysis by the application. Andrana analyzes applications in three steps. First, the application is disassembled to obtain its code. Then, using static analysis, the application's features are extracted. Finally, a classification algorithm decides from the set of present features if the application is malicious.

One of the most important obstacle to static analysis is *obfuscation*. A code is obfuscated to make it hard to understand and analyze while retaining its original semantics. Although obfuscation has its legitimate uses (e.g., protection of intellectual property), it is often used by malware authors in an attempt to hide the malicious behaviors of their applications. Andrana can identify a number of obfuscation techniques and takes advantage of this information to improve the precision of its analysis.

In summary, our contributions in this chapter are:

- We introduce Andrana, a malware detection tool able to quickly and accurately determine if an application is malicious (Section 4.5).
- We present the set of features that Andrana uses to classify applications. It includes the obfuscation techniques used by the application (Section 4.6).
- We have trained and tested classifiers using different machine learning algorithms on a dataset of approximately 5 000 applications. Our best classifier has an accuracy of 94.90% and a false negative rate of 1.59%. (Section 4.7).
- We report on two of our experiments to improve the overall accuracy and usability of Andrana: (1) using string analysis tools to improve the detection rate of API calls, (2) executing Andrana on a mobile device (Section 4.8).

4.4 Background on Android

Before presenting Andrana, we must first introduce a few Android-related concepts and terminology, namely, the components of Android applications, Android's permission system and the structure of application packages.

4.4.1 Components

Android applications are composed of four types of components:

- **Activities**: An activity is a single, focused task that the user can do (e.g., send an email, take a photo). Applications always have one main activity (i.e., the one that is presented to the user when the application starts). An application can only do one activity at a time.

- **Services**: A service is an application component that can perform operations in the background (e.g., play music). Services that are started will continue to run in the background, even if the user switches to another application.

- **Intents**: An intent is a message that can be transmitted to another component or application. They are usually used to start an activity or a service.

- **Content Providers**: Content providers manage access to data. They provide a standard interface that allows data to be shared between processes. Android comes with built-in content providers that manage data such as images, videos and contacts.

4.4.2 Permissions

Android uses a permission system to restrict the operations that applications can perform. Android permissions are divided into two categories:

- **Normal**: Normal permissions are ones that cannot really harm the user, system or other applications (e.g., change the wallpaper) and are automatically granted by the system [6].

- **Dangerous**: Dangerous permissions are ones that involve the user's private information or that can affect the operation of other applications [7]. For example, the ability to access the user's contacts, internet or SMS are all considered to be dangerous permissions. These permissions have to be explicitly granted by the user.

4.4.3 Application Packages

Android applications are packaged into a single *.apk* file which contains:

- **Executable**: Android applications are written in Java and compiled to Java bytecode (*.class* files). The *.class* files are then translated to Dalvik bytecode and combined into a single Dalvik executable file named *classes.dex*.

- **Manifest**: Every Android application is accompanied by a manifest file, named *AndroidManifest.xml*, whose role is to specify the metadata of the application (e.g., title, package name, icon, minimal API version required) as well as its components and requested permissions.

- **Certificate**: Android applications must be signed with a certificate whose private key is known only to its developers. The purpose of this certificate is to identify (and distinguish) application authors.

- **Assets**: The assets used by the application (e.g., images, videos, libraries).

- **Resources**: They are additional content that the code uses such as user interface strings, layouts and animation instructions.

4.5 Overview of Andrana

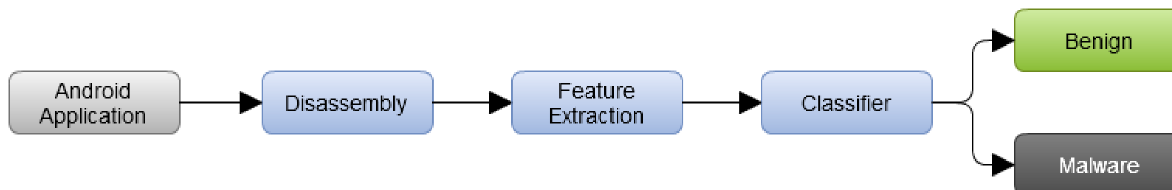

Figure 4.1: General flow of Andrana

Andrana analyzes applications in three steps: disassembly, feature extraction and classification (see Figure 4.1).

Step 1: Disassembly

To analyze the code of the application and extract its features, we must first disassemble it. Fortunately, Android applications are based on Java, which is easy to disassemble and decompile. Moreover, Java forces multiple constraints on the structure of the code, which prevents manipulations that could make static analysis less effective (e.g., explicit pointer manipulations).

To disassemble the application, we use a tool called Apktool [8]. It converts Dalvik bytecode into Smali [9], a more readable form of the bytecode.

Step 2: Feature Extraction

Once the application has been disassembled, its features are extracted using static analysis. These features, presented in Section 4.6, are characteristics and properties that the classifier will use in Step 3 to distinguish malicious from benign applications. It is the most computationally intensive step of the analysis.

Step 3: Classification

Finally, the detected features are fed to a binary classifier that classifies the application as either "benign" or "malware". To generate the most accurate classifier possible, we have tried a variety of Machine Learning algorithms (see Section 4.7). They were trained and tested on a dataset of approximately 5 000 applications.

The whole process takes on average 30 seconds and 280 MB of memory (see Figure 4.2) on a desktop computer (Intel Core i5-4200U with 4 GB of RAM). We were able to produce

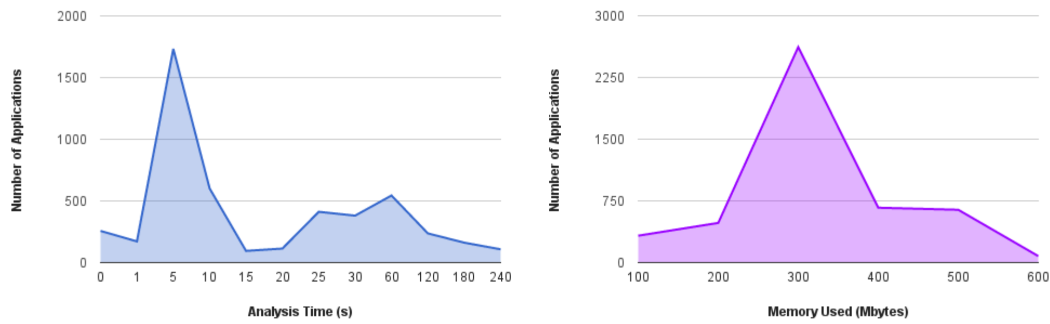

Figure 4.2: Analysis time and memory usage distributions.

an optimized Android version which utilizes a reduced set of significant features and whose analysis takes on average less than a second and 12 MB of memory (see Section 4.8).

4.6 Feature Extraction

In this section, we present the features extracted by Andrana. These features characterize the behavior of an application and are used by the classifier to determine if an application is malicious or benign. In addition to the features that are typically extracted in similar tools (see Section 4.9), such as requested permissions and API calls, Andrana also detects a number of obfuscation techniques and tools used by the application.

4.6.1 Features Extracted from the Manifest and Certificate

Requested Permissions

We extract the permissions requested by the application from the manifest file. Certain combinations of requested permissions can be indicative of a malicious intent. For example, an application that requests permissions to access the microphone and start a service could be covertly listening in on conversations.

Components

From the manifest file, we extract the application's components and determine if the application executes code in the background, which intents it listens to and which content providers it accesses.

Invalid Certificate

We verify the validity of certificates using a utility called *jarsigner* [10]. An invalid certificate indicates that the application has been tampered with.

4.6.2 Features Extracted from the Code

API Calls

We extract API calls from the code and, when possible, we also extract the value of their parameters. The latter are useful, for example, when trying to detect the attempt to send an email. This is done by looking for the function call `Activity.startActivity("act=android.intent.action.sendto dat=mailto:")`. Andrana considers that this feature is present if there is a call to this function and a string containing the value `"act=android.intent.action.sendto dat=mailto:"` somewhere in the code.

Necessary Permissions

By analyzing the API functions used in the code, we extract the permissions that are actually necessary to run the application. This allows to detect incongruities between the permissions requested by the application and those needed. Missing permissions could indicate that the application uses a root exploit to elevate its privileges during execution. To extract this information, Andrana uses an exhaustive mapping between the API calls and their required permissions. This mapping, which Google does not offer, is generated using PScout [11]. Note that since PScout's mapping is only an approximation, it may lead to false positives (i.e., reporting that there are missing permissions when, in fact, it is not true).

Obfuscation Techniques Used

We identify the obfuscation techniques possibly used by the application. The common techniques are: renaming, reflection, encryption and dynamic loading [12]. Note that their presence does not necessarily mean that the application uses obfuscation, only that it *may* have. It is the role of the learning algorithm to consider this feature as important or not.

Renaming A simple way to obfuscate a code is to rename its packages, classes, methods and variables. For example, a class "Car" could be renamed "diZx9cA" or "इटीपरीक्षा" (Java supports unicode characters). This technique is particularly effective against human analysts as the purpose of a class or method has to be guessed from its content. It also makes it harder to recognize the method elsewhere in the code.

To detect the use of renaming, we exploit the fact that class names usually contain common names (e.g., File, Car, User) and methods contain verbs (e.g., getInstance, setColor). Knowing this, the first strategy of Andrana is to look for classes that have single-letter names (e.g., b.class). If there are many of them, then we assume that renaming has been used. If none are found, then we use an n-gram-based language detection library [13] to detect the language used to name the classes and functions of the application. If the result varies widely across the application, then we assume that renaming has been used.

Reflection Reflection refers to the ability of the code to inspect itself at runtime. It can be used to get information on available classes, methods, fields, etc. More importantly, it can also be used to instantiate objects, invoke methods and access fields at runtime, without knowing their names at compile time. For example, using reflection, an instance of `ConnectivityManager` is created and method `getActiveNetworkInfo` is invoked in Listing 4.1.

```
Class c = Class.forName("android.net.ConnectivityManager");
Object o = c.newInstance()

Method m = c.getDeclaredMethod("getActiveNetworkInfo", ...);
method.invoke(o, null);
```

Listing 4.1: Instantiating an object and calling a function using reflection

The use of reflection itself can be detected easily, by looking for standard reflection API calls.

Encryption Encryption can be used to obfuscate the strings of the code. For instance, it could be used to statically hide the names of classes instanced using reflection, as in the following listing.

```
String className = decrypt(encryptedClassName);
Class c = Class.forName(className);
```

Listing 4.2: Instanciating an object of a statically unknown class using reflection

To detect the possible use of encryption, Andrana looks for standard cryptography API calls.

Dynamic Loading Java's reflection API also allows developers to dynamically load code (.apk, .dex, .jar or .class files). This code can be hidden in encrypted/compressed assets or data arrays. However, to load this code, applications must use Android's API `getClassLoader` function. Once loaded, the reflection API must be used to access the classes, methods and fields of the dynamically loaded code. Android applications can also dynamically load native libraries through the *Java Native Interface* (JNI). This not only allows Java code to invoke native functions, but also native code to invoke Java functions. According to Zhou et al. [14], approximately 5% of Android applications invoke native code.

To detect the use of dynamic loading, Andrana looks for instances of classes `DexClassLoader` and `ClassLoader`. To detect the use of native libraries, we look for calls to the API `System.loadLibrary`. Note that, for the moment, Andrana only detects the use of dynamic loading and native libraries: the libraries are not analyzed.

Commercial Obfuscation Tools

While developers can manually obfuscate the code themselves, most of them use commercially available obfuscation tools. Andrana is able to detect the use of these tools using the techniques

described by Apvrille and Nigam [15]. The obfuscation tools that are currently detected are ProGuard, DexGuard and APKProtect.

- ProGuard renames packages, classes, methods and variables using either the alphabet (default behavior) or a dictionnary of words. ProGuard comes with the Android SDK and runs automatically when building an application in release mode. As such, it is the most popular obfuscation tool. Andrana can detect the use of ProGuard by looking for strings such as "a/a/a->a" in smali code.
- DexGuard is the commercial version of ProGuard. It also renames the packages, classes, methods and variables, but uses by default non-ASCII characters which reduces even more the readability of the code. It also encrypts the strings present in the code. Andrana detects the use of DexGuard by looking for names that contain non-ASCII characters.
- APKProtect can be detected by searching for the string `"apkrotect"` in the *.dex* file.

Sandbox Detection

Certain malwares have the ability to deactivate their malicious behaviors when they detect that they are in a sandbox. This may indicate a malicious intent, as it could invalidate the results of a dynamic analysis. It does not affect static analyses, of course.

