PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure - PSL

Intertwining Symbolic Execution and Abstract Interpretation for the Analysis of Security Properties

Soutenue par

Ignacio Tiraboschi

Le 1/09/2024

École doctorale nº386

École doctorale de Sciences Mathématiques de Paris Centre

Spécialité

Informatique

Préparée au

Antique

Composition du jury :

TBD Titre, établissement	*Président du jury*
David NAUMANN Professor, Stevens Institute of Technology	*Rapporteur*
Cristian CADAR Professor, Imperial College London	*Rapporteur*
Francesco LOGOZZO Software Engineer, Meta	*Examinateur*
Bruno BLANCHET Directeur de recherche, INRIA	*Examinateur*
Lesly-Ann DANIEL Postdoc, KU Leuven	*Examinatrice*
Xavier RIVAL Directeur de recherche, École normale supérieure	*Directeur de thèse*
Tamara REZK Directrice de recherche, INRIA Sophia Antipolis	*Directrice de thèse*

ESPCI PARIS | PSL

Remerciements

This will be filled at a later time.

Résumé

Ce travail se concentre sur l'application de *l'analyse statique* pour la vérification ou la réfutation automatique de *propriétés de flux d'information*, en se concentrant sur l'interprétation abstraite et l'exécution symbolique. et l'exécution symbolique. Plus précisément, nous nous concentrons sur deux propriétés de flux d'informations : *non-interférence*, et *secret faible*.

La thèse est divisée en deux parties. Dans la première partie de la thèse, nous explorons une analyse statique basée sur des symboles qui communique avec une analyse des dépendances pour la vérification de l'intégrité du système. avec une analyse des dépendances pour la vérification de la non-interférence. Notre contribution est un domaine de produit réduit entre un domaine symbolique et un domaine de dépendances pour l'analyse solide de la non-interférence dans un langage impératif simple. Nous proposons également un produit réduit entre une exécution symbolique non relationnelle et des domaines numériques tels que les intervalles et les polyèdres convexes.

La deuxième partie consiste à explorer le concept d'un taint-tracker symbolique statique. Nous développons une sémantique formelle pour un traqueur de taches symbolique, ainsi que l'adaptation du secret faible pour les cas d'utilisation modernes. Ensuite, en supposant l'existence d'un taint-tracker sain mais imprécis, nous proposons une analyse combinée qui utilise à la fois le taint-tracker sain et le taint-tracker imprécis. analyse combinée qui utilise à la fois les traqueurs de taches sonores et symboliques. Enfin, nous instancions l'analyse combinée avec Pysa, un taint-tracker sonore développé par Meta, et notre outil Pysta. Notre outil affine les résultats de l'analyseur de taches sonores, réduisant ainsi la charge de travail des développeurs pour l'examen manuel des alarmes. d'examiner manuellement les alarmes.

Mots clés : analyse statique, flux d'informations, interprétation abstraite, exécution symbolique

Abstract

This work focuses on the application of *static analysis* for the automatic verification or refutation of *information flow properties*, focusing on abstract interpretation and symbolic execution. More specifically, we focus on two information flow properties: *noninterference*, and *weak secrecy*.

The thesis is split in two parts. In the first part of the thesis we explore a symbolically driven static analysis that communicates with a dependences analysis for the verification of noninterference. Our contribution is a reduced product domain between a symbolic domain and a dependences domain for the sound analysis of Noninterference in a simple imperative language. We also offer a reduced product between a non-relational symbolic execution and numerical domains such as intervals and convex polyhedra.

The second part consists on exploring the concept of a static symbolic taint-tracker. We develop a formal semantics for a symbolic taint-tracker, together with the adaptation of weak secrecy for modern use-cases. Then, by assuming the existence of a sound-but-imprecise taint-tracker, we contribute a combined analysis that uses both the sound and the symbolic taint-trackers. Finally, we instantiate the combined analysis with Pysa, a sound taint-tracker developed by Meta, and our tool Pysta. Our tool refines the output of the sound taint-tracker, lowering the load of developers to manually review alarms.

Keywords : static analysis, information flow, abstract interpretation, symbolic execution

Contents

Liste des figures

List of Tables

Chapter 1

Introduction

1 Security challenges and properties

We constantly interact with computer systems that perform a variety of tasks. Some, such as web applications, handle sensitive information that can cause damage if leaked. Others, like a water plant, are critical systems, and their failure can result in physical damage, affecting thousands of people.

Often, these systems are connected to the Internet, making them vulnerable to computer attacks. For instance, in 2017, a weakness in Apache Struts (CVE-2017-5638) caused the American company Equifax to leak sensitive information of approximately 150 million people. The leaked data included full names and social security number, making it one of the most serious cyber-security attacks in history. We consider weaknesses or flaws that can be exploited by cybercriminals to be *vulnerabilities* in the system.

In the 2017 data breach of Equifax, the attack consisted on targeting a vulnerable Apache Struts parser. By sending maliciously crafted messages, the attackers were able to inject executable code. At that stage, the attackers gained access to the system, and progressively went deeper withing system until they reached databases with confidential information. This attack falls into the category of *information-flow vulnerabilities*. Information-flow vulnerabilities occur when a system fails to properly control the movement of sensitive data.

An information flow can happen in two ways: *explicit flows* are direct manipulations of the information, such as copying a variable, or computing a value from a variable. *Implicit flows* happen as a byproduct of the manipulation of the information, and are harder to detect. For instance, the assignment $\texttt{y} = \texttt{x}$ induces an explicit flow from `x` to `y`. By contrast, `if` $\texttt{x} > 0$ `then` $\texttt{y} = 0$ induces an indirect flow from `x` to `y`, since the value of `y` indirectly depends on `x`. If we observe that the value of `y` is not 0, we learn that $\texttt{x} \leq 0$.

Using the notion of explicit and implicit flows, we can define policies that establish which behaviors are allowed in a system. For example, an information flow policy might stipulate that "neither explicit nor implicit information flows cause the leakage of confidential information". Another example is a policy stating that "malicious inputs never reach sensitive functions via

explicit flows".

It is possible to formalize these properties by introducing well-established *semantic properties*. A semantic property is a characteristic of a program that relates to its execution, rather that its syntax. The work of Goguen and Meseguer in 1982 defines a semantic property called *noninterference* [Goguen, 1982], which addresses the leakage of confidential information through implicit and explicit information flows. The work of Volpano in 1999 defines *weak secrecy*, a weaker semantic property, similar to noninterference, that ignores implicit flows. We can use these two semantic properties to formalize the information-flow policies presented earlier.

In the next subsections we study these two semantic properties further.

1.1 Weak secrecy

Weak secrecy [Volpano, 1999] is designed to model explicit flows of information. Taken from the definition of Volpano, variables are separated into *high* or *low*, where *high* corresponds to sensitive information. By sensitive, we mainly refer to two cases: confidential information such as passwords, but also information that might be originated from an untrusted source, such as a user input. In essence, we do not associate high or low with the value, but with the origin of the value. Finally, throughout the execution, the attacker can only observe low variables. Thus, for a program to be weak secret, high values must never flow into low variables.

This property is of specific interest because it can be adapted to describe several information-flow policies. For instance, let us assume there is a system that is performing SQL queries to a database. Then, we consider the policy stating that "a SQL injection attack cannot be performed". The high value is an external input that is stored in the program. However, instead of determining if the program is secure or not based on the assignment of low variables, we focus on the function performing the SQL query. The SQL query function acts as a sink in a taint analysis. Thus, the program is secure if the external input never flows to the SQL query function.

Determining whether a program satisfies weak secrecy can be performed by checking the absence of illegal flows for each individual execution. This kind of properties that depend on individual executions are called *trace properties*. Trace properties can be described as *sets of execution traces*.

1.2 Noninterference

Noninterference [Goguen, 1982] is a semantic property that considers both explicit and implicit flows of information, in contrast to weak secrecy.

In noninterference, the attacker can observe the low variables in the memory at the beginning and end of the execution. Assuming the memory is partitioned in high and low variables, for a program to be noninterferent, the high values must not affect the low variables in the execution, neither explicitly nor implicitly.

Weak secrecy can be expressed as a trace property. However, many important semantic properties such as noninterference cannot be expressed in that manner. To characterize noninterference, we need to consider pairs of executions. By taking two executions that agree on low

variables, the program is noninterferent if the final memories still agree on low variables. That is, the executions are indistinguishable for the attacker. The noninterference property is a *2-safety property* [Clarkson, 2008], meaning that it is a safety property requiring the observation of two simultaneous executions.

2 Verification methods

Seeing the existence of vulnerabilities, our goal is to eliminate them whenever possible. Different techniques are used for this purpose, but none of them is perfect.

Testing is widely adopted due to its ease of use, while providing good confidence in the functionality of code. Several tools exist to automate the process of writing test-cases while providing high coverage [Cadar, 2008; Cadar, 2021]. However, testing cannot provide full coverage, leaving vulnerabilities undetected.

Model checking [Merz, 2001] is another technique that has been broadly adopted. A model checker works by taking a model—an abstraction of the system—and a set of properties to verify. These properties are often expressed in temporal logics. If the model checker succeeds, it assures that the properties are satisfied in the model. When failing, it can provide counter-examples illustrating why the properties do not hold. The downside is that, since the model is an abstraction, the properties hold in the model but might not in the real system. Similarly, the counter-examples apply to the model, but might not be counter-examples in the real system.

Static analysis works by automatically examining source code without executing it, deriving information about the semantics of the program. In contrast to model checking, this technique does not require the definition of an abstract model of the system, removing potential human errors. Unlike testing, it can provide full coverage by analyzing the semantics of the program, formally verifying the property of interest.

Given a language and a property of interest, an ideal static analyzer would always answer whether a program satisfies or not the property in finite time. If we take the example from the previous section with the policy "neither explicit nor implicit information flows cause the leakage of confidential information", we would like to be able to determine for any program that the policy is never violated, without human intervention. Unfortunately, this is not possible for Turing-complete languages as shown by Rice's Theorem [Rice, 1953].

Rice's Theorem is a fundamental result in computability theory that provides insights into the undecidability of program properties, stating that, for a Turing-complete language, any non-trivial semantic properties of programs are undecidable. A property is non-trivial if it holds for some programs, and does not for others.

Ideal analyzer

Let $\mathscr{P}$ be a non-trivial semantic property. Let p be a program in a Turing-complete language. Let `analyzer` be an analyzer that takes a program as input and responds A (as in "accept")

or R (as in "reject") in finite time. We say `analyzer` is *ideal* if

$$(\texttt{analyzer}(p) = A) \Leftrightarrow p \text{ satisfies } \mathscr{P}.$$

Rice's theorem implies that this double implication cannot hold. Instead, by decomposing it, we arrive at two core concepts of static analysis which we call *soundness* and *completeness*.

$$\begin{aligned} \text{Soundness:} \quad & (\texttt{analyzer}(p) = A) \Rightarrow p \text{ satisfies } \mathscr{P} \\ \text{Completeness:} \quad & (\texttt{analyzer}(p) = A) \Leftarrow p \text{ satisfies } \mathscr{P} \end{aligned}$$

Depending on the property of interest and the use-case, the user must choose whether to have a sound or complete static analyzer.

Sound static analyzers are appropriate to verify properties. When the analyzer returns "accept", we have a guarantee that the program holds for the property of interest. Instead, a complete analyzer is appropriate for refuting a program. We can transform the completeness definition to be more direct:

$$\texttt{analyzer}(p) = R \Rightarrow p \text{ does not satisfy } \mathscr{P}.$$

We provided a universal understanding of the limitations associated with static analyzers. However, we say nothing about the core approaches to perform the static analysis of programs. As the name suggests, a static analyzer does not execute the program in the conventional way, but it can be seen as executing the program with an "abstraction" of the semantics. We are particularly interested in two approaches: *symbolic execution* [Boyer, 1975; King, 1976; Cadar, 2013], where concrete values are replaced by mathematical symbols, and the control flow of the program generates arithmetic or logical constraints that can be tested; and *abstract interpretation* [Cousot, 1977] where the concrete values are replaced by an abstract domain that retains the relevant information of the execution, accompanied by an abstract semantics.

2.1 Symbolic execution based static analysis

Symbolic execution is a technique that relies on replacing concrete values in a program execution for *symbolic values*. Symbolic values are mathematical symbols that can represent several values simultaneously.

To build a symbolic executor, it is not enough to swap the concrete values for symbolic ones. A *symbolic state* must contain the map from variables to symbols, and a set of constraints for the symbols that are mapped. For example, if we take the program `if x > 0 then y = 0`, the initial state maps `x`, and `y` to two different symbols: $[\texttt{x} \mapsto \boldsymbol{x}; \texttt{y} \mapsto \boldsymbol{y}]$. We call this a *symbolic store*. When evaluating the guard of the `if` statement, nothing is known about symbol $\boldsymbol{x}$. Thus, it is possible to take both branches of the `if`. However, the symbolic state must contain information to reflect which path was chosen. At the current example, by assuming the guard is satisfied, a symbolic constraint $\boldsymbol{x} > 0$ is generated. In turn, it is stored in what we call the *symbolic path*. Finally, a symbolic state is a pair composed of a symbolic store and a symbolic path.

Figure 1.1: A lattice of confidentiality levels with two levels: high and low.

Symbolic execution annotates all the path information, allowing for a very precise description of the program semantics. For that reason, symbolic execution analyzers are usually complete and notably utilized for finding vulnerabilities, such as in KLEE [Cadar, 2008]. To do so, symbolic execution discharges queries to an SMT solver, that checks if the constraints in the symbolic path are satisfiable. If so, the SMT solver can produce a counter-example.

In the previous example, we supposed that the symbolic execution analysis chose the true-branch of the `if` statement. However, this would disregard all executions that follow the false-branch. In order to cover the largest amount of the concrete semantics of the program, the symbolic executor must collect different paths. Yet, collecting different paths is not costless. At each control-flow statement, the state must be duplicated to explore the different paths, leading to what is called *state-explosion*.

The problem of state-explosion is even worse when reasoning about loops. Each iteration of a loop requires the duplication of the state. Physical limitations of computers imply that only so many iterations of a loop can be performed. Moreover, a condition such as $\mathtt{y} < \mathtt{z}$ might always be satisfiable in symbolic execution if the value of $\mathtt{z}$ is unknown.

As mentioned, one of the key aspects of static analysis is that it must terminate in finite time for any program. Hence, we cannot allow loops to iterate freely. A way to deal with the boundless iteration of loops is to set a threshold, limiting the amount of times a loop can be accessed. After the bound of iterations is met, the trace is dropped, ensuring the termination of the analysis.

Noninterference is a hyperproperty, meaning that we need to observe simultaneous traces in order to study it. Symbolic execution can be adapted to a *relational symbolic execution* [Farina, 2019] for this purpose. In the case of noninterference, the basis of relational symbolic execution is to map variables to two symbols. The agreement of low variables can be done by constraining the symbols to be equal at the beginning of the execution.

2.2 Abstract interpretation

In a concrete execution, the program state is represented by a map from variables to concrete values, a memory. Instead, in an *abstract interpretation* [Cousot, 1977], the program state is represented by an *abstract element*. The set of all abstract elements is called an *abstract domain*. For the case of confidentiality, we can define the dependences abstract domain that abstracts variables to their security level on a security lattice. More specifically, by focusing on the lattice of Figure 1.1, the abstract elements are sets of constraints of the form $\mathbb{H} \to \mathtt{x}$ and $\mathbb{L} \to \mathtt{x}$ where $\mathtt{x}$ is a variable. Finally, the dependences domain is the set of all abstract elements of this shape.

Using the dependences abstraction, the concrete values are stripped down to their security level. However, discarding information causes the abstraction to be less precise. The abstraction must be picked carefully, to contain enough information to successfully analyze programs.

This loss of precision can be observed through a *concretization*. We define concretizations as follows: given an abstract element a, the concretization is the set of program states that satisfy it, denoted by $\gamma(a)$. In the case of Figure 1.1, the concretization of $\{\mathbb{H} \rightarrow \mathtt{x}\ ;\ \mathbb{L} \rightarrow \mathtt{y}\}$ is the set of all memories where `x` is high and `y` is low. Any other information has been disregarded, leaving us with a much more general description of concrete states.

An abstract semantics, is a set of semantic rules where the state is an abstract element that gets refined with each execution step. The two main operations that abstract semantics relies on are **join** and **widen**. Operation **join** takes two abstract elements and produces a single abstract element that contains both original states. This operation is used, for instance, for analyzing `if` statements. For ensuring the termination of loops, the **widen** operation is used. This operation ensures finding a fix-point of the loop in finite-time.

Both of these operations are sound, meaning that they ensure the full coverage of the program semantics. What this implies, is that these operations perform over-approximations. Hence, abstract interpretation is imprecise not only because of the abstract domain not being descriptive enough, but also the way abstract semantics are defined.

2.3 Comparison and research question

Both symbolic execution and abstract interpretation can be employed to verify program properties. The precision of symbolic execution allows to analyze the semantics of a program closely to its concrete execution. This also allows generating traces that trigger the property violation. However, this precision comes at the cost of potential exponential growth in memory, coupled with the immense computational time associated with SMT solvers.

Abstract interpretation, focuses on describing states in a simpler, concise manner, allowing for faster execution times, and uses mechanisms such as **join** and **widen** to ensure termination, and to not multiply abstract states such as symbolic execution. This approach can work extremely well, but the loss of information usually implies the impossibility of bug-finding.

In essence, we see abstract interpretation being more inclined to perform sound analysis of programs, and symbolic execution being more inclined to perform complete analysis of programs. Divided between these two techniques, in essence, the developer is forced to choose either soundness or completeness.

Research questions Symbolic execution and abstract interpretation based analyses are very different; the two methods seem opposed in their approach. Indeed, this is what prompts the following questions:

- Symbolic execution does not over-approximate the program semantics, and can only be unrolled up to a threshold. If a loop needs to be unrolled more times than the allowed limit, the analysis fails. Meanwhile, abstract interpretation assures termination of the analysis

```
1 # 𝕙 → priv ; 𝕝 → y
2 if (priv > 0):
3     y = 5
4 else:
5     y = 5
```

(a) Noninterferent program: fails with dependencies.

```
1 # 𝕙 → priv ; 𝕝 → i,z
2 i = 0
3 while (i < z):
4     i += 1
5     priv += 2
```

(b) Noninterferent program: fails with symbolic execution.

```
1 # 𝕙 → priv ; 𝕝 → i,y
2 while (i < priv):
3     if (i > priv):
4         log(priv)
5     i += 1
```

(c) Explicit secret program: generates false alarm.

```
1 # 𝕙 → priv ; 𝕝 → y
2 if 0 == y**2 + y*2 - 8:
3     log(priv)
```

(d) Not explicit secret program: requires symbolic execution.

Figure 1.2: Motivating examples. All variables are of type `int`.

by using operations such as **widen**. *At the cost of completeness, can we use abstract interpretation as an over-approximation technique on top of a symbolic execution analysis to achieve soundness?*

- While abstract interpretation is useful to verify programs, when a program is rejected the user does not get a satisfactory answer. Instead, it causes the user to spend time understanding why the abstract interpretation rejected the program, and then manually verify whether there is a real bug, or if it is just a false-positive. Meanwhile, symbolic execution is suited to return possible input values, guiding the user to the specific trace that violates the property. *Is it possible, once the abstract interpretation analyzer has been executed, to refine its output through symbolic execution?*

In the following sections, we aim to delve deeper into the proposed questions. We take on the first question in Section 3, and the second question in Section 4

3 Sound symbolic execution via abstract interpretation

Relational symbolic execution based static analysis for noninterference Standard symbolic execution is usually complete. However, to verify a property as noninterference, we need not only a relational symbolic execution, but a sound one. Examples 1.2a and 1.2b illustrate the problem.

In Example 1.2a, variable `priv` stores confidential information. Assuming `y`stores value y. In any execution of this program, `y` stores 5. Using a relational symbolic execution Here, relational symbolic execution can safely check all possible interleavings: following true in both executions, following false in both executions, and one going to the true branch while the other execution goes to the false branch. For instance, one of the diverging paths produces constraints $(priv_0 > 0 \land priv_1 \leq 0)$. In this example it is possible to check all possible paths.

In Example 1.2b, variable `i` is initialized at 0. The value of `z` is unknown, and there is a loop with the guard `i < z`, in which variables `i` and `priv` are augmented by 1 and 2 correspondingly. This program is noninterferent, as the final value of `i` is z when $z > 0$ or 0 otherwise. However, since the value of `z` is unknown, the loop can be analyzed infinitely many times. This results in the impossibility to verify the program with relational symbolic execution.

The alternative is to make the relational symbolic execution sound by over-approximating loops. Nevertheless, if the loop over-approximation is not *precise enough*, the analyzer fails to verify programs. Let us explore a possible over-approximation approach for loops:

- first, by analyzing the body of the loop it is possible to approximate which variables might be modified in the execution of the loop;
- once the modifiable variables have been calculated, assign *fresh* (previously unused) symbols to those variables;
- finally, add the negation of the guard as a symbolic constraint.

The problem with this approach is that, by mapping variables to two symbols, it renders the verification of noninterference impossible. This happens since we cannot establish the equality between the new symbols without extra information. In the case of Example 1.2b, applying this method makes `i` be mapped to two symbols. Since the two symbols cannot be proved equal, the program cannot be verified.

Dependences based static analysis for noninterference Many static analyses that work for noninterference rely on some form of dependence abstraction as formalized in, e.g., [Assaf, 2017] or [Hunt, 2006]. For our purpose, we assume an ordered set of security levels $\{\mathbb{L}, \mathbb{H}\}$ and that each value fed into a program via an input variable is given a security level. A dependency, noted as $l \rightarrow \mathtt{x}$ with $l \in \{\mathbb{L}, \mathbb{H}\}$, expresses the agreement of `x` in both executions when observing from level l. The dependences analysis has a dependences state, defined to be a mapping from variables to security levels.

For Example 1.2a, the analysis determines that the assignments are conditioned by the value of `priv`, which is initially high. Since dependences only keep track of the security level of a variable, the assignment for `y` to 5—a constant—is low, but these assignments happen inside an `if` statement with a guard that depends on high variable `priv`. Thus, the dependency $\mathbb{L} \rightarrow \mathtt{y}$ is dropped, indicating the potential disagreement of `y`. Therefore, dependences cannot determine that the program is noninterferent. In Program 1.2b, the loop condition is only influenced by `i` and `z`, which are low. The assignment of low variables is not affected, and `i` and `z` remain low, allowing to prove noninterference, in contrast to the relational symbolic execution.

3.1 Problem

As observed, both relational symbolic execution and dependences can be used for the verification of noninterference. Neither approach is perfect—dependences being sound but imprecise, relational symbolic execution unable to handle loops properly, even if it performs a sound over

approximation—but in Examples 1.2a and 1.2b, these techniques complement each other. This raises the following question: Can we combine dependences and relational symbolic execution to enhance overall precision?

3.2 Contribution I

In this manuscript, we design a combination of relational symbolic execution and abstract interpretation based analyses for the verification of noninterference. The abstract interpretation based analysis is used to refine the over approximation of loops, by passing information to the symbolic execution state. In this thesis, we investigate this technique by using a dependences domain, but also non-relational domains such as intervals. To do so, we first formalize the relationship between dependences and relational symbolic execution, and define two functions: to transform a relational symbolic state into a dependences and state, and to extract constraints from a dependences state to use in relational symbolic execution. We can exploit the constraints generated by dependences and smartly choose how to over approximate the variables.

4 Symbolic execution over abstract interpretation outputs

Weak secrecy can be utilized to formalize the behavior of taint trackers. Pysa [Meta, 2023], a taint-tracker developed by Meta, is such a case of a tool that closely follows weak secrecy. Pysa is an open-source static analyzer, based in abstract interpretation, that performs taint-tracking in Python, and is used at a production level in Meta.

Pysa is designed to analyse immense codebases. To do so, it must perform various over approximations. These over approximations involve keeping track solely of taints, instead of values, even collapsing structures to fit in memory. For instance, if an element in a list is tainted, Pysa might assume that all the elements are tainted.

These aspects lower the precision of the tool. Indeed, these over approximations cause the tool to generate false-positves. Hence, developers must manually check each alarm, and decide whether the alarm is reporting a real problem, or must be disregarded. Needless to say, this process is not optimal.

For the following examples we assume that function `log` is a sink in Pysa. In Example 1.2c, function `log` can never be executed, since the intersection of the guard of the loop and `if` is empty. However, Pysa assumes that the both the guard of the loop and the `if` are satisfiable simultaneously. This leads to assuming that `log(priv)` is executed, resulting in an alarm. Instead, using symbolic execution, the SMT solver determines that this path is unfeasible, concluding that the program is secure.

In Example 1.2d, function `log` is reachable but only if the value of `y` is the positive root of the polynomial $y^2 + 2y - 8$. This means that the program is not weak secret. In Pysa, an alarm is raised. However, Pysa does not provide us input values to trigger the trace. Instead, the symbolic execution can generate constraints and pass them onto an SMT solver. Since the guard expression is a second degree polynomial, the SMT returns that the input value for `y` is 2. Hence, we can use symbolic execution to decide whether some alarms are real, and get counter-examples.

4.1 Problem

Sound taint analyzers such as Pysa are very powerful. These raise alarms that might be real or not, and the process of filtering these alarms is not direct. This motivates the following question: *can a sound taint analyzer be combined with a symbolic execution analysi, refining its results by detecting false alarms and generating counter-examples?*

4.2 Contribution II

We propose a complete symbolic execution taint analysis that can be used to find counter-examples, and to refute false-alarms. We illustrate a combined taint analysis, that uses a sound taint analysis in conjunction with our symbolic taint analysis. The combines taint analysis consists on executing the sound taint analysis and attempting to automatically filter the alarms generated. Assuming that sound taint analysis produces an alarm, the symbolic taint analysis will attempt to determine whether the alarm is real or a false-positive.

On top of that, we present Pysta, our prototype symbolic taint analysis tool. Our tool aims to instantiate the combined taint analysis, to show how the output of a tool such as Pysa can be utilized to guide a more precise symbolic taint analysis, refining the more imprecise tool.

5 Outline of the thesis

The thesis is divided into two parts. Part I expands on Section 3 from the introduction. In Chapter 3, we implement a sound non-relational symbolic execution. In Chapter 4, we define the reduced product of the semantics of I.2 with an intervals abstract domain, to raise precision of loop overapproximation. In Chapter 5, we define a sound relational symbolic execution, that can be instantiated with the non-relational symbolic execution of sections I.2 and I.3. In Chapter 6, we define the main contribution: the reduced product between the semantics of I.4 and dependences. In Chapter 7, we discuss the implementation of the analysis and we evaluate it in a set of small but challenging programs. Finally, in Chapter 8 we discuss related work, and conclude in **??**.

Part II expands on Section 4. In Chapter 10, we formalize taints and present a formal concrete semantics for a taint-tracker. In Chapter 11 we adapt weak secrecy for our more general taints and language. In Chapter 12, we present our main contribution of this part, a complete symbolic taint-tracker semantics, and we present a proof of completeness. In Chapter 13, we present the combined analyzer between a sound and our symbolic taint-tracker. In Chapter 14, we discuss the implementation and its limitations, showing our tool Pysta and Pysa, and instantiating the combination of Chapter 13. In Chapter 15 we evaluate the tool on a set of examples. In Chapter 16, we discuss related work, and we conclude in **??**.