To detect the use of sandbox detection, we look for strings whose values are typically present in Android sandboxes. Vidas and Christin [16] enumerate some of the most common ones.

Disassembly Failure

While disassembly works in most cases, it can sometimes fail. Disassembly failure clearly indicates an attempt to thwart analysis. For this reason, it is part of our feature set.

4.7 Classification and Evaluation

In this section, we evaluate the performance of multiple Machine Learning algorithms.

4.7.1 Dataset

To train and test our algorithms, we have collected and analyzed a dataset of approximately 5 000 applications, 80% of which were malware. To avoid overfitting, the malware samples were randomly selected from two repositories: Contagio [17] and Virus Share [18]. The benign samples came from Google's Play Store various "Top 25". We noted that 47% of the samples used some kind of obfuscation.

Our dataset contains more malware samples than benign samples for two reasons. The main reason is that it is hard to obtain benign applications. Indeed, while there are many repositories of malicious Android applications, we found none that contained certified benign applications. Had we taken a larger number of applications from the Play Store, we would have risked introducing malicious samples into our dataset of benign samples. Another reason for using more malware samples is that it has a desirable side effect on the learning algorithm: it will lead the algorithm to try to make fewer bad classifications on this class. Hence, the number of false negatives (i.e., applications classified as "benign" when they are in fact malicious) will be naturally lower than the number of false positive.

4.7.2 Learning Algorithms

In order to obtain the best classifier possible, we have experimented with different learning algorithms: Support Vector Machines (SVM), k-Nearest Neighbors (KNN), Decision Trees (DT), Adaboost and Random Forest (RF).

Support Vector Machines (SVM) [19] is a learning algorithm that finds a maximal margin hyperplane in the vector space induced by the examples. The SVM can also take into account a *kernel function*, which encodes a notion of similarity between examples. Instead of producing a linear classifier in the *input space*, the SVM can produce a linear classifier in the space induced by the chosen kernel function. In our experiments, we use the *Radial Basis Function* (*RBF*) kernel $k(x, x') = e^{-\gamma||x-x'||_2^2}$, where γ is a parameter of the kernel function. The SVM also considers a hyperparameter C that controls the trade-off between maximizing the margin and permitting misclassification of training examples.

k-Nearest Neighbors (KNN) [20] is a learning algorithm that classifies a new data point by considering the k most similar training examples and by choosing the most frequent label among these examples. Here, k is a hyperparameter of the algorithm: different values of k might give different results. The most similar examples are computed using any similarity function, such as the Euclidean distance.

Decision Tree (DT) [21] is a learning algorithm that classifies examples by applying a decision rule at each internal node. The label of the example is decided at a leaf of the tree. Decision trees are learned by considering a measure of quality for a split such as the Gini impurity or the entropy for the information gain. In our experiment, we use the Gini impurity.

Adaboost [22] is an *ensemble classifier*, that considers many *base classifiers* and learns a weighted combination of these classifiers. At each iteration, a new base classifier is chosen (or generated) to focus on examples that are incorrectly classified by the current weighted

combination. The algorithm usually stops after a fixed number of iterations, or when the maximum number of base classifiers is attained. This maximum number of base classifiers is a hyperparameter of the algorithm.

Random Forest (RF) [23] is, similarly to adaboost, an ensemble classifier. It builds a majority vote of decision tree classifiers, by considering sub-samples of the data and by controlling the correlation between the trees. The number of trees or tree construction parameters such as the maximal depth are hyperparameters of the algorithm.

4.7.3 Performance Metrics

To evaluate the performance of the resulting classifiers, we measured their True Positive Ratio (TPR). It represents the proportion of malware applications that are correctly classified:

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

where TP is the number of malware applications that are correctly classified and FN is the number of malware applications that are classified as “benign”. Similarly, we measured their True Negative Ratio (TNR), which represents the proportion of benign application that are correctly classified:

$$\text{TNR} = \frac{\text{TN}}{\text{TN} + \text{FP}}$$

where TN is the number of benign applications that are correctly classified and FP is the number of benign applications that are classified as “malware”. Finally, we measured their overall accuracy, which represents the proportion of applications that are correctly classified:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}.$$

4.7.4 Evaluation

According to Hoeffding’s bound [24], with at least 600 test samples, the real risk is almost equal to the risk on test with 95% confidence. Hence, we chose to use the following splitting scheme in our experiments: 2/3 ($\sim$ 3300 samples) for the training set and 1/3 ($\sim$ 1700 samples) for the testing set. For each algorithm, we chose the hyperparameters using a 5-folds cross-validation on the training set and chose the hyperparameter values that optimized the accuracy. Table 4.1 shows the hyperparameter values on which the cross-validation was performed for each algorithm. Finally, we trained the algorithm using the whole training set, and predicted the examples of the testing set. Note that all reported values are metrics calculated on the testing set, containing examples that have not been seen during training time. Table 4.2 shows the resulting accuracies for each algorithm.

We now discuss on whether an increase in the size of the training dataset can possibly improve the learning algorithms’ performance. For this experiment, we first split the dataset into a

Learning Algorithm	Hyperparameter	Values
SVM	C	$\{0.001, 0.01, 0.1, 1, 10, 100, 1000\}$
	γ	$\{100, 10, 1.0, 0.1, 0.01, 0.001, 0.0001\}$
KNN	k	$\{1, 2, 3, 4, 5, 10, 15, 20, 25, 50, 100\}$
Decision Trees (DT)	max_leaf_nodes	$\{5, 10, 15, 20, 25, 30, 40, 50\}$
	$min_samples_leaf$	$\{1, 2, 3, 5, 10, 20\}$
AdaBoost	`n_estimators`	$\{5, 10, 25, 50, 100, 250, 500, 1000\}$
RandomForest (RF)	`n_estimators`	$\{2, 5, 10, 25, 50, 100, 500, 1000, 2000, 3000\}$

Table 4.1: Considered values for each hyperparameter, for each algorithm.

Learning Algorithm	Accuracy%	TPR%	TNR%
SVM	94.72	**98.64**	78.43
KNN	94.11	97.74	79.06
Decision Trees (DT)	93.20	97.43	75.62
AdaBoost	94.11	98.26	76.87
RandomForest (RF)	**94.90**	98.41	**80.31**

Table 4.2: A comparison of the classifiers' metrics, Accuracy, True Positive Ratio and True Negative Ratio, using different machine learning algorithms.

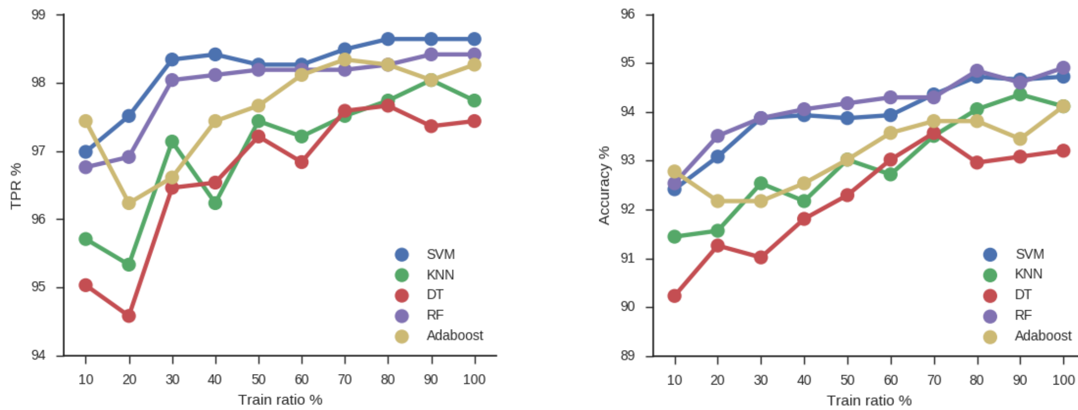

Figure 4.3: The progression of true positives ratio and accuracy on the test set for different ratios of training set and for each learning algorithm. It is calculated using the best configuration of hyperparameters outputted by a 5-fold cross-validation.

training set (2/3) and a test set (1/3). Then, we followed the same procedure as above, but applied exclusively to a ratio of the training set and without altering the test set. Figure 4.3 shows that an increase of the training ratio leads to a fluctuating improvement of the accuracy. The non-monotonous behavior of the accuracy is a common occurrence in statistical learning and is mainly caused by noise in the dataset. Still, one can see that the accuracy tends to increase when the training ratio increases. So, we can expect a higher performance by using a larger dataset.

4.8 Additional Experiments

This section presents the various experiments that we did in order to improve the overall accuracy and usability of Andrana.

4.8.1 E1: Using String Analysis Tools

As previously mentionned, API calls can be invoked through reflection. To detect those calls, we look for their class and method names in the strings of the code. Of course, strings are not necessarily hard coded, they can also be dynamically built. For instance, in Figure 4.3, the class instantiated could be either `"java.lang.String"` or `"java.lang.Integer"`.

```java
String a = "java.lang.";
String b;
if (random) { b = "String"; } else { b = "Integer"; }
String className = a + b;
Class c = Class.forName(className);
Object o = c.newInstance();
```

Listing 4.3: Dynamically built class name

To take into account cases where the class and/or function names are dynamically created, we have experimented with a tool called *Java String Analyzer* (JSA) [25, 26]. JSA performs a static analysis of Java programs to predict the possible values of string variables. This allows us to determine that the possible values for the string variable `className` in Listing 4.3 are {`"java.lang.String", "java.lang.Integer"`}.

However, JSA is not able to analyze entire Android applications in a reasonable time or without running out of memory. Li et al. [27] encountered similar problems with JSA and hypothesize that this problem is due to the fact that it uses a variable-pair-based method to do the global inter-procedural aliasing analysis. This method has an $O(n^2)$ memory complexity, where n is the number of variables in the application.

We have also experimented with another string analysis tool called *Violist* [27]. While it is considerably faster than JSA and can actually be used to analyze Android applications, it still requires too much time (around 4 minutes) and resources (up to 2.4 GB of memory) for our purpose: the analysis has to be executable on a mobile device. Furthermore, in our test on 10 applications that used reflection, it did not lead to the detection of additional features. For these reasons, we chose to not use them in Andrana. Besides, as seen in the previous section, it turns out that a precise string analysis is not required to accurately classify applications. This is because Andrana uses a wide variety of features to classify applications, some of which are not affected by obfuscation techniques (e.g., permissions, certificate, disassembly failure).

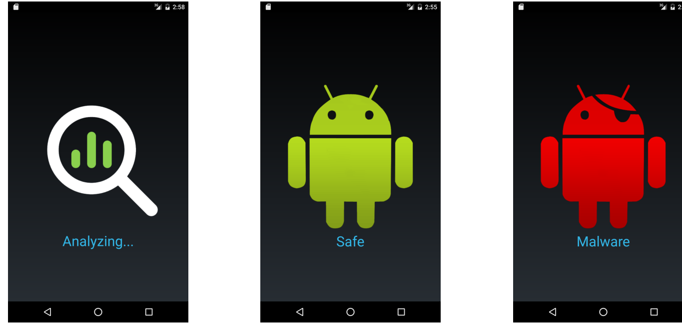

Figure 4.4: Andrana's interface on Android.

4.8.2 E2: Executing Andrana on a Mobile Device

Mobile devices generally have low computing power and memory compared to desktop computers. Consequently, if Andrana is to run directly on such devices, it must be very efficient. To evaluate Andrana's runtime performance on mobile devices, we have implemented a version of it for Android (see Figure 4.4). In order to minimize Andrana's analysis time on Android, we chose to use the decision tree as the classifier. As previously shown, it is accurate (93.20%) and requires only a small subset of our features to classify applications (between 3 and 9 features). We also optimized Andrana's Android version so that it uses as little memory as possible. We analyzed 150 randomly selected applications from our dataset on a Nexus 5X and, on average, the analysis took only 814 milliseconds and 12 MB of memory, much quicker than our desktop version which extracts *all* features. Besides its performance, another advantage of using the decision tree classifier is that it is simple to understand and interpret.

4.9 Related Work

Research on malware detection tools for Android has been very active in recent years. We present in this section the approaches that are most similar to ours.

Static Malware Detection Tools for Android

Sato et al. [28] present a method to calculate the malignancy score of an application based entirely on the information found in its manifest file. Namely, the permissions requested, intent filters (their action, category and priority), number of permissions defined and application name. They trained their classifier on a dataset of 365 samples and report an accuracy of 90%.