Part I

Combination of Relational Symbolic Execution and Dependences for Noninterference

Chapter 2

Towards sound symbolic execution via abstractions

Usually, symbolic execution is not sound, leading to it being used as a testing technique, rather than a verification technique. In this part of the manuscript we will develop sound over-approximations of loops, with and without, abstractions. In the non-relational symbolic execution we will use a standard numerial abstraction. Instead, for relational symbolic execution we will use a dependences abstractions. Finally, our goal is to verify noninterferent programs with our sound relational symbolic execution.

1 Sound non-relational symbolic execution

In Chapter 3, we first present the standard semantics of symbolic execution. Two main problems arise: the semantics are not relational, and they are not sound. The lack of soundness implies that the semantics are not immediately fit for the verification of programs. For that matter, the chapter first focuses on creating a broad over-approximation of loops, that we call SoundSE. The over-approximation works by first checking which variables *might* be modified in the body of the loop. Then, those variables must be assigned new fresh symbolic values. For instance, let us focus on the noninterferent Program 1.2b. Let us assume that the initial value of `z` is 10. If the threshold of loop unrollings is lower than 10, the value of `i` and `priv` must be over-approximated, resulting in the mapping $\texttt{i} \mapsto i_1$, with the only constraint for i_1 being that is greater than 10. Hence, it is not possible to determine the precise value of i.

In Chapter 4, the goal is to raise the precision of SoundSE through numerical abstractions, that allow us to infer interesting information from loops. One of the core concepts is that of *reduced product* between our symbolic and abstract states. A reduced product boils down to defining a new state that contains both original states, adapting the semantics to execute with this new state, and mainly, defining a *reduction function*. The reduction function will take the new state and will "share" information between the abstract and symbolic state, generating new constraints; in other words, raising the precision of the symbolic state. We call this semantics

RedSoundSE. In Program 1.2b, a convex polyhedra analysis, is able to determine that `i` = `z` if the loop is accessed, and `i` = 0 otherwise.

Semantics SoundSE and RedSoundSE are non-relational semantics, meaning that we cannot use them to do verification of noninterference. What we can do, is use them to define a relational symbolic execution, as we will explain in the following section.

2 Sound relational symbolic execution

In Chapter 5, we first present a standard relational symbolic execution semantics that is not sound as it cannot over-approximate loops. Similarly to Chapter 3, in the chapter we will adapt the semantics to be sound, but imprecise. The sole difference in this approach will be that, now the relational symbolic execution must assign a pair of symbolic values, encompassing both executions. SoundRSE can instance either SoundSE or RedSoundSE. Indeed, Program 1.2b will behave exactly as in SoundSE if we instance it, and the value of `i` will not be precise enough to establish the equality of it between the two executions. However, this relational analysis, named SoundRSE, can verify *some* programs given its relational nature. By instancing RedSoundSE, the value of `i` can be determined to be 10 for any execution accessing the loop, and the program can be proved noninteferent.

To raise its precision, in Chapter 6 we introduce a dependences abstraction, that maps security levels to variables. The dependences abstraction can easily be connected to the symbolic states through, what we call, *translation* and *extraction* functions. These functions allow us to transform relational stores to dependences states, and then extract the dependences when required. Then, the over-approximation of loops narrows down to the analysis by dependences of the loop, and the extraction of the new dependences, which we consider a new analysis named RedSoundRSE. While SoundRSE was not able to verify Program 1.2b without the numerical abstraction, RedSoundRSE is. A dependences analysis will notice that the value of `i` is only constrained by `z`, which is low. Therefore, the value of `i` can only depend on `z`, and `priv` cannot interfere. Thus, the program is noninterferent.

3 Relationship between analyses

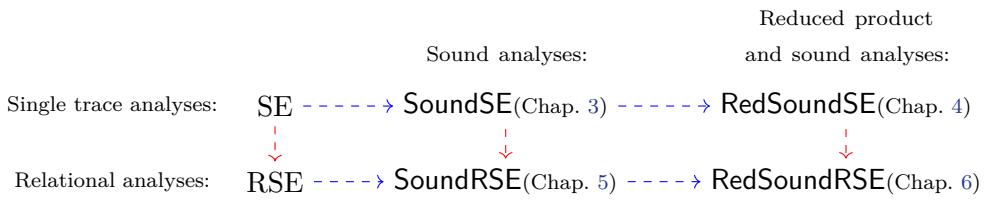


Figure 2.1: Relation between different symbolic execution analyses.

In Figure 2.1 we illustrate all the symbolic execution semantics from this part. Red dashed lines represent a dependency in terms of: a relational analysis depends on a single trace analysis. Blue dashed lines represent an "enhancement" of the analysis generally speaking. SE [Boyer, 1975]

is a conventional symbolic execution and RSE [Milushev, 2012; Palikareva, 2016] is its extension to relational properties. We avoid to use the term SE and RSE later in the manuscript. Instead, we will refer them as *standard symbolic execution* and *standard relational symbolic execution*, respectively. These two semantics are unsound in general, except for the work of [Farina, 2019]. The rest of the analyses are sound and are our contributions: SoundSE evolves from the standard symbolic execution (SE), and eventually reaches RedSoundSE through the addition of numerical abstractions. SoundRSE evolves from the standard relational symbolic execution (RSE), and using dependeces becomes RedSoundRSE. While in the graph it looks like RedSoundRSE its the relational upgrade of RedSoundSE, the red arrow here implies more of a moral dependency. In reality, RedSoundRSE can be instanced by either of the non-relational symbolic executions. Same applies for SoundRSE.

Chapter 3

SoundSE: Sound Symbolic Execution

In this chapter, our goal is to first define a standard non-relational symbolic execution, that will introduce all the concepts of symbolic execution. From there, we extend this symbolic execution with a rule to over-approximate loops after a threshold, making the symbolic execution sound. While the over-approximation will be imprecise, it is a first version, that we will gradually upgrade throughout the chapters. Also, the sound symbolic execution semantics will be instanced in later chapters for building a relational symbolic execution.

We will reintroduce the examples from Chapter 1, plus a new one that will guide us through this part.

1 Language syntax and semantics

Next we define a toy language that we will use for the whole part, and the concrete semantics that we will symbolically abstract.

1.1 Syntax

We let $\mathbb{V}$ and $\mathbb{X}$ be the set of values and program variables respectively, and $\oplus$, $\ominus$ be binary operators. The set of all expressions $\mathbf{e}$ is $\mathbf{E}$. The set of all boolean expressions $\mathbf{b}$ is $\mathbf{B}$. A statement $\mathbf{s}$ is either a skip, an assignment, a condition, or a loop. Finally, a command $\mathbf{c}$ is a finite sequence of statements.

$$\begin{array}{lcl} \mathbf{e} & ::= & n\ (n \in \mathbb{V}) \mid \mathtt{x}\ (\mathtt{x} \in \mathbb{X}) \mid \mathbf{e} \oplus \mathbf{e} \\ \mathbf{s} & ::= & \mathtt{skip} \mid x := \mathbf{e} \mid \mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}\ \mathtt{else}\ \mathbf{c} \mid \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c} \end{array} \qquad \begin{array}{lcl} \mathbf{b} & ::= & \mathbf{tt} \mid \mathbf{ff} \mid \mathbf{e} \ominus \mathbf{e} \mid \neg \mathbf{b} \\ \mathbf{c} & ::= & \mathbf{s} \mid \mathbf{s}; \mathbf{c} \end{array}$$

1.2 Semantics

A *concrete store* (a memory) is a function μ from program variables $\mathbb{X}$ to concrete values $\mathbb{V}$. We write $\mathbb{M} = \mathcal{P}(\mathbb{X} \to \mathbb{V})$ for the set of concrete stores and write $[\mathtt{x} \mapsto n, \dots]$ for an explicit store, where $n \in \mathbb{V}$.

$$\frac{}{\mu \vdash n \downarrow n} \qquad \frac{}{\mu \vdash \mathbf{x} \downarrow \mu(\mathbf{x})} \qquad \frac{\mu \vdash \mathbf{b} \downarrow b}{\mu \vdash \neg \mathbf{b} \downarrow \neg b}$$

$$\frac{\mu \vdash \mathbf{e}_1 \downarrow n_1 \qquad \mu \vdash \mathbf{e}_2 \downarrow n_2 \qquad n_3 \triangleq n_1 \oplus n_2}{\mu \vdash \mathbf{e}_1 \oplus \mathbf{e}_2 \downarrow n_3} \qquad \frac{\mu \vdash \mathbf{e}_1 \downarrow n_1 \qquad \mu \vdash \mathbf{e}_2 \downarrow n_2 \qquad b \triangleq n_1 \oslash n_2}{\mu \vdash \mathbf{e}_1 \oslash \mathbf{e}_2 \downarrow n_3}$$

Figure 3.1: Concrete evaluation of expressions

We define the evaluation of expressions $(\downarrow) : (\mathbf{E} \cup \mathbf{B}) \to \mathbb{V} \cup \{\mathbf{tt}; \mathbf{ff}\}$ with a context $\mu \in \mathbb{M}$ in big-step style in Figure 3.1. We denote $\mu \vdash \mathbf{e} \downarrow n$ the evaluation of $\mathbf{e}$ into value n with concrete store μ. We denote $\mu \vdash \mathbf{b} \downarrow b$ the evaluation of $\mathbf{b}$ into boolean b with concrete store μ.

$$\frac{\mu \vdash \mathbf{e} \downarrow n}{(\mathbf{x} := \mathbf{e}, \mu) \to (\texttt{skip}, \mu[\mathbf{x} \mapsto n])}$$

$$\frac{(\mathbf{c}_1, \mu) \to (\mathbf{c}_1', \mu')}{(\mathbf{c}_1; \mathbf{c}_2, \mu) \to (\mathbf{c}_1'; \mathbf{c}_2, \mu')} \qquad \frac{}{(\texttt{skip}; \mathbf{c}_2, \mu) \to (\mathbf{c}_2, \mu)}$$

$$\frac{\mu \vdash \mathbf{b} \downarrow \mathbf{tt}}{(\texttt{if } \mathbf{b} \texttt{ then } \mathbf{c}_1 \texttt{ else } \mathbf{c}_2, \mu) \to (\mathbf{c}_1, \mu)} \qquad \frac{\mu \vdash \mathbf{b} \downarrow \mathbf{ff}}{(\texttt{if } \mathbf{b} \texttt{ then } \mathbf{c}_1 \texttt{ else } \mathbf{c}_2, \mu) \to (\mathbf{c}_2, \mu)}$$

$$\frac{\mu \vdash \mathbf{b} \downarrow \mathbf{tt}}{(\texttt{while } \mathbf{b} \texttt{ do } \mathbf{c}, \mu) \to (\mathbf{c}; \texttt{while } \mathbf{b} \texttt{ do } \mathbf{c}, \mu)} \qquad \frac{\mu \vdash \mathbf{b} \downarrow \mathbf{ff}}{(\texttt{while } \mathbf{b} \texttt{ do } \mathbf{c}, \mu) \to (\texttt{skip}, \mu)}$$

Figure 3.2: Concrete step relation

A *concrete state* is a pair $(\mathbf{c}, \mu) \in \mathbf{C} \times \mathbb{M}$ of a command and a store. Concrete semantics of the toy language is defined in small-step fashion as a step relation $(\to) : \mathbf{C} \times \mathbb{M} \to \mathbf{C} \times \mathbb{M}$, defined in Figure 3.2.

2 Standard symbolic execution

The core principle of symbolic execution is to map program variables into expressions made of *symbolic values* that denote the initial value of the program variables. Then, instead of finishing the execution with concrete values, the execution accumulates constraints that reflect, symbolically, the semantics of the program. To start, we need to define symbolic values that replace concrete values, and symbolic stores, the counterpart of concrete stores.

2.1 Symbolic expressions and stores

We let $\overline{\mathbb{V}} = \{\boldsymbol{x}, \boldsymbol{y}, \ldots\}$ denote the set of symbolic variables and note for clarity $\boldsymbol{x}$ the symbolic variable associated to program variable $\mathbf{x}$ (not to be confused with concrete values). A *symbolic store* is a function ρ from program variables to *symbolic expressions* the set of which is noted $\mathbb{E}$, namely expressions defined like the programming language expressions using symbolic values

instead of program variables. We write $\overline{\mathbb{M}} = \mathcal{P}(\mathbb{X} \to \mathbb{E})$ for the set of symbolic stores and write $[\mathtt{x} \mapsto (\boldsymbol{x}), \ldots]$ for an explicitly given symbolic store. To tie properly symbolic stores and concrete stores, we need to relate symbolic values and concrete values. To this end, we let a *valuation* be a function $\nu : \overline{\mathbb{V}} \longrightarrow \mathbb{V}$. Moreover, given a symbolic expression ε, we let $[\![\varepsilon]\!]$ be a partial function that maps a valuation ν to the value obtained when evaluating the expression obtained by replacing each symbolic value $\boldsymbol{x}$ in e with $\nu(\boldsymbol{x})$. We can now express the concretization of symbolic stores.

Definition 1 (Symbolic store concretization)**.**

$$\begin{array}{rccl} \gamma_{\overline{\mathbb{M}}} : & \overline{\mathbb{M}} & \longrightarrow & \mathcal{P}(\mathbb{M} \times (\overline{\mathbb{V}} \to \mathbb{V})) \\ & \rho & \longmapsto & \{(\mu, \nu) \mid \forall \mathtt{x} \in \mathbb{X},\ \mu(\mathtt{x}) = [\![\rho(\mathtt{x})]\!](\nu)\} \end{array}$$

Example 1 (Symbolic store and concretization)**.** *We reintroduce Program 1.2a and 1.2b.*

```
1  # 𝕳 → priv ; 𝕷 → y
2  if (priv > 0):
3      y = 5
4  else:
5      y = 5
```

```
1  # 𝕳 → priv ; 𝕷 → i,z
2  i = 0
3  while (i < z):
4      i += 1
5      priv += 2
```

In Program 1.2a, all executions will generate the same symbolic store $\rho = [\mathtt{y} \mapsto 5; \mathtt{priv} \mapsto \boldsymbol{priv}]$*. A possible concretization is*

$$\mu = [\mathtt{y} \mapsto 5; \mathtt{priv} \mapsto 0] \qquad \nu = [\boldsymbol{priv} \mapsto 0]$$

In Program 1.2b, a possible final store is $\rho = [\mathtt{i} \mapsto 10; \mathtt{z} \mapsto \boldsymbol{z}; \mathtt{priv} \mapsto \boldsymbol{priv} + 20]$*, which happens after executing the loop 10 times. The concretization is* $\gamma_{\overline{\mathbb{M}}}(\rho)$ *an infinite set of stores and valuations. For instance, a pair* $(\mu, \nu) \in \gamma_{\overline{\mathbb{M}}}(\rho)$ *could be*

$$\mu = [\mathtt{i} \mapsto 10; \mathtt{z} \mapsto 10; \mathtt{priv} \mapsto 20] \qquad \nu = [\boldsymbol{z} \mapsto 10; \boldsymbol{priv} \mapsto 0]$$

Notice that, while a valid store and valuation have been given (according to the concrete semantics), the definition of the concretization might lead to generating spurious stores (stores that do not belong to any concrete execution). This happens because there is no connection between `z` *and* `i`*, as can be seen in the symbolic store. For instance,*

$$\mu = [\mathtt{i} \mapsto 10; \mathtt{z} \mapsto 0; \mathtt{priv} \mapsto 20] \qquad \nu = [\boldsymbol{z} \mapsto 0; \boldsymbol{priv} \mapsto 0]$$

also belongs to the concretization of ρ*.*

In the next subsection, we constrain the symbolic store and define a new concretization function. This way, we can overcome the limitations of $\gamma_{\overline{\mathbb{M}}}$, getting rid of spurious stores.

2.2 Symbolic path and precise store

To precisely characterize the outcome of an execution path, a symbolic store is too abstract. Hence, symbolic execution also utilizes a symbolic expression to constrain the store, referred to as *symbolic path*, that accounts for the conditions encountered during a path.

A *symbolic boolean expression* is a symbolic expression that can be evaluated to true or false. The set of all symbolic boolean expressions is denoted by $\mathbb{B}$. A *symbolic path* π is a set of symbolic boolean expressions. We assume that for a given path π, the elements of the set are separated by a conjunction. A *symbolic precise store* is a pair $\kappa = (\rho, \pi)$ where $\rho \in \overline{\mathbb{M}}$ and $\pi \in \mathcal{P}(\mathbb{B})$. We write $\mathbb{K}$ for the set of symbolic precise stores.

We overload the $[\![\cdot]\!]$ operator for symbolic paths, where $[\![\pi]\!]$ is a partial function from valuations to true or false.

Definition 2 (Symbolic precise store concretization)**.** *The* symbolic state concretization $\gamma_{\mathbb{K}}$ *is defined by:*

$$\begin{array}{rccl} \gamma_{\mathbb{K}} : & \mathbb{K} & \longrightarrow & \mathcal{P}(\mathbb{M} \times (\overline{\mathbb{V}} \to \mathbb{V})) \\ & (\rho, \pi) & \longmapsto & \{(\mu, \nu) \in \gamma_{\overline{\mathbb{M}}}(\rho) \mid [\![\pi]\!](\nu) = \boldsymbol{tt}\} \end{array}$$

Example 2 (Symbolic precise store and concretization)**.** *In Program 1.2a, having the symbolic path only helps to differentiate if the initial value of* `priv` *is greater than 0 or otherwise. If* $\pi \triangleq \boldsymbol{priv} > 0$*, then*

$$\mu = [\mathtt{y} \mapsto 5; \mathtt{priv} \mapsto 0] \qquad \nu = [\boldsymbol{priv} \mapsto 0]$$

is not in the concretization of that precise store, but it is for the other branch.

We consider Program 1.2b. Previously, in Example 1, we proposed the symbolic store $\rho = [\mathtt{i} \mapsto 10; \mathtt{z} \mapsto \boldsymbol{z}; \mathtt{priv} \mapsto \boldsymbol{priv} + 20]$*. But, without the symbolic path, the concretization generates spurious concrete stores. For example,*

$$\mu = [\mathtt{i} \mapsto 10; \mathtt{z} \mapsto 0; \mathtt{priv} \mapsto 20] \qquad \nu = [\boldsymbol{z} \mapsto 0; \boldsymbol{priv} \mapsto 0]$$

is in the concretization of ρ*. Given that* $\rho(\mathtt{i}) = 10$*, it must be the case that the value of* $\boldsymbol{z}$ *is 10, and the loop was executed 10 times. Then, a possible symbolic path is* $\pi \triangleq 10 = \boldsymbol{z}$*. Hence, while* $(\mu, \nu) \in \gamma_{\overline{\mathbb{M}}}(\rho)$*, adding the symbolic path rules out the pair:* $(\mu, \nu) \notin \gamma_{\mathbb{K}}(\rho, \pi)$*.*

2.3 Symbolic execution step

The main piece of the symbolic execution algorithm is the step relation, which closely follows the small step semantics of the programs. We define it by a transition relation $\rightharpoonup_s$ between *symbolic execution states* that are made of a program command and a symbolic precise store. To do so, we will first define the evaluation of expressions, and then the step semantics.

Symbolic evaluation We define the symbolic evaluation of an expression or condition in a symbolic store, which produces a symbolic expression. We note $\rho \vdash \mathbf{e} \Downarrow \varepsilon$ the evaluation of $\mathbf{e}$ into symbolic expression ε in symbolic store ρ. Usually, this evaluation step boils down to the

substitution of the variables in $\mathbf{e}$ with the symbolic expressions they are mapped to in ρ, possibly with some simplifications.

Example. *For instance, let us assume expression* $\boldsymbol{e} = \mathtt{x} + \mathtt{y}$*, and a symbolic map* $\rho = [\mathtt{x} \mapsto (\boldsymbol{x}); \mathtt{y} \mapsto (\boldsymbol{x} + 10)]$*. Then, the evaluation is as follows*

$$\rho \vdash \boldsymbol{e} \Downarrow (\boldsymbol{x} + \boldsymbol{x} + 10)$$

Satisfiability of constraints Second, we define the conservative satisfiability test of a symbolic path. This step is usually performed by an external tool such as an SMT solver, so we do not detail its internals here. We note that this test may conservatively return as a result that a symbolic path *may* be satisfiable. We note $\mathbf{may}(\pi)$ when π may be satisfiable.

$$\textsc{s-assign}\ \frac{\rho \vdash \mathbf{e} \Downarrow \varepsilon}{(\mathtt{x} := \mathbf{e}, (\rho, \pi)) \rightharpoonup_s (\mathtt{skip}, (\rho[\mathtt{x} \mapsto \varepsilon], \pi))}$$

$$\textsc{s-seq-exit}\ \frac{}{(\mathtt{skip}; \mathbf{c}_1, \kappa) \rightharpoonup_s (\mathbf{c}_1, \kappa)} \qquad \textsc{s-seq}\ \frac{(\mathbf{c}_0, \kappa) \rightharpoonup_s (\mathbf{c}_0', \kappa')}{(\mathbf{c}_0; \mathbf{c}_1, \kappa) \rightharpoonup_s (\mathbf{c}_0'; \mathbf{c}_1, \kappa')}$$

$$\textsc{s-if-t}\ \frac{\rho \vdash \mathbf{b} \Downarrow \beta \qquad \pi' \triangleq \pi \wedge \beta \qquad \mathbf{may}(\pi')}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1, (\rho, \pi)) \rightharpoonup_s (\mathbf{c}_0, (\rho, \pi))}$$

$$\textsc{s-if-f}\ \frac{\rho \vdash \mathbf{b} \Downarrow \beta \qquad \pi' \triangleq \pi \wedge \neg\beta \qquad \mathbf{may}(\pi')}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1, (\rho, \pi)) \rightharpoonup_s (\mathbf{c}_1, (\rho, \pi))}$$

$$\textsc{s-loop-t}\ \frac{\rho \vdash \mathbf{b} \Downarrow \beta \qquad \pi' \triangleq \pi \wedge \beta \qquad \mathbf{may}(\pi')}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}, (\rho, \pi)) \rightharpoonup_s (\mathbf{c}; \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}, (\rho, \pi))}$$

$$\textsc{s-loop-f}\ \frac{\rho \vdash \mathbf{b} \Downarrow \beta \qquad \pi' \triangleq \pi \wedge \neg\beta \qquad \mathbf{may}(\pi')}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}, (\rho, \pi)) \rightharpoonup_s (\mathtt{skip}, (\rho, \pi))}$$

Figure 3.3: Symbolic execution step relation

Symbolic execution step Semantic rules are introduced in Figure 3.3. Rules such as S-ASSIGN, S-SEQ and S-SEQ-EXIT are trivial.

S-IF-T and S-IF-F are applied when `if` statement is met. These rules have to first evaluate the condition, then add it to the symbolic path, and check if the path is satisfiable. In the case of S-IF-F, the guard is negated. If the guard (does not) holds, command $\mathbf{c}_0$ ($\mathbf{c}_1$) is chosen to continue the execution. Notice that, both rules might apply to the same guard, contrary to a concrete execution. For instance, in Program 1.2a, guard $\mathtt{priv} > 0$ and its negation may hold, assuming the value of `priv` is unknown.

S-LOOP-T assumes the guard of the loop to hold. The next command to execute is a sequence of $\mathbf{c}$ and the loop. To exit a loop, S-LOOP-F evaluates the guard to false. This removes the loop for the next step. Notice that the given semantics in Figure 3.3 does not over-approximate the

```
1  # ℍ → priv ; 𝕃 → i
2  while i > priv:
3    i += 1
4    priv += 2
```

Figure 3.4: Unsafe program.

loop, and might never terminate. For example, in Program 1.2b, since the value of `z` is unkown, the semantics will unroll the loop infinitely.

2.4 Soundness

The standard symbolic execution semantics are sound, since every possible concrete execution is covered.

Theorem 1 (Soundness of a single symbolic execution step)**.** *Let* $(\boldsymbol{c}, \mu)$ *and* $(\boldsymbol{c}', \mu') \in \mathbb{S}$ *be two states such that* $(\boldsymbol{c}, \mu) \to (\boldsymbol{c}', \mu')$*,* $\kappa \in \mathbb{K}$ *a symbolic precise store, and* ν *be a valuation such that* $(\mu, \nu) \in \gamma_{\mathbb{K}}(\kappa)$*. Then, there exists a symbolic precise store* κ' *such that* $(\mu', \nu) \in \gamma_{\mathbb{K}}(\kappa')$ *and* $(\boldsymbol{c}, \kappa) \rightharpoonup_s (\boldsymbol{c}', \kappa')$*.*

Proof of Theorem 1 is done via structural induction on the syntax of commands, and it is assumed since this semantics are not new.

2.5 Completeness

A very desirable feature of symbolic execution is the ability to produce counter-examples. Since no over-approximation is done, the standard symbolic execution semantics are complete, and can generate counter-example.

Theorem 2 (Completeness)**.** *Let* $\boldsymbol{c}$ *be a command,* $\kappa, \kappa' \in \mathbb{K}$ *be two precise stores,* $w, w' \in \mathbb{W}$*, such that* $(\boldsymbol{c}, \kappa) \rightharpoonup_s^* (\texttt{skip}, \kappa')$*. Then, for all* $(\mu', \nu') \in \gamma_{\mathbb{K}}(\kappa')$*, it exists* $(\mu, \nu) \in \gamma_{\mathbb{K}}(\kappa)$ *such that* $(\boldsymbol{c}, \mu) \to^* (\texttt{skip}, \mu')$*.*

Proof of Theorem 2 is done via structural induction on the syntax of commands, and it is assumed since this semantics are not new.

Example. *In Program 3.4, the value of* `i` *depends on the value of* `priv`*. Since this symbolic execution is not relational, it is not possible to perform a noninterference analysis. Yet, we can use the SMT solver to generate relevant (as in real) traces. By executing the loop at least one time and exiting, a possible symbolic precise store is*

$$\rho = [\texttt{i} \mapsto (\boldsymbol{i} + 1); \texttt{priv} \mapsto (\boldsymbol{priv} + 2)] \qquad \pi \triangleq \boldsymbol{priv} + 2 \geq \boldsymbol{i} + 1$$

The SMT solver can be used to generate a store, for example

$$\mu = [\texttt{i} \mapsto 11; \texttt{priv} \mapsto 11] \qquad \nu = [\boldsymbol{i} \mapsto 10; \boldsymbol{priv} \mapsto 9]$$

Notice that, store μ is the final store, while the valuation ν holds the initial values of `i` *and* `priv`*.*

2.6 Limitations

Clearly, the exhaustive application of the symbolic execution step relation defined in Figure 3.3 would never terminate loops of unbounded length, which is not a desirable feature. Therefore, a common tactic in symbolic execution is to abort the exploration when the traces reach certain execution length. In turn, this makes the semantics unsound, as there will be valid traces that are ignored based on their length.

In the next section, we aim to mend this limitations by introducing an over-approximation mechanism.

3 SoundSE: Sound depth bounded symbolic execution

Our goal is to be able to do verification of properties with symbolic execution. For that reason, we propose to modify the standard symbolic execution semantics by implementing an over-approximation mechanism, specifically for loop statements. The over-approximation we will propose in this section will generate imprecise results, and, in essence, it will consist on clearing the value of variables that *might* have been modified.