Aafer et al. [29] present a classifier, named DroidAPIMiner, that uses the API calls present in the code of the application to determine whether an application is benign or malicious. To determine the most relevant API calls for malware detection, they statically analyzed a

large corpus of malware and looked at the most frequent API calls. They report a maximum accuracy of 99% using a KNN classifier.

Arp et al. [30] present another classifier, named Drebin, which uses statically detected features. Namely, they extract the hardware components (e.g., GPS, camera, microphone) used by the application by looking at the permissions requested in the manifest file, the requested permissions, the API calls present in the code, IP addresses and URLs found in the code. They use the SVM machine learning algorithm to produce a classifier. It has an accuracy of approximately 94%. Their Android implementation requires, on average, 10 seconds to return a result.

Since the datasets used in these approaches are not actually available for analysis, we cannot directly compare their performance with Andrana's. We also do not know if their samples were as heavily obfuscated as ours. All we can say is that Andrana seems to equal them in terms of accuracy and surpass them in terms of speed. We expect that by using a larger dataset of applications, like the 20 000 used by DroidAPIMiner, we could improve even more our accuracy. So that others may compare their results with ours, our dataset is available online [31].

There are also various antiviruses available on Google's Play Store (e.g., AVG, Norton, Avira). Antiviruses mostly use pattern matching algorithms to identify known malware (i.e., they look for specific sequences of instructions). This means that different patterns must be used to detect variations of the same malware. Andrana's main advantage over antiviruses is that it can not only detect known malware and their variations, but also unknown malware.

Dynamic and Hybrid Malware Detection Tools for Android

Crowdroid [32], Andromaly [33] and MADAM [34] detect malware infections by looking for anomalous behavior. To detect anomalies, they monitor system metrics such as CPU consumption, number of running processes, number of packets sent through WiFi and/or the API calls performed at runtime by an application. Machine learning techniques are then used to distinguish standard behaviors from those of an infected device.

DroidRanger [14] use both static and dynamic analysis to perform a large-scale study of several application markets. Instead of using machine learning techniques to automatically learn to classify applications, they use a variety of heuristics. Using their tool, they were able to identify 211 malicious applications present on the markets, 32 of which were on Google's Play Store.

DroidDolphin [35] inserts a monitor into applications to log their API calls and then executes them. The authors generate a classifier using this information and a dataset of 34 000 applications. They report an accuracy of 86.1%.

Andrubis [4] and its successor Marvin [3] uses approximately 500 000 features, detected using a combination of static and dynamic analyses, to train and test their classifier on a dataset of over 135 000 applications. They report an accuracy of 98.24%.

Andrana's main advantages over these approaches are that it introduces no runtime overhead and that its analysis can be performed on the user's mobile device, very quickly.

4.10 Conclusion

In this chapter, we have presented Andrana, a lightweight malware detection tool for Android. It uses static analysis to extract an application's features and then uses a classifier to determine if it is benign or malicious. We have trained and tested multiple classifiers using a variety of Machine Learning algorithms and a dataset of approximately 5 000 applications, 4 000 of which were malwares. The dataset is available online [31]. Its samples came from multiple sources to avoid overfitting. Our best classifier has an accuracy of 94.90% and a false negative rate of 1.59%, which is comparable to other similar tools. As indicated by the upward trends of Figure 4.3, the use of larger datasets should lead to even higher accuracies.

As almost half of our dataset used reflection, we considered using two string analysis tools, JSA and Violist, to improve the detection rate of our features, but their use turned out to be computationally too expensive for our purpose.

We have implemented a version of Andrana for Android and our tests reveal that, on average, it can analyze applications in less than a second using only 12 MB of memory, faster and more efficiently than any similar tools. Since our implementation uses a decision tree as its classifier, users can easily understand what lead the application to be classified as malware/benign.

Acknowledgments

We would like to thank François Laviolette for his suggestions and Souad El Hatib for her help with the string analysis tools. This project was funded by Thales and the NSERC.

4.11 Bibliography

[1] "Smartphone os market share, q1 2015," 2015, accessed July 7, 2016. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] A. Atzeni, T. Su, M. Baltatu, R. D'Alessandro, and G. Pessiva, "How dangerous is your android app?: an evaluation methodology," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2014, pp. 130–139.

[3] M. Lindorfer, M. Neugschwandtner, and C. Platzer, "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis," in *Computer Software and Applications Conference (COMPSAC), 39th Annual*, vol. 2. IEEE, 2015, pp. 422–433.

[4] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, "Andrubis–1,000,000 apps later: A view on current android malware behaviors," in *2014 Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*. IEEE, 2014, pp. 3–17.

[5] "Appbrain," accessed July 18, 2016. Available: http://www.appbrain.com/stats/number-of-android-apps

[6] "Android operating system security," accessed July 5, 2016. Available: http://developer.android.com/guide/topics/security/permissions.html

[7] "Permissions classified as dangerous," accessed July 5, 2016. Available: http://developer.android.com/guide/topics/security/permissions.html#normal-dangerous

[8] "Apktool," accessed July 5, 2016. Available: https://ibotpeaches.github.io/Apktool/

[9] "Smali/baksmali," accessed July 20, 2016. Available: https://github.com/JesusFreke/smali

[10] "jarsigner," accessed September 13, 2018. Available: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html

[11] "Pscout," accessed July 5, 2016. Available: https://github.com/dweinstein/pscout

[12] D. Maiorca, D. Ariu, I. Corona, M. Aresu, and G. Giacinto, "Stealth attacks: An extended insight into the obfuscation effects on android malware," *Computers & Security*, vol. 51, pp. 16–31, 2015.

[13] "Language detection library," accessed July 5, 2016. Available: https://github.com/shuyo/language-detection

[14] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *NDSS*, vol. 25, 2012, pp. 50–52.

[15] A. Apvrille and R. Nigam, "Obfuscation in android malware, and how to fight back," *Virus Bull*, pp. 1–10, 2014.

[16] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, 2014, pp. 447–458.

[17] “Contagio,” accessed July 16, 2016. Available: http://contagiominidump.blogspot.ca/

[18] “Virus share,” accessed July 14, 2016. Available: https://virusshare.com/

[19] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. Available: http://dx.doi.org/10.1007/BF00994018

[20] P. Cunningham and S. J. Delany, “k-nearest neighbour classifiers,” *Multiple Classifier Systems*, pp. 1–17, 2007.

[21] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[22] R. E. Schapire and Y. Singer, “Improved boosting using confidence-rated predictions,” *Machine Learning*, vol. 37, no. 3, pp. 297–336, 1999.

[23] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[24] W. Hoeffding, “Probability inequalities for sums of bounded random variables,” *Journal of the American statistical association*, vol. 58, no. 301, pp. 13–30, 1963.

[25] A. S. Christensen, A. Møller, and M. I. Schwartzbach, *Precise analysis of string expressions*. Springer, 2003.

[26] “Java string analyzer (jsa),” accessed July 5, 2016. Available: http://www.brics.dk/JSA/

[27] D. Li, Y. Lyu, M. Wan, and W. G. Halfond, “String analysis for java and android applications,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 661–672.

[28] R. Sato, D. Chiba, and S. Goto, “Detecting android malware by analyzing manifest files,” *Proc. of the Asia-Pacific Advanced Network*, vol. 36, pp. 23–31, 2013.

[29] Y. Aafer, W. Du, and H. Yin, “Droidapiminer: Mining api-level features for robust malware detection in android,” in *Sec. and Priv. in Comm. Networks*. Springer, 2013, pp. 86–103.

[30] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, K. Rieck, and C. Siemens, “Drebin: Effective and explainable detection of android malware in your pocket,” in *Proc. of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[31] “Lsfm,” accessed September 30, 2016. Available: http://lsfm.ift.ulaval.ca/recherche/andrana/

[32] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, “Crowdroid: behavior-based malware detection system for android,” in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[33] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "“andromaly”: a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[34] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: a multi-level anomaly detector for android malware," in *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security.* Springer, 2012, pp. 240–253.

[35] W.-C. Wu and S.-H. Hung, "Droiddolphin: a dynamic android malware detection framework using big data and machine learning," in *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems.* ACM, 2014, pp. 247–252.

Chapter 5

Information-Flow Control with Fading Labels

Author: Andrew Bedford

Conference: *International Conference on Privacy, Security and Trust (PST)*

Status: peer reviewed; published[1]; presented

Year: 2017

5.1 Résumé

Les mécanismes de contrôle des flots d'information existants utilisent généralement la même quantité de ressources pour protéger les informations de divers niveaux. Toutefois, dans les systèmes où les ressources sont plus limitées, il peut être plus approprié d'utiliser plus de ressources pour protéger l'information qui est plus importante, et moins de ressources pour protéger l'information qui est moins importante. Pour adresser ce problème, nous introduisons dans ce chapitre le concept d'étiquettes disparaissente (*fading labels* en anglais).

Les étiquettes disparaissente sont des étiquettes qui cessent de se propager après un nombre fixe d'utilisation. En paramétrant les mécanismes de façon que les étiquettes qui sont associées à de l'information importante disparaissent plus lentement que celles associées à de l'information moins importante, elles permettent aux mécanismes d'utiliser plus de ressources pour traquer et contrôler l'information importante. Les mécanismes qui utilisent ces étiquettes n'appliqueront pas toujours la noninterférence étant donné que des fuites d'information peuvent se produire une fois que l'étiquette a disparu. Ce que ces mécanismes appliquent comme politique de sécurité est plutôt une variation de la noninterférence que nous avons nommée la *depth-limited noninterference* et que nous introduisons dans ce chapitre.

[1] The published version is available via http://doi.ieeecomputersociety.org/10.1109/PST.2017.00053

5.2 Abstract

Existing information-flow control mechanisms usually invest the same amount of resources to protect information of varying importance. However, in systems where resources are limited, it may be more appropriate to spend more resources to protect information that is more important (e.g., passwords), and less resources to protect information that is less important (e.g., current location). To address this issue, we introduce the concept of *fading labels*.

Fading labels are labels whose taint stops propagating after a fixed amount of uses (i.e., they fade away). By parameterizing mechanisms so that labels associated to important information fade away more slowly than those associated to less important information, they allow mechanisms to use more resources to track important information than other information. Since leaks of information may occur once a label fades away, mechanisms that use them may not always enforce noninterference. What they do enforce is a relaxed version of noninterference, called *depth-limited noninterference*, which we also introduce in this chapter.

5.3 Introduction

Information-flow control mechanisms are mechanisms that enforce information-flow policies (e.g., information from a top secret file should not be sent over the network). This is usually done by associating sensitive information with a label, which is then propagated whenever the information is used; a process called *tainting*.

These mechanisms, be they static [1], dynamic [2] or hybrid [3], generally invest the same amount of resources to protect information of varying importance. However, in systems where ressources are limited such as smartphones and tablets, it may be more appropriate to spend more resources to protect information that is more important (e.g., passwords), and less resources to protect information that is less important (e.g., current location).

For this reason, we introduce in this chapter the concept of *fading labels*. They are labels whose taint stops propagating after a fixed amount of uses. They allow mechanisms to use more resources to track important information than other information.

Contributions

- We introduce the concept of fading labels in Section 5.4 and a relaxed version of noninterference called depth-limited noninterference in Section 5.5.
- We provide an example of a mechanism that uses fading labels in Section 5.6.
- We discuss their advantages, disadvantages and possible variations in Section 5.7.

5.4 Fading Labels

To illustrate the concept, consider the program of Listing 5.1.

```
read value from privateFile;
w := 0;
x := value + 1;
x' := value + 2;
y := x mod 3;
z := y * 4;
write z to publicFile
```

Listing 5.1: Derived values

Normally, variable `value` and all of its derivatives (i.e., variables `x`, `x'`, `y`, `z`) would be tainted with `privateFile`'s label. This approach leads to the tainting of increasingly large portions of programs over time; a problem known as *taint creep*.

The more a program is tainted, the more resources will be needed by the mechanism. This is especially true if the mechanism needs to perform additional computations to prevent leaks through covert channels, such as calling a termination oracle to prevent progress leaks [4] or executing dummy operations to prevent timing leaks [5]. Hence, in order to reduce the amount of resources required, we chose to reduce the number of tainted variables. So that the security of sensitive information is not compromised too much, we chose to do so by limiting the *depth* at which a taint is propagated. For example, in Listing 5.1, variable `z` is derived from `y`, which is derived from `x`, which is derived from `value` (illustrated in Fig. 5.1). For this reason, relative to `value`, `x` is at depth 1, `y` is at depth 2 where `privateFile`'s label stops being propagated, and `z` is at depth 3. The idea is to parametrize mechanisms so that labels associated to important information are propagated more deeply than those associated to less important information.