3.1 Sound symbolic states

In order to define the over-approximation of loops it is necessary to extend symbolic states. The first addition is a precision flag, useful to keep track of whether there has been an over-approximation. Secondly, to keep track of how many times a loop has been visited, we add a counter and a function that determines when to perform the over-approximation.

Precision flag The *precision flag* states whether the symbolic execution has performed any over-approximation due to exhausting the bound of loop unrollings. Even if the analyzer is not fully complete, some traces have not been over-approximated. Thanks to this flag, non-over-approximated traces can be passed to the SMT solver to generate input values.

Counter and $\mathfrak{step}$ function The *counter* is used to keep track of the loop iterations. We define set $\mathbb{W}$ as the set of counters, with a special element $w_0 \in \mathbb{W}$ that denotes the initial counter status with respect to bound control. To operate over counters, we require a function $\mathfrak{step}$ which inputs two commands $\mathbf{c}$, $\mathbf{c}'$, and a counter w. It produces a result of the form (b, w') where b is a boolean, and w' is the next counter. Value b is $\mathbf{tt}$ if and only if a step from $\mathbf{c}$ to $\mathbf{c}'$ can be done without exhausting the iteration bounds, and with the new counter w'. If b is $\mathbf{ff}$, the iteration bound has been reached and the state needs to be over-approximated.

We give the definition of what we call the "classic counter" that will be used throughout the manuscript.

Definition 3 (Classic counter)**.** *The most typical way to bound symbolic execution limits the number of iteration of each loop to pre-defined number k. Then, $\mathbb{W}$ consists of stacks of integers, w_0 is the empty stack, and $\mathfrak{step}$: adds a one on top of the stack when entering a new loop, and pops the value on top of the stack when exiting a loop. More importantly, $\mathfrak{step}$ increments the value n at the top of the stack when $n \leq k$ and moving to the next iteration (rule S-LOOP-T); on the other hand, when $n > k$, it pops n and returns the $\mathbf{ff}$ precision flag.*

To ensure termination, $\mathbb{W}$ and $\mathfrak{step}$ should satisfy the following *well-foundedness* property: for any infinite sequence of commands $(\mathbf{c}_i)_i$ the infinite sequence $(w_i)_i$ defined by $\mathfrak{step}(\mathbf{c}_i, \mathbf{c}_{i+1}, w_i) = (\mathbf{tt}, w_{i+1})$ should be stationary, which we assume here.

3.2 Over-approximation of loops

At this point, the counter is set up, and we can determine when we want to perform an over-approximation. For this goal, our approach is to define a function called $\mathfrak{modif}$ that inputs a symbolic precise store, and a loop, and returns an over-approximation of the store.

Since we want to only over-approximate loops, we need an operation that splits a sequence into its head and its tail. Assuming $\mathbf{s}$ is a language statement, and $\mathbf{c}$ is a language command, we call this operation $\mathfrak{deseq}$, and it is defined as follows.

$$\begin{array}{lcl} \mathfrak{deseq}(\mathbf{s}) & = & (\mathbf{s}, \texttt{skip}) \\ \mathfrak{deseq}(\mathbf{s}; \mathbf{c}) & = & (\mathbf{s}, \mathbf{c}) \end{array}$$

Then, the modification function $\mathfrak{modif}$ should behave in the following way.

Definition 4 (Modification function is sound)**.** *Let $\boldsymbol{c} = (\texttt{while } \boldsymbol{b} \texttt{ do } \boldsymbol{c}_0); \boldsymbol{c}_1$ be a sequence starting with a* `while` *statement. Let μ a store, ν a valuation, and κ a symbolic precise store, such that $(\mu, \nu) \in \gamma_{\mathbb{K}}(\kappa)$. Then, $\mathfrak{modif}$ is such that*

$$(\boldsymbol{c}, \mu) \xrightarrow{*} (\boldsymbol{c}_1, \mu') \implies \mathfrak{modif}(\kappa, \boldsymbol{c}) = (\kappa', \boldsymbol{c}_1) \wedge \exists \nu' : \nu \preceq \nu' \wedge (\mu', \nu') \in \gamma_{\mathbb{K}}(\kappa')$$

We define a possible implementation of $\mathfrak{modif}$.

Definition 5 (Simple $\mathfrak{modif}$)**.** *Given a precise store κ, and a command $\boldsymbol{s} =$* `while` $\boldsymbol{b}$ `do` $\boldsymbol{c}_0$ *where $\boldsymbol{c}_1$ can be any command, or just* `skip`*. Then, we define $\mathfrak{modif} = (\rho, \pi)$ as follows:*

1. *Calculate the set $\mathfrak{m}$ of modifiable variables by doing a syntactic check over $\boldsymbol{c}_0$: any variable that appears in the left-hand side of an assignment gets added to $\mathfrak{m}$.*
2. *We define a new symbolic store ρ' such that*

$$\rho'(\mathtt{x}) = \begin{cases} \boldsymbol{x} & \text{if } \mathtt{x} \in \mathfrak{m} \text{ with fresh symbol } \boldsymbol{x} \\ \rho(\mathtt{x}) & \text{otherwise} \end{cases}$$

We call symbolic variables fresh *if it has not appeared previously in the symbolic precise store.*

3. Evaluate the guard $\boldsymbol{b}$*, and add its negation to the symbolic path.*

$$\pi' = \pi \wedge \neg\beta \text{ where } (\boldsymbol{b}, \rho') \vdash_s (\beta)$$

4. Return (ρ', π')*.*

Lemma 1. *Simple* $\mathfrak{modif}$ *is sound.*

Proof of Lemma 1 is done via induction ?

3.3 Sound symbolic semantics

$$\text{S-NEXT}\ \frac{(\mathbf{c}, \kappa) \rightharpoonup_s (\mathbf{c}', \kappa') \qquad \mathfrak{step}(\mathbf{c}, \mathbf{c}', w) = (\mathbf{tt}, w')}{(\mathbf{c}, \kappa, w, b) \rightharpoonup_s (\mathbf{c}', \kappa', w', b)}$$

$$\text{S-APPROX-MANY}\ \frac{\begin{array}{c}(\mathbf{c}, \kappa) \rightharpoonup_s (\mathbf{c}', \kappa') \qquad \mathfrak{step}(\mathbf{c}, \mathbf{c}', w) = (\mathbf{ff}, w') \\ \mathfrak{deseq}(\mathbf{c}) = (\mathbf{c}_0, \mathbf{c}_1) \qquad \mathfrak{modif}(\kappa, \mathbf{c}_0) = \kappa''\end{array}}{(\mathbf{c}, \kappa, w, b) \rightharpoonup_s (\mathbf{c}_1, \kappa'', w', \mathbf{ff})}$$

Figure 3.5: SoundSE: Sound bounded symbolic execution step relation

Based on these definitions, *depth bounded symbolic execution* is defined by a transition relation over 4-tuples made of a command, a symbolic state, an element of $\mathbb{W}$, and a boolean, referred to as symbolic state. We overload the notation $\rightharpoonup_s$ for this relation, which is defined based on the previously defined $\rightharpoonup_s$. The rules are provided in Figure 3.5:

- Rule S-NEXT carries out an atomic step of symbolic execution that requires no over approximation; function $\mathfrak{step}$ returns the precision flag b and a new counter;
- Rule S-APPROX-MANY carries out a global approximation step; indeed, as $\mathfrak{step}$ returns $\mathbf{ff}$, the function $\mathfrak{modif}$ is applied to the symbolic state to over-approximate the effect of an arbitrary number of steps of execution of $\mathbf{c}$; alongside with the new counter state the $\mathbf{ff}$ precision is propagated forward.

Under the well-foundedness assumption, exhaustive iteration of the available symbolic execution rules from any initial symbolic state will terminate and produce finitely many symbolic states.

Example 3. *We assume the bounding of Definition 3 with its limit of iterations* $k = 5$*, and the* $\mathfrak{modif}$ *as in Definition 5. In Program 1.2b, after executing the loop 5 times, the symbolic precise store is*

$$\rho = [\mathtt{i} \mapsto (5); \mathtt{z} \mapsto (\boldsymbol{z}); \mathtt{priv} \mapsto (\boldsymbol{priv} + 10)] \qquad \pi \triangleq 5 < \boldsymbol{z}$$

At the next attempt to execute the loop, function $\mathfrak{step}$ *will return* $\boldsymbol{ff}$*. Then, function* $\mathfrak{modif}$ *is called, assigning fresh values for variables* `i` *and* `priv` *since they might have been modified. The resulting precise store is*

$$\rho = [\mathtt{i} \mapsto (\boldsymbol{i}_1); \mathtt{z} \mapsto (\boldsymbol{z}); \mathtt{priv} \mapsto (\boldsymbol{priv}_1)] \qquad \pi \triangleq \boldsymbol{i}_1 = \boldsymbol{z}$$

3.4 Soundness

To express the soundness of this algorithm, we need to account for the creation of symbolic values by function $\mathfrak{modif}$, which means that valuations also need to be extended. To this end, we note $\nu \preceq \nu'$ when the domain of valuation ν is included into that of ν' and when both ν and ν' agree on the intersection of their domains.

We now obtain the following soundness statement:

Theorem 3 (Soundness of any sequence of SoundSE steps)**.** *Let $(\boldsymbol{c}, \mu) \in \mathbb{S}$ be a state and μ' be a store such that $(\boldsymbol{c}, \mu) \to^* (\texttt{skip}, \mu')$. Let $\kappa \in \mathbb{K}$ be a symbolic precise store and ν be a valuation such that $(\mu, \nu) \in \gamma_{\mathbb{K}}(\kappa)$. Let $w \in \mathbb{W}$ be a counter. Then, there exists a symbolic precise store κ', a valuation ν', and a counter $w' \in \mathbb{W}$ such that $\nu \preceq \nu'$, $(\mu', \nu') \in \gamma_{\mathbb{K}}(\kappa')$, and $(\boldsymbol{c}, \kappa, w, b) \rightharpoonup_s^* (\texttt{skip}, \kappa', w', b')$.*

Proof. There is an execution $(\mathbf{c}, \mu) \xrightarrow{n} (\texttt{skip}, \mu)$ where $\mathbf{c}$ terminates in n steps. Let κ be such that $(\mu, \nu) \in \gamma_{\mathbb{K}}(\kappa)$ By doing induction over n, steps will either be simulated one-to-one by S-NEXT, or a sequence of steps will be simulated by S-APPROX-MANY. When rule S-APPROX-MANY, we use the hypothesis that $\mathfrak{modif}$ is providing a sound over-approximation of the loop. Eventually, by applying this rules exhaustively, the execution of $\mathbf{c}$ terminates, and we have $(\mu', \nu') \in \gamma_{\mathbb{K}}(\kappa')$ and $\nu \preceq \nu'$. Thus, the SoundSE semantics are sound. □

Example 4 (Symbolic execution)**.** *For Program 1.2a, symbolic execution performs no over-approximation, resulting in precise stores with precision flag **tt**.*

*For Program 1.2b, as in Example 3, the over-approximation of the loop is performed, turning the precision flag to **ff**, with* `i` *and* `priv` *being assigned fresh symbolic values generated by rule* S-APPROX-MANY*. There are no constraints over the fresh symbols, except the negation of the guard which involves* `i`*. Thus, the exact value of* `i` *cannot be calculated, as the precision flag implies.*

3.5 Refutation

Theorem 4 (Refutation up to a bound of SoundSE)**.** *Let $\boldsymbol{c}$ be a command, $\kappa, \kappa' \in \mathbb{K}$ be two precise stores, $w, w' \in \mathbb{W}$, such that $(\boldsymbol{c}, \kappa, w, \boldsymbol{tt}) \rightharpoonup_s^* (\texttt{skip}, \kappa', w', \boldsymbol{tt})$. Then, for all $(\mu', \nu') \in \gamma_{\mathbb{K}}(\kappa')$, it exists $(\mu, \nu) \in \gamma_{\mathbb{K}}(\kappa)$ such that $(\boldsymbol{c}, \mu) \to^* (\texttt{skip}, \mu')$.*

Proof. The SoundSE semantics has a boolean flag b that starts as true (**tt**). Applications of rule S-NEXT do not alter the precision flag. Therefore, the only way to have a false (**ff**) precision flag is to apply rule S-APPROX-MANY. Indeed, this rule over-approximates states, and loses completeness of the analysis. Thus, Theorem 2 still applies, and this theorem holds. □

Example 5 (Symbolic execution completeness up to a bound)**.** *We consider the cases discussed in Example 4. Using the bounding of Definition 3, the result produced for program 1.2a is complete whereas that for Program 1.2b generates some final symbolic state with precision flag **ff**, hence for which Theorem 4 does not apply on these traces. For traces that exited the loop before the bound k, the SMT solver can still generate counter-examples.*

Chapter 4

RedSoundSE: product of SoundSE and abstractions

To enhance the precision of loop over-approximation, this chapter introduces a modified rule that utilises both symbolic execution and abstract interpretation. The notion that combines these two methods is called *reduced product*, which involves defining new states that incorporate both symbolic and abstract stores, and defining a new semantics that executes both symbolically and abstractly. Since the symbolic execution semantics is small-step, this new reduced product semantics must also follow that principle, contrary to standard abstract interpretation analyses. However, we still want to define a classical abstract interpretation semantics that allow us to soundly over-approximate loops. Ultimately, this semantics will rely on a function called *reduction function* that refines the representation of such states by "sharing" information between the symbolic and abstract stores. Hence, after over-approximating a loop, the reduction function will alter the symbolic store raising the general precision of the analysis. This new reduced product semantics will be called RedSoundSE.