For our purposes, we assume that the levels of information are organized in a finite lattice $(\mathcal{L}, \sqsubseteq)$ which contains at least two elements: L for the bottom of the lattice (least important) and H for the top of the lattice (most important), i.e. $\forall l \in \mathcal{L}, L \sqsubseteq l \wedge l \sqsubseteq H$. To each level $l \in \mathcal{L}$ is assigned an integer $maxDepth(l)$ representing the maximum propagation depth. Note that alternatively, the maximum propagation depth can be associated to channels of information rather than their level. So there could be a channel of level H and depth 5, and another one of depth 500.

We can formally define fading labels as sets of couples where the first element is the level of information and the second element is a counter that keeps track of the depth:

$$\textit{(fading labels)} \quad \ell ::= \mathcal{P}(\mathcal{L} \times \mathbb{N})$$

Each time the label is propagated, its counters are decremented. Once a counter reaches 0 (e.g., `y` in Fig. 5.1), then the couple is removed from the set. We use sets because in our context, a variable can be tainted with more than one element of the lattice. For example, if we have an assignment `a := b + c` where `b`:$\{(H, 8)\}$ and `c`:$\{(M,10)\}$, then `a`:$\{(H,7),(M,9)\}$. Note that if `c`:$\{(M,7)\}$, then `a`:$\{(H,7)\}$ because $M \sqsubseteq H$ and the remaining depth of H is greater or equal than M's.

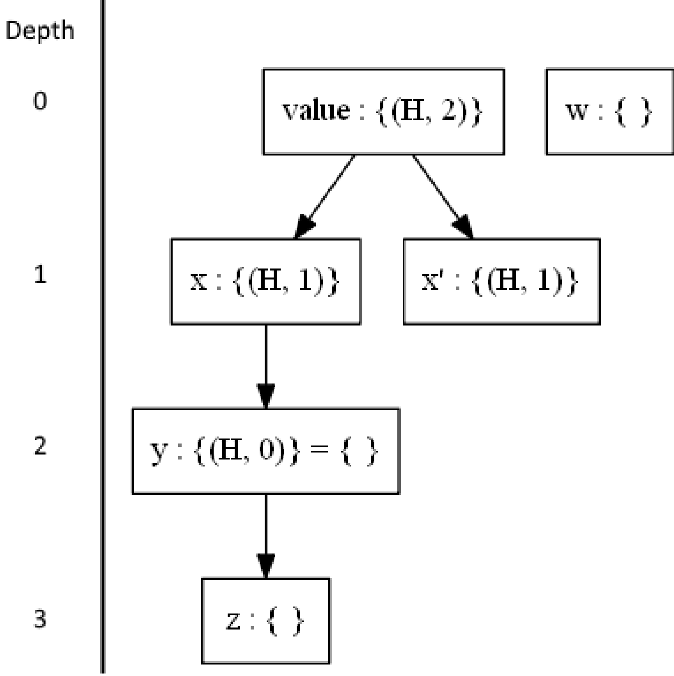

Figure 5.1: PDG-like representation of Listing 5.1

5.5 Depth-Limited Noninterference

Noninterference [6] is the security policy that is enforced by most information-flow mechanisms. Intuitively, it states that private information should not influence (interfere with) the publicly observable behavior of a program. More formally, it states that there should not be information-flows from inputs of level l_1 to outputs of level l_2 if $l_1 \not\sqsubseteq l_2$. Since fading labels stop propagating after a certain point to reduce resource-usage, mechanisms that use them do not necessarily satisfy noninterference. What they *do* satisfy is a relaxed version of noninterference that we call *depth-limited noninterference*.

In order to define depth-limited noninterference, we use *program dependence graphs* (PDG). They are a visual representation of information flows that can occur in a program. Each node represents a program statement or expression and there are two kinds of edges:

- Data dependence (a.k.a. explicit flows): An edge $x \longrightarrow y$ means that statement x assigns

a variable that is used in statement y.

- Control dependence (a.k.a. implicit flows): An edge $x \dashrightarrow y$ means that the execution of y depends of the value of expression x (typically the condition of an if/while command).

If there is a path from node x to node y, it means that information can flow from x to y. So if there are no paths from private inputs to public outputs, then the program is noninterferent. Consequently, PDG-based mechanisms such as Hammer et al. [7] enforce noninterference by searching for such paths, no matter their length. This would reveal that the program in Listing 5.1 does not satisfy noninterference as there is a path from `value` (private input) to `z` (public output).

Depth-limited noninterference is essentially the same thing, but the verified paths have a maximum length. For example, since the maximum depth of H-level information is set to 2 in Fig. 5.1, the program would satisfy depth-limited noninterference.

5.6 Example of a Mechanism

In this section, we present an example of a dynamic information-flow control mechanism that uses fading labels. The mechanism enforces policies on programs written in an imperative language that has commands for receiving and sending information. The language is based on the one used by Bedford et al. [8].

5.6.1 Syntax

Let $\mathcal{V}$ be a set of identifiers for variables, and $\mathcal{C}$ a set of predefined communication channels. The syntax is as follows.

$$\begin{array}{llll}
\textit{(variables)} & x & \in & \mathcal{V} \\
\textit{(channels)} & c & \in & \mathcal{C} \\
\textit{(integer constants)} & n & \in & \mathbb{Z} \\
\textit{(expressions)} & e & ::= & x \mid n \mid e_1 \textbf{ op } e_2 \mid \\
\textit{(commands)} & cmd & ::= & \textbf{skip} \mid x := e \mid cmd_1; cmd_2 \mid \\
 & & & \textbf{if } e \textbf{ then } cmd_1 \textbf{ else } cmd_2 \textbf{ end} \mid \\
 & & & \textbf{while } e \textbf{ do } cmd \textbf{ end} \mid \\
 & & & \textbf{read } x \textbf{ from } c \mid \textbf{write } x \textbf{ to } c
\end{array}$$

Values are integers, we use zero for false and nonzero for true. The symbol **op** stands for arithmetic or logic binary operators on integers.

We suppose that the interaction of a program with its environment (which can be a user or another program) is done through channels. Channels can be, for example, files, users,

network channels, keyboards, computer screens, etc. Without loss of generality, we consider that each channel consists of one value. Command **read** x **from** c reads the current value in channel c (without modifying the channel's content) and assigns its value to variable x. Command **write** x **to** c writes the value of variable x to channel c and overwrites the current value in the channel. The security levels of these channels is determined in advance by some security administrator. We use $\mathit{levelOfChan}(c)$ to denote the security level of channel c.

5.6.2 Semantics

A memory $m : \mathcal{V} \uplus \mathcal{C} \to \mathbb{Z}$ is a partial map from variables and channels to values, where the value of a channel is the last value sent to this channel. More precisely a memory is the disjoint union ($\uplus$) of two partial maps of the following form:

$$m_v : \mathcal{V} \to \mathbb{Z}, \qquad m_c : \mathcal{C} \to \mathbb{Z},$$

We omit the subscript whenever the context is clear. We write $m(e) = v$ to indicate that the evaluation of expression e under memory m returns v.

The semantics of the language is illustrated in Fig. 5.2. Note that the semantics of the sequence, skip, conditional and loop commands are omitted for brevity; they correspond to the usual definitions. Program configurations are tuples $\langle \mathit{cmd}, \Gamma, m, o\rangle$ where cmd is the command to be evaluated, Γ is the current typing environment (i.e., maps variables to labels), m is the current memory and o is the current output trace. A transition between two configurations is denoted by the $\longrightarrow$ symbol. We write $\longrightarrow^*$ for the reflexive transitive closure of the $\longrightarrow$ relation.

We write $v :: vs$ for sequences where v is the first element of the sequence, and vs is the rest of the sequence. We write ϵ for the empty sequence. An output trace is a sequence of output events: it is of the form $o = (v_0, c_0) :: (v_1, c_1) :: \ldots$ where $v_k \in \mathbb{Z}$ is an integer value, and $ch_k \in \mathcal{C}$ is a channel, $k \in \mathbb{N}$. The rule for writing a value appends a new output event to the end of the trace. We abuse notation and write $o :: (v, c)$ to indicate that event (v, c) is appended to trace o.

5.6.3 Mechanism

The techniques used to prevent explicit flows are well known and used here [9]. Preventing implicit flows is not necessary to illustrate the approach, hence they are not taken into account in this mechanism. This leaves three noteworthy rules:

(Read): Reads the current value in channel c and assigns its value to variable x. The label of x corresponds to the level of c and maximum propagation depth associated to the level of channel c.

$$(\textsc{Read})\ \frac{m(c) = v \qquad c \in \mathcal{C} \qquad \mathit{levelOfChan}(c) = l_c \qquad \mathit{maxDepth}(l_c) = d}{\langle \textbf{read}\ x\ \textbf{from}\ c, \Gamma, m, o\rangle \longrightarrow \langle \textbf{skip}, \Gamma[x \mapsto (l_c, d)], m[x \mapsto v], o\rangle}$$

$$(\textsc{Assign})\ \frac{\mathit{level}\Gamma, e = \ell_e \qquad \ell'_e = \{(l_e, d_e - 1) \mid (l_e, d_e) \in \ell_e \wedge d_e - 1 > 0\} \qquad m(e) = v}{\langle x := e, \Gamma, m, o\rangle \longrightarrow \langle \textbf{skip}, \Gamma[x \mapsto \ell'_e], m[x \mapsto v], o\rangle}$$

$$(\textsc{Write})\ \frac{m(x) = v \qquad c \in \mathcal{C} \qquad \mathit{levelOfChan}(c) = l_c \qquad l_x = \bigsqcup_{(l,d) \in \Gamma(x)} l \qquad l_x \sqsubseteq l_c}{\langle \textbf{write}\ x\ \textbf{to}\ c, \Gamma, m, o\rangle \longrightarrow \langle \textbf{skip}, \Gamma, m[c \mapsto v], o :: (v, c)\rangle}$$

Figure 5.2: Semantics of the read, assign and write commands when using fading labels

(Assign): Assigns the value of expression e to variable x. It also propagates the labels of variables found in expression e. Function $\mathit{level}\Gamma, e$ returns the union of the labels of all variables in expression e. This set is then filtered to remove elements which have faded away. That is, those for which the counter has reached 0.

(Write): Writes the current value of x in channel c, provided that x's level is lower than c's. This is to prevent sensitive information from being leaked. Variable l_x corresponds to the supremum of all levels found in x's label (i.e., $\Gamma(x)$).

5.7 Discussion

5.7.1 Advantages

The use of fading labels increases the usability of information-flow control mechanisms by lowering the amount of resources needed and by increasing its permissiveness. It provides users with an easy way to parametrize mechanisms so that more resources are used to track important information and less resources are used to track less important information. Furthermore, since fading labels are similar to regular labels, they can easily be integrated into existing mechanisms.

A similar effect could be attained using multiple enforcement mechanisms and regular labels: there could be one mechanism per level of information and their precision could vary in function of this level. However, compared to fading labels, the simultaneous use of multiple enforcement mechanisms would introduce a significant runtime overhead.

Depth-limited noninterference is useful in scenarios where it is too costly to verify that a program satisfies noninterference and where an approximation is sufficient. For example, verifying concurrent programs is costly because every possible interleaving of events has to be considered. Depth-limited noninterference restricts the length of information-flow paths that have to be checked and hence reduces the cost of verification.

5.7.2 Disadvantages

While the use of fading labels increases the usability of information-flow control mechanisms, it also reduces their security; leaks of sensitive information may occur. In particular, a malicious application that is aware that it is being monitored by a mechanism which uses fading-labels could circumvent the mechanism and leak sensitive information (e.g., by inserting long dependence paths).

Another disadvantage is that there is no easy way to determine the "right" amount of uses after which a label should stop being propagated; it depends on the application being analyzed and the user's needs. Static analysis could be used to suggest values that help reduce the overhead introduced by the mechanism, while keeping the number of leaks to a minimum. This could be done by calculating the percentage of input variables that are tracked end-to-end. That is, the percentage of input variables of level ℓ for which there are no paths of length greater than $maxDepth(\ell)$ that lead to an output variable of lower level.

5.7.3 Variations

Here are a few interesting variations of the idea.

Time-Based Fading Labels

Instead of parameterizing fading labels with taint depths, timespans could be used so that the propagation stops after a certain amount of time. While this would not exactly respect depth-limited noninterference, it could be more intuitive to some users.

Usage-Based Fading Labels

Based on the observation that the further a variable is from the original source of sensitive information, the more likely it is that it will have lost information, our proposal in Section 5.4 decreases the counter each time the taint is propagated. However, this observation may not always be true. A safer alternative would be to decrease the counter only when non inversible operations are used (e.g., modulo operation). That is, only when we are sure that information is lost. This idea is closely related to the work in quantification [10], which aims at quantifying how much information is leaked by a program or output.

Probabilistic Fading Labels

Fading labels as defined in Section 5.4 stop propagating their taints once a certain depth has been reached. Another idea would be to parametrize fading labels with probabilities so that low-level variables have a low probability of propagating their taint, and high-level variables have a high probability.

5.7.4 Related Work

As far as we know, we are the first to propose a way to vary the amount of resources used by enforcement mechanisms by level of information. That being said, to reduce the amount of resources, fading labels automatically downgrades labels, a process known as *declassification*. Declassification is widely studied in language-based security [11]. While we use it as a way to reduce the amount of resources, it is typically used as a way to safely release sensitive information.

Sabelfeld et al. [9] introduces a notion called delimited release which stipulates that information may only be declassified via `declassify` commands which must be manually inserted into the code. Fading labels on the other hand automatically declassify information.

Kozyri et al. [12] propose to use automatons as labels. The automaton's state determines how the content of a variable can be used. Fading labels could be seen as a specific (and simpler) instance of this.

5.7.5 Future Work

We intend to use fading labels in an information-flow mechanism for Android. This will allow us to see how the idea holds in realistic scenarios. More specifically, we will empirically evaluate the performance of mechanisms with and without fading labels.

5.8 Bibliography

[1] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996. Available: http://dx.doi.org/10.3233/JCS-1996-42-304

[2] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, 2009, pp. 113–124. Available: http://doi.acm.org/10.1145/1554339.1554353

[3] A. Bedford, J. Desharnais, T. G. Godonou, and N. Tawbi, "Enforcing information flow by combining static and dynamic analysis," in *Foundations and Practice of Security - 6th International Symposium, FPS 2013, La Rochelle, France, October 21-22, 2013, Revised Selected Papers*, 2013, pp. 83–101. Available: https://doi.org/10.1007/978-3-319-05302-8_6

[4] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *the ACM Conference on Computer and Communications Security,*

CCS'12, Raleigh, NC, USA, October 16-18, 2012, 2012, pp. 881–893. Available: http://doi.acm.org/10.1145/2382196.2382289

[5] J. Agat, "Transforming out timing leaks," in *POPL 2000, Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, January 19-21, 2000*, 2000, pp. 40–53. Available: http://doi.acm.org/10.1145/325694.325702

[6] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, 1982, pp. 11–20. Available: https://doi.org/10.1109/SP.1982.10014

[7] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *Int. J. Inf. Sec.*, vol. 8, no. 6, pp. 399–422, 2009. Available: http://dx.doi.org/10.1007/s10207-009-0086-1

[8] A. Bedford, S. Chong, J. Desharnais, and N. Tawbi, "A progress-sensitive flow-sensitive inlined information-flow control monitor," in *ICT Systems Security and Privacy Protection - 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30 - June 1, 2016, Proceedings*, 2016, pp. 352–366. Available: http://dx.doi.org/10.1007/978-3-319-33630-5_24

[9] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *ISSS*, ser. Lecture Notes in Computer Science, vol. 3233. Springer, 2003, pp. 174–191.

[10] G. Smith, "Recent developments in quantitative information flow (invited tutorial)," in *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, 2015, pp. 23–31. Available: http://dx.doi.org/10.1109/LICS.2015.13

[11] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009. Available: http://dx.doi.org/10.3233/JCS-2009-0352

[12] E. Kozyri, O. Arden, A. C. Myers, and F. B. Schneider, "Jrif: Reactive information flow control for java," Tech. Rep., 2016. Available: https://ecommons.cornell.edu/handle/1813/41194

Chapter 6

Towards Automatically Generating Information-Flow Mechanisms

Author: Andrew Bedford

Conference: *ACM SIGPLAN Symposium on Principles of Programming Languages Student Research Competition (POPL SRC)*

Status: peer reviewed; presented

Year: 2018

6.1 Résumé

Développer des mécanismes de contrôle des flots d'information peut être une tâche difficile dû aux nombreux flots d'information à considérer. Une partie de ce problème vient du fait que les techniques pour prévenir les flots explicites et implicites doivent être appliquées manuellement lors de la conception de nouveaux mécanismes.

Afin de rendre cette tâche plus facile et réduire le risque d'erreur, nous présentons dans ce chapitre un outil appelé *Ott-IFC* qui, étant donné la spécification d'un langage de programmation, est capable d'appliquer automatiquement ces techniques pour générer la spécification d'un mécanisme de contrôle des flots d'information. Plus particulièrement, il analyse la syntaxe et sémantique d'un langage de programmation impératif sur lequel on veut appliquer la noninterférence et génère la sémantique d'un moniteur de sécurité hybride en utilisant des règles de réécriture.

6.2 Abstract

Developing information-flow control mechanisms can be a difficult and time-consuming task due to the numerous and subtle ways through which information may flow in a program. Part of the problem comes from the fact that, while techniques to prevent insecure explicit and implicit flows are well known and widely used, they still need to be applied manually when designing new mechanisms.

In order to make this task less laborious, we present in this chapter a tool called *Ott-IFC* that can, given a programming language's specification, automatically apply those techniques and generate information-flow control mechanism specifications. More specifically, it analyzes the syntax and semantics of an imperative programming language on which we want to enforce noninterference, and uses rewriting rules to generate a hybrid runtime monitor's semantics.

6.3 Introduction

Modern operating systems rely mostly on access-control mechanisms to protect users information. However, access control mechanisms are insufficient as they cannot regulate the propagation of information once it has been released for processing. To address this issue, a research trend called *language-based information-flow security* [1] has emerged. The idea is to use techniques from programming languages, such as program analysis, monitoring, rewriting and type checking, to enforce information-flow policies (e.g., information from a private file should not be saved in a public file). Mechanisms that enforce such policies (e.g., [2, 3, 4, 5]) are called *information-flow control mechanisms*.

Most information-flow control mechanisms seek to enforce a policy called *noninterference* [6], which essentially states that private information may not interfere with the publicly observable behavior of a program. To enforce noninterference, a mechanism must take into account two types of information flows: *explicit flows* and *implicit flows* [7].

An insecure explicit information flow occurs when private information influences public information through a data dependency. For example in Listing 6.1, the value that is written to `publicFile` depends on the value of `x`, which in turn depends on the value of `privateValue`. Hence, any output of `x` will reveal something about `privateValue`.

```
x := privateValue + 42;
write x to publicFile
```

Listing 6.1: Insecure explicit flow

Explicit flows can be prevented by associating labels to sensitive information and propagating them whenever the information is used; a process known as *tainting*.

An insecure implicit information flow occurs when private information influences public infor-

mation through the control-flow of the application. For example in Listing 6.2, the value that is written to `publicFile` depends on the condition `privateValue > 0`.

```
if (privateValue > 0) then
  write 0 to publicFile
else
  write 1 to publicFile
end
```

Listing 6.2: Insecure implicit flow

Implicit flows can be prevented using a *program counter*, which keeps track of the context in which a command is executed.

While techniques to prevent insecure explicit and implicit flows are well known and widely used [1], they are, to the best of our knowledge, still being applied manually when designing information-flow control mechanisms. This can lead to errors, failed proof attempts and time wasted. In order to make this task less laborious, we present in this chapter a tool called Ott-IFC that can, given an imperative programming language's specification (i.e., syntax and semantics), automatically apply those techniques and generate an information-flow monitor's specification.

6.4 Overview of Ott-IFC

As the name suggests, the specifications that Ott-IFC takes as input (and outputs) are written for Ott [8, 9]. Ott is a tool that can generate LaTeX, Coq [10] or Isabelle/HOL [11] versions of a programming language's specification. The specification is written in a concise and readable ASCII notation that resembles what one would write in informal mathematics.

Ott-IFC requires that the language specification follows a certain format. Namely, it requires:

- **R1** that the operational semantics of the language be either small-step or big-step operational semantics;
- **R2*** that the program configurations be of the form $\langle c, m, o \rangle$, where c is the command to be evaluated, m is the current memory (which maps variables to their values) and o is the current output trace;
- **R3*** that the outputs be appended to the output trace using the notation $o :: (channel, value)$, where *value* represents the output's value and *channel* its location;
- **R4** that the syntax be composed of *commands*, which may read/write the memory and produce outputs, and *expressions*, which may only read the memory;
- **R5** that the semantics preconditions does not include calls to functions that may have side-effects on the program configuration;

R6 that the semantics preconditions do not create any additional branching (e.g., $b \Rightarrow i = 1 \ \wedge \neg b \Rightarrow i = 2$).

Note that certain of these requirements are not restrictions on the language itself, but rather on how the specification is written; these requirements are denoted by a blue star (*). In general, requirements **R1**, **R2** and **R3** are there to give the language specification a certain structure so that we may more easily parse it. Requirement **R4** is there so that we may more easily identify the variables that may affect the control-flow. It also means that Ott-IFC does not support most functional languages; only imperative languages are supported. This is because in functional languages, the distinction between commands and expressions is less apparent and sometimes nonexistent. Note that this is not a technical limitation, but rather a choice that we made in order to simplify the approach. Requirement **R5** is there to ensure that any possible effects on the program configurations are expressed in the semantics, otherwise the generated mechanism could be unsound. Requirement **R6** is there to ensure that commands affecting the control-flow will need multiple rules to be defined, hence making it easier to detect such commands.

A language that satisfies these requirements is the imperative language whose syntax is defined in Listing 6.3 and semantics in Listings 6.4, 6.5, and 6.6.

```
arith_expr, a ::= x | n | a1 + a2 | a1 * a2
bool_expr, b ::= true | false | a1 < a2
commands, c ::= skip | x := a | c1 ; c2 | read x from ch | write x to ch |
                if b then c1 else c2 end | while b do c end
```

Listing 6.3: Syntax of a simple imperative language

```
%%% Lower Than %%%
<a1, m, o> --> <a1', m, o>
---------------------------------- :: lt_aexp_aexp
<a1 < a2, m, o> --> <a1' < a2, m, o>

<a2, m, o> --> <a2', m, o>
----------------------------------- :: lt_int_aexp
<n1 < a2, m, o> --> <n1 < n2, m, o>

n1 < n2 = true
------------------------------- :: lt_int_int_true
<n1 < n2, m, o> --> <true, m, o>

n1 < n2 = false
------------------------------ :: lt_int_int_false
<n1 < n2, m, o> --> <false, m, o>
```

Listing 6.4: Small-step semantics of boolean expressions

```
%%% Variable %%%
m(x) = n
---------------------------------------- :: lookup
<x, m, o> --> <n, m, o>

%%% Constant %%%
---------------------------------- :: int_constant
<n, m, o> --> <n, m, o>

%%% Addition %%%
<a1, m, o> --> <a1', m, o>
--------------------------------- :: add_aexp_aexp
<a1 + a2, m, o> --> <a1' + a2, m, o>


<a2, m, o> --> <a2', m, o>
---------------------------------- :: add_int_aexp
<n1 + a2, m, o> --> <n1 + n2, m, o>

n1 + n2 = n3
----------------------------------- :: add_int_int
<n1 + n2, m, o> --> <n3, m, o>

%%% Multiplication %%%
<a1, m, o> --> <a1', m, o>
-------------------------------- :: mult_aexp_aexp
<a1 * a2, m, o> --> <a1' * a2, m, o>

<a2, m, o> --> <a2', m, o>
--------------------------------- :: mult_int_aexp
<n1 * a2, m, o> --> <n1 * n2, m, o>

n1 * n2 = n3
---------------------------------- :: mult_int_int
<n1 * n2, m, o> --> <n3, m, o>
```

Listing 6.5: Small-step semantics of arithmetic expressions

```
%%% Assignment %%%
<a, m, o> --> <a', m, o>
----------------------------------- :: assign_aexp
<x := a, m, o> --> <x := a', m, o>

------------------------------------ :: assign_int
<x := n, m, o> --> <skip, m[x |-> n], o>

%%% Sequence %%%
<c1, m, o> --> <c1', m', o'>
------------------------------------------ :: seq1
<c1 ; c2, m, o> --> <c1' ; c2, m', o'>

------------------------------------------ :: seq2
<skip ; c2, m, o> --> <c2, m, o>

%%% Read %%%
m(ch) = n
------------------------------------------ :: read
<read x from ch, m, o> --> <skip, m[x |-> n], o>

%%% Write %%%
m(x) = n
----------------------------------------- :: write
<write x to ch, m, o> --> <skip, m[ch |-> n], o::(ch, n)>

%%% If %%%
<b, m, o> --> <b', m, o>
-------------------------------------- :: if_eval
<if b then c1 else c2 end, m, o> --> <if b' then c1 else c2 end, m, o>

-------------------------------------- :: if_true
<if true then c1 else c2 end, m, o> --> <c1, m, o>

------------------------------------- :: if_false
<if false then c1 else c2 end, m, o> --> <c2, m, o>

%%% While %%%
----------------------------------------- :: while
<while b do c end, m, o> --> <if b then c;while b do c end else skip end, m, o>
```

Listing 6.6: Small-step semantics of commands

To generate an information-flow monitor from this specification, Ott-IFC uses the algorithm presented in Algorithm 1.