1 Abstract interpretation

Abstract interpretation semantics are usually sound, producing a single abstraction that encompasses the whole semantics of a program. This is of great interest to us, as the over-approximation presented in the previous chapter is extremely imprecise. Still, we will also require a small-step abstract semantics that follows closely that of the symbolic execution. To do so, we will first define abstract domains—which are abstractions of concrete values—two abstract transport functions—this functions allow us to operate over the abstractions—and two abstract semantics: a classical abstract interpretation semantics, and a small-step abstract semantics.

1.1 Abstract domain

An *abstract domain* [Cousot, 1977] is an abstraction of concrete values. Abstract domains come in many shapes, such as *signs domain* where numerical values are abstracted to their sign (negative

or positive), *parity domain* where values are abstracted to either even or odd, and so on. Domains can be defined as the need arises, depending on the type of property that is being studied. A key aspect of domains is that by retaining a limited amount of information, it is still possible to infer interesting information about a program execution.

For our purpose, we assume a general abstract domain $\mathbb{A}$ describing sets of stores, together with a *concretization function* $\gamma_{\mathbb{A}}$. We call elements of $\mathbb{A}$ *abstractions* or *abstract stores*. A concretization function, takes an abstraction $a \in \mathbb{A}$ and returns a set of concrete stores that are being abstracted by a.

$$\gamma_{\mathbb{A}} : \mathbb{A} \longrightarrow \mathcal{P}(\mathbb{M})$$

Abstract domains are complete lattices $(\mathbb{A}, \sqsubseteq)$, and as such, we assume the existence of a top and bottom elements, where $\gamma_{\mathbb{A}}(\top) = \mathbb{M}$ and $\gamma_{\mathbb{A}}(\bot) = \emptyset$. Naturally, $\forall a : \bot \sqsubseteq a \sqsubseteq \top$. The lattice operation lub (least upper bound), often called *join* in the context, and denoted by $\sqcup$, returns an abstraction that encompasses both abstractions. That is

$$\gamma_{\mathbb{A}}(a_0) \subseteq \gamma_{\mathbb{A}}(a_0 \sqcup a_1) \ \wedge \ \gamma_{\mathbb{A}}(a_1) \subseteq \gamma_{\mathbb{A}}(a_0 \sqcup a_1)$$

Example (Intervals and convex polyhedra abstract domains)**.** *In the intervals domain [Cousot, 1977], the abstract elements are defined by constraints of the form* $l_{\mathtt{x}} \leq \mathtt{x}$, $\mathtt{x} \leq h_{\mathtt{x}}$ *where* $\mathtt{x}$ *is a variable,* $l_{\mathtt{x}}$ *is the lower bound of* $\mathtt{x}$*, and* $h_{\mathtt{x}}$ *is the higher bound of* $\mathtt{x}$*. For instance, an intervals abstraction might describe the possible values of* $\mathtt{x}$ *as* $0 \leq \mathtt{x}10$*, implying that any memory where the value of* $\mathtt{x}$ *is between 0 and 10 is in its concretization. Since intervals cannot relate variables, that is, it is not possible to write* $\mathtt{x} \leq \mathtt{y}$*, we consider it a* non-relational domain*.*

In the convex polyhedra abstract domain [Cousot, 1978], the abstract elements are conjunctions of linear inequality constraints. These constraints can describe the relationship between different variables. For example, an `if` *statement with guard* $\mathtt{x} < \mathtt{y} * 2$ *that constraints the upper bound of* $\mathtt{x}$ *based on that of* $\mathtt{y}$*. This inequality can be expressed by convex polyhedra. We say that this domain is a* relational domain*. We say that the convex polyhedra domain is a* relational *domain. This relational constraint cannot be expressed by the intervals domain. Instead, an over-approximation of the constraint is used.*

1.2 Abstract transport functions

To define a small-step abstract semantics, we require two *abstract transport functions* that allow us to modify the abstractions. We will overload the function $[\![\cdot]\!]$ once again, to replace variables in an expression by their mapping in a store.

Definition (Abstract assignment)**.** $\mathfrak{assign}_{\mathtt{x},e} : \mathbb{A} \longrightarrow \mathbb{A}$ *is parameterized by a variable* $\mathtt{x}$ *and an expression* $\boldsymbol{e}$ *and is such that*

$$\forall a \in \mathbb{A}, \ \{\mu[\mathtt{x} \mapsto [\![\boldsymbol{e}]\!](\mu)] \mid \mu \in \gamma_{\mathbb{A}}(a)\} \ \subseteq \ \gamma_{\mathbb{A}}(\mathfrak{assign}_{\mathtt{x},\boldsymbol{e}}(a))$$

Function $\mathfrak{assign}_{\mathtt{x},\mathtt{e}}$ allows to modify an abstraction by simulating the assignment of $\mathbf{e}$ to $\mathtt{x}$. For example, in the context of intervals, let $a = \{0 \leq \mathtt{y} \leq 10\}$. Then, the assignment $\mathtt{x} = \mathtt{y}$

causes $\mathtt{x}$ to take be constrained by the same interval $\mathtt{assign}_{\mathtt{x},\mathtt{y}}(a) = \{0 \leq \mathtt{y} \leq 10\ ;\ 0 \leq \mathtt{x} \leq 10\}$.

Definition (Abstract condition)**.** $\mathtt{guard}_{\boldsymbol{b}} : \mathbb{A} \longrightarrow \mathbb{A}$ *is parameterized by a boolean expression* $\boldsymbol{b}$ *and is such that*

$$\forall a \in \mathbb{A},\ \{\mu \in \gamma_{\mathbb{A}}(a) \mid [\![\boldsymbol{b}]\!](\mu) = \boldsymbol{tt}\} \ \subseteq\ \gamma_{\mathbb{A}}(\mathtt{guard}_{\boldsymbol{b}}(a))$$

Function $\mathtt{guard}$ constraints the abstract store. For example, given a guard $5 \leq \mathtt{x}$ with the abstract store $a = \{0 \leq \mathtt{x} \leq 10\}$ tightens the interval of $\mathtt{x}$ to $\mathtt{guard}_{\mathtt{b}}(a) = \{5 \leq \mathtt{x} \leq 10\}$.

1.3 Abstract interpretation based static analysis

In order to over-approximate loops precisely, we need to define an *abstract interpretation based analysis* that always terminates and is sound.

In the abstract semantics, $\mathbb{A}$ is assumed to be a parameter of the analysis. It may consist of any numerical abstraction, such as the interval abstract domain or the domain of convex polyhedra.

Definition 6 (Sound abstract semantics)**.** *The static analysis function* $[\![\boldsymbol{c}]\!]^\sharp_{\mathbb{A}} : \mathbb{A} \longrightarrow \mathbb{A}$ *is sound in the sense that, for all command* $\boldsymbol{c}$ *and all abstract state* a*,*

$$\{\mu' \in \mathbb{M} \mid \exists \mu \in \gamma_{\mathbb{A}}(a),\ (\boldsymbol{c}, \mu) \xrightarrow{*} (\mathtt{skip}, \mu')\} \subseteq \gamma_{\mathbb{A}}([\![\boldsymbol{c}]\!]^\sharp_{\mathbb{A}}(a))$$

meaning that $\gamma_{\mathbb{A}}([\![\boldsymbol{c}]\!]^\sharp_{\mathbb{A}}(a))$ *is a sound over-approximation of the concrete semantics.*

To ensure termination, we have function $\mathsf{lfp}^{\sqsubseteq}_a : \mathbf{C} \longrightarrow \mathbb{A}$ parameterized by an abstraction a that calculates the least fixed point of a command starting by a. This function will apply $[\![a]\!]^\sharp_{\mathbb{A}}$ exhaustively until it reaches a fixed point, assured by the application of a *widening operator* [Cousot, 1977]. We characterize the widening operator as follows.

Definition 7 (Widening operator)**.** *A* widening operator *over an abstract domain* $\mathbb{A}$ *is a binary operator* $\triangledown$*, such that*

(i) for all abstract elements $a_0, a_1 \in \mathbb{A}$*,*

$$\gamma_{\mathbb{A}}(a_0) \cup \gamma_{\mathbb{A}}(a_1) \subseteq \gamma_{\mathbb{A}}(a_0 \triangledown a_1)$$

(ii) ll sequences $(a_n)_{n\in\mathbb{N}}$ *of abstract elements, the sequence* $(a'_n)_{n\in\mathbb{N}}$ *defined below converges:*

$$\begin{cases} a'_0 &= a_0 \\ a'_{n+1} &= a'_n \triangledown a_n \end{cases}$$

Then, $\mathsf{lfp}^{\sqsubseteq}_a$ will apply the condition and body of the loop over a, widening at each iteration, until a fixed point is reached. A definition of the semantics can be found in Figure 4.1.

Example 6 (Convex polyhedra abstraction)**.** *For Program 1.2a, using convex polyhedra, the abstract interpretation will calculate the two branches of the* `if` *independently, and then use the*

$$[\![\texttt{skip}]\!]^\sharp_{\mathbb{A}}(a) = a$$

$$[\![\mathtt{x} = \mathbf{e}]\!]^\sharp_{\mathbb{A}}(a) = \mathfrak{assign}_{\mathtt{x},\mathbf{e}}(a) \qquad [\![\mathbf{c}_0; \mathbf{c}_0]\!]^\sharp_{\mathbb{A}}(a) = [\![\mathbf{c}_1]\!]^\sharp_{\mathbb{A}} \circ [\![\mathbf{c}_0]\!]^\sharp_{\mathbb{A}}(a)$$

$$[\![\texttt{if } \mathbf{b} \texttt{ then } \mathbf{c}_0 \texttt{ else } \mathbf{c}_1]\!]^\sharp_{\mathbb{A}} = \left([\![\mathbf{c}_0]\!]^\sharp_{\mathbb{A}} \circ \mathfrak{guard}_{\mathbf{b}}(a)\right) \sqcup \left([\![\mathbf{c}_1]\!]^\sharp_{\mathbb{A}} \circ \mathfrak{guard}_{\neg\mathbf{b}}(a)\right)$$

$$[\![\texttt{while } \mathbf{b} \texttt{ do } \mathbf{c}_0]\!]^\sharp_{\mathbb{A}} = \mathfrak{guard}_{\neg\mathbf{b}} \circ \mathsf{lfp}^{\sqsubseteq}_{a} [\![\texttt{if } \mathbf{b} \texttt{ then } \mathbf{c}_0 \texttt{ else skip}]\!]^\sharp_{\mathbb{A}}$$

Figure 4.1: Abstract interpretation semantics

$$\text{A-ASSIGN}\ \frac{a' \triangleq \mathfrak{assign}_{\mathtt{x},\mathbf{e}}(a)}{(\mathtt{x} := \mathbf{e}, a) \rightharpoonup_{\mathbb{A}} (\texttt{skip}, a')} \qquad \text{A-IF-T}\ \frac{a' \triangleq \mathfrak{guard}_{\mathbf{b}}(a) \qquad a' \neq \bot}{(\texttt{if } \mathbf{b} \texttt{ then } \mathbf{c}_0 \texttt{ else } \mathbf{c}_1, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}_0, a')}$$

$$\text{A-IF-F}\ \frac{a' \triangleq \mathfrak{guard}_{\neg\mathbf{b}}(a) \qquad a' \neq \bot}{(\texttt{if } \mathbf{b} \texttt{ then } \mathbf{c}_0 \texttt{ else } \mathbf{c}_1, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}_0, a')}$$

$$\text{A-SEQ-EXIT}\ \frac{}{(\texttt{skip}; \mathbf{c}_1, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}_1, a)} \qquad \text{A-SEQ}\ \frac{(\mathbf{c}_0, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}'_0, a')}{(\mathbf{c}_0; \mathbf{c}_1, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}'_0; \mathbf{c}_1, a')}$$

$$\text{A-LOOP-F}\ \frac{a' \triangleq \mathfrak{guard}_{\neg\mathbf{b}}(a) \qquad a' \neq \bot}{(\texttt{while } \mathbf{b} \texttt{ do } \mathbf{c}_0, a) \rightharpoonup_{\mathbb{A}} (\texttt{skip}, a')}$$

$$\text{A-LOOP-T}\ \frac{a' \triangleq \mathfrak{guard}_{\mathbf{b}}(a) \qquad a' \neq \bot}{(\texttt{while } \mathbf{b} \texttt{ do } \mathbf{c}_0, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}; \texttt{while } \mathbf{b} \texttt{ do } \mathbf{c}_0, a')}$$

Figure 4.2: Abstract execution step

join operation.

$$\begin{array}{rcl} [\![\mathbf{c}_0]\!]^\sharp_{\mathbb{A}} \circ \mathfrak{guard}_{\mathbf{b}}(a) & = & \{1 \leq \mathtt{priv}\ ;\ \mathtt{y} = 5\} \\ [\![\mathbf{c}_1]\!]^\sharp_{\mathbb{A}} \circ \mathfrak{guard}_{\neg\mathbf{b}}(a) & = & \{\mathtt{priv} \leq 0\ ;\ \mathtt{y} = 5\} \end{array} \overset{\sqcup}{\Longrightarrow} \{\mathtt{y} = 5\}$$

The join will cause the constraint over `priv` *to be lost, but the equality on* `y` *is kept.*

For Program 1.2b, the loop could potentially execute infinitely. Then, the least fixed point function will execute the loop and apply the widening operator until a fixed point is found.

$$\begin{array}{rcl} a'_1 & = & \{0 \leq \mathtt{i} \leq 0\} \\ a'_2 & = & \{0 \leq \mathtt{i} \leq 1\} \\ a'_3 & = & \{0 \leq \mathtt{i} \leq \infty\} \end{array}$$

At a'_3 *the fixed point is found, and after negating the guard the abstraction is*

$$a = \{\mathtt{z} \leq \mathtt{i}\ ;\ 0 \leq \mathtt{i} \leq \infty\}$$

Since the value of `z` *could potentially be negative, either* `i` *stayed with value 0, or it grew.*

1.4 Abstract step semantics

The abstract interpretation based analysis fits perfectly to perform the over-approximation of loops as we desire. However, the symbolic execution semantics work in a small-step fashion. Thus, we define a *small-step abstract semantics* that imitates the symbolic execution semantics so that these "synchronize" in the execution of a statement. We provide its definition in Figure 4.2 by using the abstract operations defined earlier, making it very straightforward.