Algorithm 1 Ott-IFC's algorithm

```
1: procedure GenMonitor(syntax, semantics)
2:     Identify nonterminal symbols of expressions and commands
3:     for each command in syntax do
4:         Identify rules for command in semantics
5:         Build the order-evaluation graph of command
6:         for each rule of command do
7:             Insert the typing environment E and program counter pc variables into the
               program configurations found in rule
8:             if rule or one of its successor in the graph modifies the memory then
9:                 Update the modified variable's label with the label of all expression variables
                   present in the rule and the pc variable
10:            end if
11:            if rule or one of its successor in the graph produces an output then
12:                Insert guard condition
13:            end if
14:            if command may affect the program's control flow then
15:                Update the pc
16:                Insert call to updateModifVars
17:            end if
18:        end for
19:    end for
20: end procedure
```

The algorithm can be decomposed into three main steps: identifying commands and expressions, constructing evaluation-order graphs, rewriting the semantics.

6.4.1 Step 1: Identifying commands and expressions

To identify which nonterminal symbols (e.g., `a`, `b`, `c`) of the syntax correspond to commands and which correspond to expressions, we need to identify the semantics rule associated to each nonterminal symbol. To do so, the algorithm analyzes each rule of the syntax and generates a set of strings that represents their possible values by recursively substituting the values of nonterminal symbols. For example, for the nonterminal symbol `bool_expr` (also written `b`), it returns the set of strings {**true**, **false**, `a < a`, `x < a`, `a < x`, `n < a`, `a < n`, `a + a < a`, ...}. This set of string is not exhaustive, only sufficiently so to be able to identify their use in the semantics.

Using this set of strings, the algorithm detects the rules that are associated to the nonterminal symbols by looking at the first element of the initial state configurations. In the case of `bool_expr`, Ott-IFC would identify that the rules associated to this nonterminal symbol are

those of Listing 6.4. Since none of these rules modify the memory or produce outputs, we can conclude that the nonterminal symbol `bool_expr` is an expression.

Using the same reasoning, the algorithm concludes that:

- the nonterminal symbols of expression are `arith_expr`, `a`, `bool_expr`, `b`, `x`, `n`, and the rules associated to those are the ones in Listings 6.5 and 6.4;
- the commands nonterminals are `commands, c`, and the rules associated to those are the ones in Listing 6.6.

6.4.2 Step 2: Constructing evaluation-order graphs

The next step consists in constructing *evaluation-order graphs* for each command. These graphs represent the order in which the semantics rules may be evaluated for a specific command (see Figure 6.1).

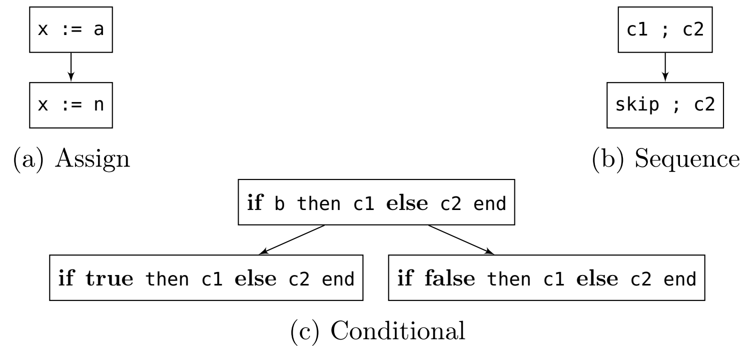

Figure 6.1: Evaluation-order graphs of the assign, sequence and conditional commands.

To construct these graphs, the algorithm uses the set of strings produced in Step 1 to detect that the rules associated to the assign command are `assign_aexp` and `assign_int`, and that `assign_aexp` will be evaluated first because `x := a` is more general than `x := n`. By more general, we mean that the set of strings that matches with `x := a` also matches with `x := n`, but the reverse is not true.

6.4.3 Step 3: Rule-based rewriting

The final step is to perform the actual rewriting of the semantics rules to insert a runtime monitor that enforces noninterference (i.e., a monitor that prevents insecure explicit and implicit flows of information). To do so, the algorithm replicates the thought process that a human would have when producing such a mechanism.

The algorithm starts by inserting a program counter `pc`, and a typing environment `E`, which maps variables to their labels, in each command configurations. That is, configurations that have the form `<c, m, o>` are changed to `<c, m, o, pc, E>`.

To prevent explicit flows, which are caused by a data-dependency, the algorithm identifies the commands that may modify the memory `m` (e.g., the assign command). In their rules, it updates the modified variable's label with the label of the expression variables that are used in the rule. If they also produce an output, then it inserts a guard condition to ensure that no leak of information occurs.

To prevent implicit flows, it identifies the commands that may influence the control-flow of the application using the evaluation-order graphs. That is, commands for which a program configuration may lead to two different program configurations (e.g., the if command). In other words, commands whose evaluation-order graph contains branches. It then updates the program counter `pc` with the labels of the expression variables that are present in the rule.

So, for the language of our example, we would obtain the following rules:

Read The `read` command reads the content of channel `ch` and assigns its value to `x`. For each expression variable found in the preconditions (i.e., `ch, n`), a label variable definition is inserted the output (lines 2 and 3 of Listing 6.7). Note that `|-` is the ASCII representation of $\vdash$, and `|_|` is the representation of the supremum operator $\sqcup$. Since the memory is modified, the label of the expressions that influence the assigned variable's value (i.e., the ones used in the rule) is propagated and the context in which the assignment occurs is taken into account (line 5).

```
1 m(ch) = n
2 E |- ch : lch
3 E |- n : ln
4 ----------------------------------------- :: read
5 <read x from ch, m, o, pc, E> --> <skip, m[x |-> n], o, pc, E[x |-> pc |_| lch |_| ln]>
```

Listing 6.7: Ott-IFC's output for the "read" command

Write The `write` command writes the value of `x` on the channel `ch`. Because this command produces an output (i.e., it appends a value to `o`), the algorithm adds a guard condition that ensures that no leak of information will occur at runtime (line 5 of Listing 6.8). We do not update the label of the channel on which the output occurs even if it is modified because, if the execution is not stopped by the monitor, then it means that the channel's label is already greater than the label of the expression that is written.

```
m(x) = n
E |- x : lx
E |- n : ln
E |- ch : lch
lx |_| ln |_| pc <= lch
---------------------------------------- :: write
<write x to ch, m, o, pc, E> --> <skip, m[ch |-> n], o::(ch, n), pc, E>
```

Listing 6.8: Ott-IFC's output for the "write" command

Assign While the `assign_aexp` rule does not directly modify the memory, one of its successor in the evaluation-order graph (i.e., `assign_int`) does. For this reason, the algorithm updates the label of the modified variable in, not only `assign_int`, but also `assign_aexp` (lines 5 and 10 of Listing 6.9). This is to take into account the label of variables that are in the expression `a` (before they disappear). This means that, in this case, the generated monitor will not allow the label of variables to be "downgraded" as a result of an assignment.

```
<a, m, o> --> <a', m, o>
E |- x : lx
E |- a : la
---------------------------------- :: assign_aexp
<x := a, m, o, pc, E> --> <x := a', m, o, pc, E[x |-> lx |_| pc |_| la]>

E |- x : lx
E |- n : ln
----------------------------------- :: assign_int
<x := n, m, o, pc, E> --> <skip, m[x |-> n], o, pc, E[x |-> lx |_| pc |_| ln]>
```

Listing 6.9: Ott-IFC's output for the "assign" command

It may be interesting to note that, had the big-step version of the semantics been given to Ott-IFC instead, the generated monitor will allow the label of variables to be downgraded (see Listing 6.10). This is because in the big-step version, the assign command has only one rule and it is executed in one step. In other words, the form of the specification given as input influences the permissiveness of the monitor generated by Ott-IFC (more on this in Section 6.5).

```
<a, m, o> || <n, m, o>
E |- a : la
---------------------------------------- :: assign
<x := a, m, o, pc, E> || <skip, m[x |-> n], o, pc, E[x |-> pc |_| la]>
```

Listing 6.10: Ott-IFC's output the big-step version of the "assign" command

Sequence The rules for the sequence do not modify the memory or produce outputs so, like the skip command, the only changes in the rules are the addition of the `pc` and `E` variables (see Listing 6.11). Notice that the `pc` variable on line 3 of `seq1` has the same value in the initial configuration and final configuration. This means that, though the value of the `pc` variable may change during the execution of a command (e.g., the if command), once the command has finished its execution, it is restored to its previous value (as per usual).

```
<c1, m, o, pc, E> --> <c1', m', o', pc', E'>
----------------------------------------- :: seq1
<c1 ; c2, m, o, pc, E> --> <c1' ; c2, m', o', pc, E'>

----------------------------------------- :: seq2
<skip ; c2, m, o, pc, E> --> <c2, m, o, pc, E>
```

Listing 6.11: Ott-IFC's output for the "sequence" command

If In this language, the **if** command is the only that can directly cause the control-flow of a program to branch out. We can tell this by looking at its evaluation-order graph (Figure 6.1): there is a choice between **if true** and **if false**. For this reason, the algorithm updates the `pc` variable with the labels of the expression variables that are present in the rule (i.e., only `b` in this case).

It also inserts a call to the function `updateModifVars`, which has to be implemented by the user. This function must identify the variables that could have been modified in either `c1` or `c2`, and update their labels with the label of `b` (line 3 of Listing 6.12). This is to ensure that the labels of the variables after executing the **if** command are always the same, no matter which branch is taken during execution. Note that the command variables that are used in the rule (variables `c1` and `c2`) are detected in the same way as expression variables are detected: using the set of strings generated in Step 1.

```
<b, m, o> --> <b', m, o>
E |- b : lb
E1 = updateModifVars(E, pc |_| lb, {c1,c2})
-------------------------------------- :: if_eval
<if b then c1 else c2 end, m, o, pc, E> -->
<if b' then c1 else c2 end, m, o, pc |_| lb, E1>

-------------------------------------- :: if_true
<if true then c1 else c2 end, m, o, pc, E> --> <c1, m, o, pc, E>

------------------------------------ :: if_false
<if false then c1 else c2 end, m, o, pc, E> --> <c2, m, o, pc, E>
```

Listing 6.12: Ott-IFC's output for the "if" command

While Like for the skip and sequence commands, the semantics rule of the while command does not modify the memory or the output trace, hence the only change is the addition of the `pc` and `E` variables to the configurations. Note that, while the `pc` variable is not updated here with the label of the condition variable `b`, it will be when the **if** (present in the final configuration) is evaluated (line 3 of Listing 6.13).

```
1 -------------------------------------- :: while
2 <while b do c end, m, o, pc, E> -->
3 <if b then c ; while b do c end else skip end, m, o, pc, E>
```

Listing 6.13: Ott-IFC's output for the "while" command

6.5 Discussion

As far as we know, we are the first ones to propose a way to generate information-flow control mechanism specifications from programming language specifications. Our implementation is available online and open-source [12]. Once it is further developed, we expect that our tool will be particularly useful to researchers in language-based security who need to quickly develop mechanisms. The development process of a mechanism using Ott and Ott-IFC would look like this:

1. Write specification of the language on which we want to enforce noninterference in Ott.
2. Use Ott-IFC to generate the mechanism's specification.
3. Use Ott to export the mechanism's specification to LaTeX, Coq or Isabelle/HOL and complete the implementation.