These abstract semantics cannot ensure termination, similarly to the standard symbolic execution. Also, each abstraction follows a single path, which contrasts with classical abstract interpretation were operations such as join and widening are used to ensure the coverage of all possibilities.

Example 7. *For Program 1.2a, since there is no join, collecting all possible executions results in two abstractions.*

$$\begin{aligned} a_{tt} &= \{1 \leq \texttt{priv};\ \texttt{y} = 5\} \\ a_{ff} &= \{\texttt{priv} \leq 0;\ \texttt{y} = 5\} \end{aligned}$$

For Program 1.2b, an infinite amount of abstractions are generated. We list some of them.

$$\begin{aligned} a_0 &= \{\texttt{z} \leq 0\ ;\ \texttt{i} = 0\} \\ a_1 &= \{\texttt{z} = 1\ ;\ \texttt{i} = 1\} \\ a_2 &= \{\texttt{z} = 2\ ;\ \texttt{i} = 2\} \\ &\vdots \end{aligned}$$

2 Reduction of symbolic precise stores and abstract states

Reduced product [Cousot, 1979] aims at expressing precisely conjunctions of constraints expressed in distinct abstract domains. First, a *product domain* is defined. In our case, the product domain will hold a precise symbolic store, and an abstraction. Then, the reduction function will exchange information between these two, to enhance the general precision of the analysis.

2.1 Product domain

We let a *precise product store* be a pair $(\kappa, a) \in \mathbb{K} \times \mathbb{A}$. In our case, the definition needs to be adapted slightly as symbolic execution and abstract domain $\mathbb{A}$ do not abstract exactly the same objects:

Definition 8 (Product domain)**.** *The* product abstract domain *consists of the set* $\mathbb{K} \times \mathbb{A}$ *and the concretization function* $\gamma_{\mathbb{K}\times\mathbb{A}}$ *defined as follows:*

$$\begin{array}{rccl} \gamma_{\mathbb{K}\times\mathbb{A}} : & \mathbb{K} \times \mathbb{A} & \longrightarrow & \mathcal{P}(\mathbb{M} \times (\overline{\mathbb{V}} \to \mathbb{V})) \\ & (\kappa, a) & \longmapsto & \{(\mu, \nu) \in \gamma_{\overline{\mathbb{M}}}(\kappa) \mid \mu \in \gamma_{\mathbb{A}}(a)\} \end{array}$$

2.2 Reduction

In a precise product store (κ, a), the goal is to enhance precision by exchanging information between κ and a. This is done through a *reduction* function, which rewrites an abstract element with another of equal concretization, but that supports more precise analysis operations.

Then the reduction follow must satisfy the following

Definition 9 (Reduction is sound)**.** *Let* $(\kappa, a) \in \mathbb{K} \times \mathbb{A}$*. Then,*

$$(\gamma_{\mathbb{K}\times\mathbb{A}} \circ \mathfrak{reduction})(\kappa, a) = \gamma_{\mathbb{K}\times\mathbb{A}}(\kappa, a)$$

Our approach is based on the assumption that the abstract domain $\mathbb{A}$ supports a function called $\mathfrak{constr}$, that maps an abstract state a to a logical formula over program variables such that, if $\mu \in \gamma_{\mathbb{A}}(a)$ then μ satisfies formula $\mathfrak{constr}(a)$. Some abstract domains—specifically intervals and convex polyhedra—utilize an internal representation based on conjunction of constraints, in which case $\mathfrak{constr}$ is trivial.

We provide a possible definition for the reduction function.

Definition 10 (Simple reduction)**.**

$$\begin{array}{rccl} \mathfrak{reduction}: & \mathbb{K}\times\mathbb{A} & \longrightarrow & \mathbb{K}\times\mathbb{A} \\ & ((\rho,\pi),a) & \longmapsto & ((\rho,\pi'),a) \quad \textit{where} \quad \pi' = \pi \wedge \mathfrak{constr}(a)[\vec{\mathtt{x}} \mapsto \rho(\vec{\mathtt{x}})] \end{array}$$

Note that $[\vec{\mathtt{x}} \mapsto \rho(\vec{\mathtt{x}})]$ in the above definition, symbolizes the replacement of each program variable present in $\mathfrak{constr}(a)$ into its definition in ρ. This step follows from the fact that a constrains program variables whereas π constrains valuations. This general reduction function may be refined into a more precise one, where the resulting symbolic path is simplified, possibly to the **ff** formula. Furthermore, this reduction only modifies the symbolic path π, but it is possible to define a reduction operation that also rewrites the abstract state a.

Example 8. *Let* $(\kappa, a) = (([\mathtt{z} \mapsto \boldsymbol{z} * 2], (10 \leq \boldsymbol{z})), \{20 \leq \mathtt{z} \leq 40\})$*. In this precise product store,* a *is more precise than* κ *over* $\mathtt{z}$*. By applying the reduction function the symbolic path* π *is updated.*

$$\mathfrak{reduction}(\kappa, a) = (([\mathtt{z} \mapsto \boldsymbol{z} * 2], (10 \leq \boldsymbol{z} \leq 20)), \{20 \leq \mathtt{z} \leq 40\})$$

Notice that, while the abstraction is referring to variables of the program, the symbolic path refers to symbolic variables. Hence, the replacement of variables using the symbolic store must be performed in the reduction.

3 RedSoundSE: reduced product symbolic execution

Similarly to SoundSE, the new semantics, RedSoundSE, takes the form of an extension the standard symbolic execution presented in the previous chapter, meaning that we only need to define two new rules: one for simple execution steps, and one for the over-approximation of loops.

$$\textsc{s-a-next}\ \frac{(\mathbf{c},\kappa) \rightharpoonup_s (\mathbf{c}',\kappa') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w) = (\mathbf{tt},w') \qquad (\mathbf{c},a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}',a') \qquad \mathfrak{reduction}(\kappa',a') = (\kappa'',a'')}{(\mathbf{c},(\kappa,a),w,b) \rightharpoonup_{s\times\mathbb{A}} (\mathbf{c}',(\kappa'',a''),w',b)}$$

$$\textsc{s-a-approx-many}\ \frac{(\mathbf{c},\kappa) \rightharpoonup_s (\mathbf{c}',\kappa') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w) = (\mathbf{ff},w') \qquad \mathfrak{deseq}(\mathbf{c}) = (\mathbf{c}_0,\mathbf{c}_1) \qquad \mathfrak{modif}(\kappa,\mathbf{c}_0) = \kappa'' \qquad a' = [\![\mathbf{c}_0]\!]^\sharp_{\mathbb{A}}(a) \qquad \mathfrak{reduction}(\kappa'',a') = (\kappa''',a''')}{(\mathbf{c},(\kappa,a),w,b) \rightharpoonup_{s\times\mathbb{A}} (\mathbf{c}_1,(\kappa''',a'''),w',\mathbf{ff})}$$

Figure 4.3: Product of symbolic execution and abstract interpretation

3.1 Reduced product semantics

The new *product states* are 4-tuples as previously, with the difference that the symbolic precise store (previously κ) is now replaced by a precise product store: a pair (κ, a). The transition relation $\rightharpoonup_{s\times\mathbb{A}}$ between such states consists of two rules that are shown in Figure 4.3 and that extend those in Figure 3.5.

Rule S-A-NEXT corresponds to a regular symbolic execution step, and in this case, a similar abstract execution step is performed. For this rule, it is important that we use the small-step abstract semantics. After doing both one step in the symbolic and abstract stores, the reduction function is applied. This is optional depending on the reduction function: if the simple definition of reduction is used, using the reduction in this rule will not raise the precision of further execution steps.

When the exploration bound is met, rule S-A-APPROX-MANY is applied. As in rule S-APPROX-MANY, first $\mathfrak{modif}$ is used to clear modified variables and create a sound over-approximation of the store. Then, the big-step abstract semantics are used to over-approximate the loop, and the reduction function is used. In this case, the reduction function will share information between the symbolic and abstract stores, raising the precision of the following execution steps. In the case of the simple reduction function from Definition 10, this implies injecting constraints from the abstraction into the symbolic store.

Example 9 (Product analysis)**.** *In Program 1.2b, the value of* `z` *is unknown. Hence, the final value of* `i` *is unknown, but it can be narrowed down by using the convex polyhedra domain. When the loop is reached, to paths are taken: the first path does not enter the loop, assuming that the value of* `z` *is lesser-equal to 0. In the other path,* `z` *must be greater-equal to 1. After k steps, where k is the classic counter bound the precise product store is as follows.*

$$\kappa = ([\mathtt{i} \mapsto k\ ;\ \mathtt{z} \mapsto \boldsymbol{z}_0\ ;\ \mathtt{priv} \mapsto \boldsymbol{priv}_0 + k*2], k-1 < \boldsymbol{z}_0) \qquad a = \begin{cases} \mathtt{i}-1 < \mathtt{z} \\ k \leq \mathtt{i} \leq k \end{cases}$$

Then, S-A-APPROX-MANY *is applied. Before the reduction, the stores are as follows.*

$$\kappa = ([\mathtt{i} \mapsto \boldsymbol{i}_1\ ;\ \mathtt{z} \mapsto \boldsymbol{z}_0\ ;\ \mathtt{priv} \mapsto \boldsymbol{priv}_1], k-1 < \boldsymbol{z}_0\ \wedge\ \boldsymbol{i}_1 \geq \boldsymbol{z}_0) \qquad a = \begin{cases} \mathtt{i} = \mathtt{z} \\ k-1 \leq i \end{cases}$$

```
1  # 𝕙 → priv ; 𝕝 → y,i
2  if priv < 0: priv = 0
3  while i < 10:
4    i += 1
5    priv += 2
6  if priv >= 0: y += 1
7  else: y = 0
```

Figure 4.4: Safe program that requires either RedSoundSE with intervals or polyhedra, or RedSoundRSE with dependences.

Finally, after the reduction.

$$\kappa = ([\mathtt{i} \mapsto i_1 \ ; \ \mathtt{z} \mapsto z_0 \ ; \ \mathtt{priv} \mapsto priv_1], i_1 = z_0 \ \wedge \ k - 1 \leq i_1)$$

The new constraint in green, is clearly more precise than the old one.

The last example we present is program from Figure 4.4. This program is safe, but it requires either the use RedSoundSE with intervals or polyhedra.

Example 10. *In this example we will execute Program 4.4 with RedSoundSE. We denote the program with* $\boldsymbol{c}$.

We will start reasoning about the program by looking at the first line with the `if` *statement. By applying* S-IF-T*, the value of* `priv` *is 0. By applying* S-IF-F*, the value of* `priv` *must be greater or equal to 0. With this logic we can assume that* `priv` *is always greater or equal to 0 before the loop.*

The value of `i` *is unknown. Therefore, both* S-LOOP-T *and* S-LOOP-F *can be applied. Assuming the loop is never entered, the value of* `i` *is not modified, neither* `priv`*. If the loop is executed,* `i` *will grow until the guard does not hold. However, the value of* `i` *is unknown and, eventually, needs to be over-approximated.*

If we were to apply rule S-APPROX-MANY*, from SoundSE, the new value of* `i` *and* `priv` *would be completely imprecise. Instead, by using* S-A-APPROX-MANY *with a numerical domain such as intervals, it is possible to determine that the value of* `i` *is exactly 10, and the value of* `priv` *is greater or equal to 0.*

Summarizing, in both cases of executing or not executing the loop, it is possible to establish that the value of `priv` ≥ 0*. This, in turn, forces the last* `if` *statement to always execute, making the value of* `y` *equal to its original value plus 1, for every execution.*

What this implies is that, when we apply a relational symbolic execution using RedSoundSE as the basis, the analysis will determine that the value of `y` *is always shared between executions and that the program is noninterferent.*

3.2 Soundness and refutation property

The RedSoundSE analysis defined satisfies the same soundness (Theorem 3) and refutation (Theorem 4) properties as SoundSE.