Experiments We have successfully tested Ott-IFC on a few small imperative languages[1]. The languages are inspired by those of the IFC-Challenge [13]. More specifically, we have tested our approach on:

- the language presented in this chapter (using small-step and big-step semantics),
- a language with exceptions (try-catch/throw, using big-step semantics),
- a language with locally-scoped variables (let x := e in, using big-step semantics).

We have also begun investigating more complex languages, such as CompCert's Clight (a subset of C) [14], whose specification is partially written in Ott.

One interesting thing that we have observed during our experiments is that the form of the semantics given as input may impact the permissiveness of the generated monitor. For instance,

[1]The language specifications are available the project's GitHub page

the monitor that is generated from the small-step semantics of the imperative language used in this chapter is flow-insensitive and less permissive than the flow-sensitive one generated from the big-step semantics. Meaning that the former will reject the following program, while the latter will accept it.

```
x := privateInfo;
x := publicInfo;
write x to publicFile (*public on public, no information leak*)
```

Listing 6.14: Program accepted by flow-sensitive analyses

Whether this is due to the approach in general or due to the algorithm that is used remains to be seen; we suspect the latter.

Limitations For the moment, Ott-IFC can only produce one kind of mechanism: runtime monitors. Runtime monitors are the easiest to generate as they are simply modified versions of the semantics given as input. However, we expect that the same logic could be used to produce other types of mechanisms, such as type systems.

Ott-IFC supports only languages that meet the requirements listed in Section 6.4. Once our tool is further developed, we may be able to remove or weaken these restrictions, but for the moment they are necessary. They give the specification a certain structure so that we may more easily parse it. As previously mentioned, we expect restrictions **R2** and **R3** to be easiest to remove as they are not restrictions on the language itself, but rather on how they are written. Restrictions **R4**, **R5** and **R6** on the other hand are restrictions on the language itself and so, should be the hardest to remove.

The mechanisms generated by Ott-IFC may not be as permissive as those designed by humans. For example, the flow-insensitive monitor presented earlier could have been flow-sensitive and hence, more permissive. Nevertheless, even if the generated mechanism is not permissive enough for the needs of a user, it can still be used as a foundation/starting point to design a more permissive mechanism. For such a user, the work of Hritcu et al. [15] might be of interest. They show how to use QuickCheck, a property-based random-testing tool, to quickly verify that a mechanism correctly enforces noninterference. Their tool identifies errors during the design phase of the mechanism, thereby allowing users to postpone any proof attempts until they are confident of the mechanism's soundness.

6.6 Conclusion and Future Work

We have presented in this chapter Ott-IFC, a tool capable of generating an information-flow control monitor specification from a programming language's syntax and semantics. It does so by automatically applying techniques that are known to prevent explicit and implicit

information flows. Our experiments on simple imperative language show that the tool and its approach is promising. As future work, we plan on adding support for more languages, parameterization, and generating formal proofs.

Language Support Our requirements on specifications means that only certain types of languages can be used in Ott-IFC. We are currently in the process of building a repository of formalized languages so that we can test and extend our approach to a wider range of languages.

Parameterization We plan on parameterizing our tool so that users can choose the type of mechanism to generate (e.g., type system, monitor) and choose some of its features (e.g., flow-sensitivity, termination-sensitivity [16], progress-sensitivity [17], value-sensitivity [18]).

Generating Formal Proofs As previously mentioned, we expect that some users will use the generated mechanisms as a foundation to build better and more precise mechanisms. One of the most grueling task when building an information-flow control mechanism is to prove its soundness. In order to help those users, we plan on generating a skeleton of the proof in Coq or Isabelle/HOL (both languages are supported by Ott).

Verifying Existing Mechanisms The same rules that Ott-IFC uses to generate mechanisms could be used to verify the soundness of existing mechanisms and identify potential errors. For example, we could raise a warning if an output is produced but no guard condition is present.

6.7 Bibliography

[1] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003. Available: https://doi.org/10.1109/JSAC.2002.806121

[2] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2/3, pp. 167–188, 1996. Available: https://doi.org/10.3233/JCS-1996-42-304

[3] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, 2010, pp. 200–214. Available: https://doi.org/10.1109/CSF.2010.21

[4] A. Askarov, S. Chong, and H. Mantel, "Hybrid monitors for concurrent noninterference," in *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, 2015, pp. 137–151. Available: https://doi.org/10.1109/CSF.2015.17

[5] A. Bedford, S. Chong, J. Desharnais, E. Kozyri, and N. Tawbi, "A progress-sensitive flow-sensitive inlined information-flow control monitor (extended version)," *Computers & Security*, vol. 71, pp. 114–131, 2017. Available: https://doi.org/10.1016/j.cose.2017.04.001

[6] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, 1982, pp. 11–20. Available: https://doi.org/10.1109/SP.1982.10014

[7] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976. Available: http://doi.acm.org/10.1145/360051.360056

[8] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa, "Ott: Effective tool support for the working semanticist," *J. Funct. Program.*, vol. 20, no. 1, pp. 71–122, 2010. Available: https://doi.org/10.1017/S0956796809990293

[9] "Ott," http://www.cl.cam.ac.uk/~pes20/ott/.

[10] "The coq proof assistant," https://coq.inria.fr/.

[11] "Isabelle/hol," https://isabelle.in.tum.de/.

[12] A. Bedford, "Ott-ifc's repository," https://github.com/andrew-bedford/ott-ifc, 2017.

[13] "Ifc-challenge," https://ifc-challenge.appspot.com/.

[14] S. Blazy and X. Leroy, "Mechanized semantics for the clight subset of the C language," *J. Autom. Reasoning*, vol. 43, no. 3, pp. 263–288, 2009. Available: https://doi.org/10.1007/s10817-009-9148-3

[15] C. Hritcu, L. Lampropoulos, A. Spector-Zabusky, A. A. de Amorim, M. Dénès, J. Hughes, B. C. Pierce, and D. Vytiniotis, "Testing noninterference, quickly," *J. Funct. Program.*, vol. 26, p. e4, 2016. Available: https://doi.org/10.1017/S0956796816000058

[16] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, 2008, pp. 333–348. Available: https://doi.org/10.1007/978-3-540-88313-5_22

[17] S. Moore, A. Askarov, and S. Chong, "Precise enforcement of progress-sensitive security," in *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 881–893. Available: http://doi.acm.org/10.1145/2382196.2382289

[18] D. Hedin, L. Bello, and A. Sabelfeld, "Value-sensitive hybrid information flow control for a javascript-like language," in *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, 2015, pp. 351–365. Available: https://doi.org/10.1109/CSF.2015.31

Chapter 7

Coqatoo: Generating Natural Language Versions of Coq Proofs

Author: Andrew Bedford

Conference: *International Workshop on Coq for Programming Languages (CoqPL)*

Status: peer reviewed; presented

Year: 2018

7.1 Résumé

Dû à leur nombreux avantages, les preuves formelles et les assistants de preuves, tel que Coq, sont de plus en plus populaires. Toutefois, un désavantage d'utiliser un assistant de preuve est que les preuves résultantes peuvent parfois être difficile à lire et à comprendre, particulièrement pour un utilisateur moins expérimenté.

Pour adresser ce problème, nous présentons dans ce chapitre un outil appelé *Coqatoo* qui génère une version en langue naturelle de preuves Coq. Contrairement aux travaux précédents, la version en langue naturelle est générée directement à partir du code de la preuve. Cette approche permet à Coqatoo de donner à l'utlisateur un contrôle direct sur la verbosité de la preuve générée: plus le code de la preuve est détaillé, plus la version en langue naturelle sera détaillée.

7.2 Abstract

Due to their numerous advantages, formal proofs and proof assistants, such as Coq, are becoming increasingly popular. However, one disadvantage of using proof assistants is that the

resulting proofs can sometimes be hard to read and understand, particularly for less experienced users.

To address this issue, we present in this chapter a tool called *Coqatoo* that generates natural language versions of Coq proofs. Contrarily to previous work, the natural language versions are generated from high-level proof scripts instead of low-level proof-terms. By adopting this approach, Coqatoo can avoid the inherent verbosity that comes from using low-level proof-terms and gives the user direct control over the verbosity of the generated natural language version: the more detailed a proof script is, the more detailed its natural language version will be.

7.3 Introduction

When developing information-flow control mechanisms, writing proofs is unavoidable. For this task, proof assistants, such as Coq, are becoming increasingly popular due to their numerous advantages (e.g., machine-checked, automated, reusable). However, one disadvantage of using proof assistants is that the resulting proofs can sometimes be hard to read and understand, particularly for less experienced users. In an attempt to address this issue, Coscoy et al. [1] developed in 1995 an algorithm capable of generating natural language proofs from Coq proof-terms (i.e., calculus of inductive construction λ-terms) and implemented their approach in two development environments: CtCoq [2, 3] and its successor Pcoq [4, 5]. Unfortunately, these development environments are no longer available or maintained; Pcoq's last version dates from 2003 and requires Coq 7.4.

In order to bring this useful feature to modern development environments so that researchers may more easily communicate their proofs, we have implemented our own rewriting algorithm: Coqatoo.

7.4 Overview of Coqatoo

Much like Nuprl's text generation algorithm [6], Coqatoo generates natural language proofs from high-level proof scripts instead of the low-level proof-terms used by Coscoy et al. By doing so, it can avoid the verbosity that comes from using low-level proof-terms [7] and avoid losing valuable information such as the tactics that are used, the user's comments and the variable names.

Coqatoo's rewriting algorithm can be decomposed in three steps: information extraction, proof tree construction and tactic-based rewriting. We will illustrate these steps using the proof script of Listing 7.1, whose goal is to show that $\forall P, Q, R.(P \wedge Q \Rightarrow R) \Leftrightarrow (P \Rightarrow Q \Rightarrow R)$.

```
Lemma conj_imp_equiv : forall P Q R:Prop,
  (P /\ Q -> R) <-> (P -> Q -> R).
Proof.
  intros. split. intros H HP HQ. apply H. apply conj. assumption. assumption.
  intros H HPQ. inversion HPQ. apply H. assumption. assumption.
Qed.
```

Listing 7.1: Proof script given as input

Step 1: Information extraction Coqatoo starts by executing the proof's script in Coq to capture the intermediate proof states.

For example, Listing 7.2 represents the initial state of Listing 7.1's proof and Listing 7.3 represents the state after executing the first `intros` tactic, which introduces the variables `P`, `Q` and `R` into the context.

```
1 subgoal


============================
forall P Q R : Prop, (P /\ Q -> R) <-> (P -> Q -> R)
```

Listing 7.2: State before executing the first intros tactic

```
1 subgoal

P, Q, R : Prop
============================
(P /\ Q -> R) <-> (P -> Q -> R)
```

Listing 7.3: State after executing the first intros tactic

These intermediate states, which contain the current assumptions and remaining goals, allow us to identify the changes caused by a tactic's execution (e.g., added/removed variables, hypotheses or subgoals).

Step 2: Proof tree construction We then build a tree representing the proof's structure (e.g., Figure 7.1). This is a necessary step for our rewriting algorithm as it allows it to determine where bullets should be inserted and when lines should be indented.

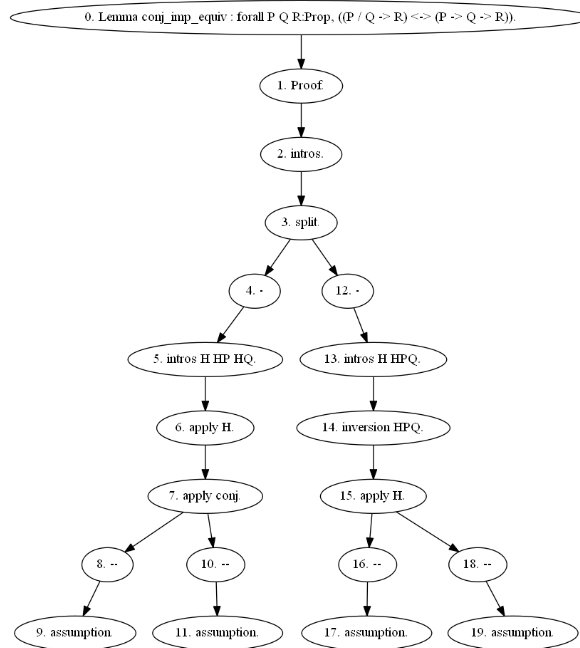

Figure 7.1: Proof tree of Listing 7.1

Step 3: Tactic-based rewriting Finally, we generate the actual final natural language version of the proof using simple rewriting rules. Each supported tactic has its own set of rules. For example, for the `intros` tactic we first determine the types of the objects that are introduced. If they are variables, then we produce a sentence of the form `"Assume that ... are arbitrary objects of type ..."`. If they are hypotheses, then we instead produce a sentence of the form `"Suppose that ... are true"`. Finally, we insert a sentence indicating what is left to prove: `"Let us show that ..."`.

Note that the sentences that we use to produce natural language versions are kept in files that are separate from the code. This allows Coqatoo to support multiple languages and proof styles. For the moment, it can output proofs in English or French, in plain text, in LaTeX or in annotation mode (see Listing 7.4 for example). In annotation mode, each tactic is accompanied with an informal explanation. We believe that this format will be particularly useful for new Coq users.

```
Lemma conj_imp_equiv : forall P Q R:Prop, ((P /\ Q -> R) <-> (P -> Q -> R)).
Proof.
(* Assume that P, Q and R are arbitrary objects of type Prop. Let us show that (P /\ Q ->
  R) <-> (P -> Q -> R) is true. *) intros.
split.
  - (* Case (P /\ Q -> R) -> P -> Q -> R: *)
    (* Suppose that P, Q and P /\ Q -> R are true. Let us show that R is true. *) intros H
   HP HQ.
    (* By our hypothesis P /\ Q -> R, we know that R is true if P /\ Q  is true. *) apply
  H.
    apply conj.
    -- (* Case P: *)
       (* True, because it is one of our assumptions. *) assumption.
    -- (* Case Q: *)
       (* True, because it is one of our assumptions. *) assumption.
  - (* Case (P -> Q -> R) -> P /\ Q -> R: *)
    (* Suppose that P /\ Q and P -> Q -> R are true. Let us show that R is true. *) intros
   H HPQ.
    (* By inversion on P /\ Q, we know that P, Q are also true. *) inversion HPQ.
    (* By our hypothesis P -> Q -> R, we know that R is true if P and Q are true. *) apply
   H.
    -- (* Case P: *)
       (* True, because it is one of our assumptions. *) assumption.
    -- (* Case Q: *)
       (* True, because it is one of our assumptions. *) assumption.
Qed.
```

Listing 7.4: Coqatoo's output in annotation mode

7.5 Comparison

Compared to Coscoy et al., our approach presents a few disadvantages and advantages.

Disadvantages

- It only works on proofs whose tactics are supported (see Section 7.6), while the approach of Coscoy et al. worked on any proof.
- It may require additional verifications to ensure that unnecessary information (e.g., an assertion which isn't used) is not included in the generated proof.

Advantages

- It enables the user to more easily control the size and verbosity of the generated proof (one or two sentences per tactic by default); the more detailed a proof script is, the more detailed its natural language version will be. The proofs generated by Coscoy et al. often quickly exploded in size (see Figure 7.2 for example).

- It maintains the order and structure of the user's original proof script; this is not necessarily the case in Coscoy et al.

Consider a set U.
Consider a R of type $(Rel\ U)$.
Assume $(Trans\ U\ R)$ $(trans)$.
Consider an element x of U.
Consider an element y of U.
Consider an element z of U.
Assume $(Inv\ U\ R\ x\ y)$ (h_1).
Assume $(Inv\ U\ R\ y\ z)$ (h_2).
* With hypothesis $trans$ **we have** $(Trans\ U\ R)$ **which is equivalent to** $\forall x, y, z{:}\, U. (R\ x\ y) \Rightarrow \cdots$.
* With hypothesis h_2 **we have** $(Inv\ U\ R\ y\ z)$ **which is equivalent to** $(R\ z\ y)$.
* With hypothesis h_1 **we have** $(Inv\ U\ R\ x\ y)$ **which is equivalent to** $(R\ y\ x)$.
Applying the first result to the two others **we get** $(R\ z\ x)$ **which is equivalent to** $(Inv\ U\ R\ x\ z)$.
We have proven $(Inv\ U\ R\ y\ z) \Rightarrow (Inv\ U\ R\ x\ z)$.
We have proven $(Inv\ U\ R\ x\ y) \Rightarrow (Inv\ U\ R\ y\ z) \Rightarrow (Inv\ U\ R\ x\ z)$.
We have proven $\forall z{:}\, U. (Inv\ U\ R\ x\ y) \Rightarrow \cdots$.
We have proven $\forall y, z{:}\, U. (Inv\ U\ R\ x\ y) \Rightarrow \cdots$.
We have proven $\forall x, y, z{:}\, U. (Inv\ U\ R\ x\ y) \Rightarrow \cdots$ **which is equivalent to** $(Trans\ U\ (Inv\ U\ R))$.
We have proven $(Trans\ U\ R) \Rightarrow (Trans\ U\ (Inv\ U\ R))$.
We have proven $\forall R{:}\, (Rel\ U). (Trans\ U\ R) \Rightarrow (Trans\ U\ (Inv\ U\ R))$.
We have proven $\forall U{:}\, Set. \forall R{:}\, (Rel\ U). (Trans\ U\ R) \Rightarrow (Trans\ U\ (Inv\ U\ R))$.

Figure 7.2: Example of a proof generated by the approach of Coscoy et al.

7.6 Future Work

Coqatoo is only a proof of concept for the moment. As such, there remains much to be done before it can be of real use.

Increase the number of supported tactics The number of tactics that it supports is limited to only a handful (see Coqatoo's GitHub repository [8] for more details). We expect that, with the help of the community, we will be able to support enough tactics to generate natural language versions of most proofs in *Software Foundations* [9], a book aimed at new Coq users which contains a large number of proofs.

Add partial support for automation In regards to automation, Coqatoo only supports the `auto` tactic: if the `auto` tactic is present within the script, it is replaced with `info_auto` in order to obtain the sequence of tactics that is used by `auto`. We plan on adding partial support for automation in the future, starting with the chaining operator `";"`. To support this operator we will use our tree representation of proofs to "distribute" tactics on branches.

Integration with development environments Once it is sufficiently developed, we plan on integrating our utility in modern Coq development environments such as CoqIDE and ProofGeneral.

Acknowledgments

We would like to thank Josée Desharnais, Nadia Tawbi, Souad El Hatib and the reviewers for their comments. We would also like to thank the Coq community for the large number of resources and tutorials that are available online.

7.7 Bibliography

[1] Y. Coscoy, G. Kahn, and L. Théry, "Extracting text from proofs," in *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, 1995, pp. 109–123. Available: https://doi.org/10.1007/BFb0014048

[2] Y. B. et al., "Ctcoq," https://www-sop.inria.fr/croap/ctcoq/ctcoq-eng.html, 1997.

[3] Y. Bertot, "The ctcoq system: Design and architecture," *Formal aspects of Computing*, vol. 11, no. 3, pp. 225–243, 1999.

[4] Y. B. et al., "Pcoq," http://www-sop.inria.fr/lemme/pcoq/, 2003.

[5] A. Amerkad, Y. Bertot, L. Pottier, and L. Rideau, "Mathematics and proof presentation in pcoq," Ph.D. dissertation, INRIA, 2001.

[6] A. M. Holland-Minkley, R. Barzilay, and R. L. Constable, "Verbalization of high-level formal proofs," in *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA.*, 1999, pp. 277–284. Available: http://www.aaai.org/Library/AAAI/1999/aaai99-041.php

[7] Y. Coscoy, "A natural language explanation for formal proofs," in *Logical Aspects of Computational Linguistics, First International Conference, LACL '96, Nancy, France, September 23-25, 1996, Selected Papers*, 1996, pp. 149–167. Available: https://doi.org/10.1007/BFb0052156

[8] A. Bedford, "Coqatoo's repository," https://github.com/andrew-bedford/coqatoo, 2017.

[9] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey, *Software foundations*, 2010, http://www.cis.upenn.edu/bcpierce/sf/current/index.html.

Conclusion

In order to improve the overall practicality and usability of information-flow control mechanisms, we have presented in this thesis new ways of identifying precisely when dynamic information-flow control should be necessary using static analysis, new ways of parameterizing the precision of an analysis and new tools to help researchers develop sound mechanisms.

More specifically, in Chapters 2 and 3, we have shown that a combination of static and dynamic analysis can increase the effectiveness of information-flow control mechanisms without introducing too much overhead. Our method consists in three steps: (1) using a type-based static analysis, we verify that the program does not contain any obvious information leaks and identify what cannot be verified accurately statically (using either an *unknown* label or sets of levels); (2) we instrument the program to prevent less obvious leaks from occurring at runtime; (3) we partially evaluate the program to minimize the instrumentation's impact on execution time.

In Chapter 4, we presented a new malware detection tool for Android called Andrana which relies on static analysis and machine-learning techniques. It can be used by users to identify the applications that are most likely to leak sensitive information (i.e., malicious applications), and hence in most need of information-flow control. It boasts an accuracy of 94% and takes less than a second to perform its analysis.

In Chapter 5, we introduced the concept of fading-labels and depth-limited noninterference. They allow users to prioritize the usage of resources to track information from certain sources. This concept should be particularly useful in systems where resources are limited (e.g., smartphones, tablets) and where a complete analysis may be too costly. As far as we know, we are the first to propose a way to vary the amount of resources used by enforcement mechanisms by level of information.

In Chapter 6, we presented Ott-IFC, a tool that can, given a programming language's specification (i.e., syntax and semantics), generate an information-flow monitor's specification. It does so by automatically applying techniques that are known to prevent explicit and implicit information flows. Our experiments on simple imperative language show that the tool and its approach are promising. To the best of our knowledge, this is the first tool of this kind. Once

further developed, this tool should allow researchers to quickly develop and test a variety of information-flow control mechanisms.

Since Ott-IFC uses Ott as its input and output language, specifications can be exported to Coq so that users may complete its implementation and prove its soundness. For this reason, we presented Coqatoo in Chapter 7, a tool that can generate natural-language version of Coq proofs. This tool can be used by researchers to more easily communicate their Coq proofs, or by new Coq users as a learning aid. Compared to previous existing work, its main advantage is that it gives the user a direct control over the size and verbosity of the generated proof.

Future Work

Improving the precision of our static analysis The more precise the static analysis, the less the dynamic analysis will have to do and hence, the less overhead will be introduced. One way to enhance the precision of our static analysis would be to use abstract interpretation. For example, suppose that abstract interpretation reveals that the value of variable `x` before executing the **if** command is always in the range $[0, 5]$, then the analysis could conclude that variable `y` always contains public information, hence removing the need to instrument the following output operation.

```
if x < 5 then
  y := publicInfo;
else
  y := privateInfo;
end;
write y to publicFile
```

Listing 7.5: Improved precision using abstract interpretation

Real world languages and applications The techniques and tools presented in this thesis could be scaled up to deal with real world languages and applications. As a first step towards this, we could start by testing our approaches on languages that have complex data structures (e.g., trees, records, classes). Concurrency will also need to be studied.

Since real world applications often need to leak information, we will need to further study the topic of declassification, which we have briefly touched in Chapter 5. An example of a scenario where an application must leak information is during a login attempt: the application must reveal to the user whether or not the password is correct.

For Coqatoo to support proofs on real world languages, we will need to add support for additional tactics so that it can be used to generate natural language versions for a wider variety of proofs. We will start by adding support for the tactics that are used in the book *Software Foundations*.

Characterizing the dangerousness of a leak In this thesis, we chose to stop the execution of the program when a leak is about to occur. Alternatively, what we could do is to alert the user when a leak is about to occur and let the user decide whether the leak is acceptable. To take his decision, the user will need information on the leak and its potential dangerousness. Hence, we will characterize what makes a leak truly dangerous and elaborate a quantifiable dangerousness metric.

Verifying existing mechanisms We believe that the same rules that Ott-IFC uses to generate mechanisms could be used to verify the soundness of existing mechanisms and identify potential errors. For example, we could raise a warning if an output is produced but no guard condition is present. As many of the mechanisms and approaches described in the literature are illustrated on simple languages, we expect this objective to be more easily attainable than scaling to real world languages.

Differential privacy using language-based security Differential privacy is a relatively new topic of research that aims to protect the privacy of individuals whose data is included in databases on which queries are performed. During our work on information-flow control, we have observed that the ideas of noninterference and differential privacy are somewhat alike: noninterference states that a variation of private inputs should not affect the public outputs, while differential privacy states that private inputs (rows in the database) should not significantly affect the results of a query. It would be interesting to see if the techniques and tools developed in this thesis could be of use for differential privacy.