Theorem 5 (Soundness of any sequence of RedSoundSE steps)**.** *Let* $(\boldsymbol{c}, \mu) \in \mathbb{S}$ *be a state and* μ' *be a store such that* $(\boldsymbol{c}, \mu) \to^* (\texttt{skip}, \mu')$*. Let* $(\kappa, a) \in \mathbb{K} \times \mathbb{A}$ *be a product precise store and* ν *be a valuation such that* $(\mu, \nu) \in \gamma_{\mathbb{K}\times\mathbb{A}}(\kappa, a)$*. Let* $w \in \mathbb{W}$ *be a counter. Then, there exists a symbolic precise store* κ'*, a valuation* ν'*, and a counter* $w' \in \mathbb{W}$ *such that* $\nu \preceq \nu'$*,* $(\mu', \nu') \in \gamma_{\mathbb{K}\times\mathbb{A}}(\kappa', a')$*, and* $(\boldsymbol{c}, (\kappa, a), w, b) \rightharpoonup_{s\times\mathbb{A}} (\texttt{skip}, (\kappa', a), w', b')$*.*

Proof. There is an execution $(\mathbf{c}, \mu) \xrightarrow{n} (\texttt{skip}, \mu)$ where $\mathbf{c}$ terminates in n steps. Let κ be such that $(\mu, \nu) \in \gamma_{\mathbb{K}}(\kappa)$ By doing induction over n, steps will either be simulated one-to-one by S-A-NEXT, or a sequence of steps will be simulated by S-A-APPROX-MANY. When rule S-A-APPROX-MANY, we use the hypothesis that $\mathfrak{modif}$ is providing a sound over-approximation of the loop, as well as the soundness of the abstract interpretation and of the reduction function. Eventually, by applying this rules exhaustively, the execution of $\mathbf{c}$ terminates, and we have $(\mu', \nu') \in \gamma_{\mathbb{K}}(\kappa')$ and $\nu \preceq \nu'$. Thus, the RedSoundSE semantics are sound. □

Theorem 6 (Refutation up to a bound of RedSoundSE)**.** *Let* $\boldsymbol{c}$ *be a command,* $(\kappa, a), (\kappa', a') \in \mathbb{K} \times \mathbb{A}$ *be two product precise stores,* $w, w' \in \mathbb{W}$*, such that* $(\boldsymbol{c}, (\kappa, a), w, \boldsymbol{tt}) \rightharpoonup_{s\times\mathbb{A}} (\texttt{skip}, (\kappa', a'), w', \boldsymbol{tt})$*. Then, for all* $(\mu', \nu') \in \gamma_{\mathbb{K}\times\mathbb{A}}(\kappa', a')$*, it exists* $(\mu, \nu) \in \gamma_{\mathbb{K}\times\mathbb{A}}(\kappa, a)$ *such that* $(\boldsymbol{c}, \mu) \rightharpoonup_{s\times\mathbb{A}} (\texttt{skip}, \mu')$*.*

Proof. The RedSoundSE semantics has a boolean flag b that starts as true ($\mathbf{tt}$). Applications of rule S-A-NEXT do not alter the precision flag. Therefore, the only way to have a false ($\mathbf{ff}$) precision flag is to apply rule S-A-APPROX-MANY. Indeed, this rule over-approximates states, and loses completeness of the analysis. Thus, Theorem 2 still applies, and this theorem holds. □

Chapter 5

SoundRSE: Sound Relational Symbolic Execution

As discussed in Chapter 1, security properties like noninterference require to reason over *pairs* of execution traces thus we now set up a *sound relational symbolic execution* technique that constructs pairs of executions. This analysis will be regarded as SoundRSE. The approach will be very similar to that of SoundSE, but with *relational symbolic expressions*.

1 Standard relational symbolic execution

1.1 Relational symbolic expressions and stores

We first define the notions of relational expression, relational store, and precise relational store.

An approach to define a relational symbolic state would be to simply build pairs of symbolic states (κ_0, κ_1). However, this representation is redundant. Indeed, to attempt proving noninterference we need to express that both symbolic states agree on low variables, which can be captured by adding equalities over symbolic values. Yet, it is more efficient to *share* common fragments of the symbolic stores and symbolic paths. Intuitively, instead of writing $\kappa_0 = [\mathtt{i} \mapsto (\imath_0)]$, $\kappa_1 = [\mathtt{i} \mapsto (\imath_1)]$, and $\imath_0 = \imath_1$, we may simply have a single symbolic variable $\imath$ and a single symbolic memory $[\mathtt{i} \mapsto (\imath)]$.

A *relational symbolic expression* is an element defined by the grammar: $\tilde{\varepsilon} ::= (\varepsilon) | (\varepsilon \mid \varepsilon)$ where ε ranges over the set $\mathbb{E}$ of symbolic expressions. We write $\mathbb{E}_2$ for the set of relational symbolic expressions. When a variable shares its value in both executions, we simply write $\mathtt{i} \mapsto (\varepsilon)$. However, when the values disagree (for instance, when it is high), the variable will be mapped to *two* symbolic expressions, written $\mathtt{x} \mapsto (\varepsilon_0 \mid \varepsilon_1)$.

Definition 11 (Relational and precise relational stores)**.** *A* relational symbolic store $\tilde{\rho}$ *is a function from variables to relational symbolic expressions. We let* $\overline{\mathbb{M}}_2 = \mathbb{X} \to \mathbb{E}_2$ *stand for their set. Finally, a* precise relational store $\tilde{\kappa}$ *is a pair* $(\tilde{\rho}, \pi) \in \mathbb{K}_2$.

1.2 Store operations

Before we define concretizations of $\overline{\mathbb{M}}_2$ and $\mathbb{K}_2$, we need to introduce two operations. The first is a projection of relational symbolic stores to get one side of it. The second operation is a pairing of two symbolic stores into a relational symbolic store.

Projections The *projections* Π_0, Π_1 map relational symbolic stores into symbolic stores. They are defined in a pointwise manner, as follows:

- if $\tilde{\rho}(\mathtt{x}) = (\varepsilon)$ then $\Pi_0(\tilde{\rho})(\mathtt{x}) = \Pi_1(\tilde{\rho})(\mathtt{x}) = \varepsilon$;
- if $\tilde{\rho}(\mathtt{x}) = (\varepsilon_0 \mid \varepsilon_1)$, then $\Pi_0(\tilde{\rho})(\mathtt{x}) = \varepsilon_0$ and $\Pi_1(\tilde{\rho})(\mathtt{x}) = \varepsilon_1$.

We overload the Π_0, Π_1 notation and also apply it to double symbolic expressions: $\Pi_0((\varepsilon)) = \Pi_1((\varepsilon)) = \varepsilon$ and if $\tilde{\varepsilon} = (\varepsilon_0 \mid \varepsilon_1)$, then $\Pi_0(\tilde{\varepsilon}) = \varepsilon_0$ and $\Pi_1(\tilde{\varepsilon}) = \varepsilon_1$.

Pairing The *pairing* $(\!|\rho_0 \mid \rho_1|\!)$ of two symbolic stores ρ_0 and ρ_1 is a relational symbolic store defined such that, for all variable $\mathtt{x}$,

$$(\!|\rho_0 \mid \rho_1|\!)(\mathtt{x}) = \begin{cases} (\varepsilon) & \text{if } \rho_0(\mathtt{x}) \text{ and } \rho_1(\mathtt{x}) \text{ are provably equal to } \varepsilon \in \mathbb{E} \\ (\rho_0(\mathtt{x}) \mid \rho_1(\mathtt{x})) & \text{otherwise} \end{cases}$$

where the notion of "provably equal" may boil down to syntactic equality of symbolic expressions or involve an external proving tool.

1.3 Concretization functions

We can now define the concretization functions for relational symbolic stores and relational precise stores. These are analogous to the concretizations in Chapter 3.

Definition 12 (Concretization functions)**.** *The* concretization of relational stores $\gamma_{\overline{\mathbb{M}}_2}$ *and* concretization of precise relational stores $\gamma_{\mathbb{K}_2}$ *are defined by:*

$$\begin{array}{rlll} \gamma_{\overline{\mathbb{M}}_2} : & \overline{\mathbb{M}}_2 & \longrightarrow & \mathcal{P}(\mathbb{M} \times \mathbb{M} \times (\overline{\mathbb{V}} \to \mathbb{V})) \\ & \tilde{\rho} & \longmapsto & \{(\mu_0, \mu_1, \nu) \mid \forall \mathtt{x} \in \mathbb{X},\ \forall i \in \{0,1\},\ \mu_i(\mathtt{x}) = [\![\Pi_i(\tilde{\rho})(\mathtt{x})]\!](\nu)\} \\ \gamma_{\mathbb{K}_2} : & \mathbb{K}_2 & \longrightarrow & \mathcal{P}(\mathbb{M} \times \mathbb{M} \times (\overline{\mathbb{V}} \to \mathbb{V})) \\ & (\tilde{\rho}, \pi) & \longmapsto & \{(\mu_0, \mu_1, \nu) \in \gamma_{\overline{\mathbb{M}}_2}(\tilde{\rho}) \mid [\![\pi]\!](\nu) = \boldsymbol{tt}\}. \end{array}$$

1.4 Relational symbolic execution states and commands

Similarly to non-relational states of SoundSE in Chapter 3, relational states, denoted $\tilde{\kappa}$, are 4-tuples with a command, a relational precise store, and components w and b.

$$\tilde{\kappa} = (\mathbf{c}, \tilde{\kappa}, w, b)$$

Since relational symbolic execution aims at describing pairs of executions, it should account for the case where the two executions follow different control flow paths. Thus, a relational

symbolic state may consist of a single command when both executions follow the same path, or two commands when they diverge. For this matter, we introduce a new type of language command, called *composition*, denoted by $\bowtie$. Thus, the language syntax is extended with special composed commands $\mathbf{c}_{\bowtie}$.

$$\mathbf{c}_{\bowtie} ::= \mathbf{c} \mid \mathbf{c} \bowtie \mathbf{c}$$

When both executions are following the same path, symbolic states remain the same. However, the executions are not following the same path, for instance, when an `if` statement is executed, the command is split in two and symbolic states are of the shape

$$((\mathbf{c}_0 \bowtie \mathbf{c}_1; \mathbf{c}_2), \tilde{\kappa}, w, b)$$

Command $\mathbf{c}_0$ (resp., $\mathbf{c}_1$) denotes the control state of the first (resp., second) execution. When both executions finishing executing their respective command, they "meet" and execute $\mathbf{c}_2$.

Notice that, when we define the semantic rules for the composition command, the counter can still be implemented as in Example 3.

1.5 Evaluation of expressions

For the evaluation of expressions, we can assume For an expression $\mathbf{e}$, the relational evaluation is denoted by Evaluation of expressions now has two account for relational stores. For that, we overload evaluation $\Downarrow$, to take an expression $\mathbf{e}$ and a relational store $\tilde{\rho}$, and return a relational symbolic expression $\tilde{\varepsilon}$. We define the relational evaluation of expressions through the standard one.

$$\tilde{\rho} \vdash \mathbf{e} \Downarrow \tilde{\varepsilon} \quad \text{where} \quad \tilde{\varepsilon} = \begin{cases} (\varepsilon_0) & \text{if } \varepsilon_0 = \varepsilon_1 \\ (\varepsilon_0 \mid \varepsilon_1) & \text{otherwise} \end{cases} \quad \text{with} \quad \begin{array}{l} \Pi_0(\tilde{\rho}) \vdash \mathbf{e} \Downarrow \varepsilon_0 \\ \Pi_1(\tilde{\rho}) \vdash \mathbf{e} \Downarrow \varepsilon_1 \end{array}$$

1.6 Relational symbolic execution semantics

We write $\rightharpoonup_{sr}$ for the relational symbolic execution step relation. Rules are shown in Figure 5.1, and discussed here.

SR-ASSIGN evaluates expression $\mathbf{e}$, returning a relational symbolic expression. The expression is then mapped in the relational symbolic store.

SR-SEQ executes a step on sequence command $\mathbf{c}_0; \mathbf{c}_1$, executing $\mathbf{c}_0$ and updating the precise relational symbolic store. If $\mathbf{c}_0$ does not finish, it is replaced by $\mathbf{c}_0'$, and the next will continue executing it. Once the head of the sequence is `skip`, SR-SEQ-EXIT exits the sequence, leaving just $\mathbf{c}_1$.

SR-IF-TT and SR-IF-FF either consider the guard to hold or to not hold in both executions. This results in picking the corresponding command $\mathbf{c}_0$ or $\mathbf{c}_1$, and continuing execution normally, with an added constraint.

SR-IF-TF is applied when the first execution follows the true branch, and the second execution follows the false branch of an `if` statement. This results in the composition of commands with

the $\bowtie$ operator. The guard is evaluated and added as a constraint, and the semantics will then execute both executions separately. The same logic applies for SR-IF-FT.

When an `while` statement is met, if $\mathfrak{step}$ is true, that is, we can unroll the loop, rule SR-LOOP-TT is applied and the body of the loop is appended to the loop, and a new constraint is added. Rule SR-LOOP-FF works similarly, but instead udpates the command to `skip`.

SR-LOOP-TF and SR-LOOP-FT work similarly to SR-IF-TF and SR-IF-FT, as long as $\mathfrak{step}$ is true. Since the executions become totally separated until they meet after the loop, the counter does not need to be modified at all.

The most interesting rules are SR-COMP-R and SR-COMP-L. SR-COMP-L executes the left command of a composition. To do so, it uses a non-relational symbolic execution, such as SoundSE or RedSoundSE. Eventually, this command will reach `skip`, at which point SR-COMP-R starts being applied. When both commands eventually are `skip`, rule SR-COMP-EXIT consumes the empty composition. We say that the two executions "meet" when SR-COMP-EXIT is applied.

$$\text{SR-ASSIGN}\ \frac{\tilde{\rho} \vdash \mathbf{e} \Downarrow \tilde{\varepsilon}}{(\mathbf{x} = \mathbf{e}, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\texttt{skip}, (\tilde{\rho}[\mathbf{x} \mapsto \langle\!\langle \tilde{\varepsilon} \rangle\!\rangle], \pi))}$$

$$\text{SR-SEQ}\ \frac{(\mathbf{c}_0, \tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}_0', \tilde{\kappa}')}{(\mathbf{c}_0; \mathbf{c}_1, \tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}_0'; \mathbf{c}_1, \tilde{\kappa}')} \qquad \text{SR-SEQ-EXIT}\ \frac{}{(\texttt{skip}; \mathbf{c}_1, \tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}_1, \tilde{\kappa})}$$

$$\text{SR-IF-TT}\ \frac{\rho \vdash \mathbf{b} \Downarrow \tilde{\beta} \qquad \pi' = \pi \wedge \Pi_0(\tilde{\beta}) \wedge \Pi_1(\tilde{\beta}) \qquad \mathbf{may}(\pi')}{(\texttt{if}\ \mathbf{b}\ \texttt{then}\ \mathbf{c}_0\ \texttt{else}\ \mathbf{c}_1, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\mathbf{c}_0, (\tilde{\rho}, \pi'))}$$

$$\text{SR-IF-TF}\ \frac{\rho \vdash \mathbf{b} \Downarrow \tilde{\beta} \qquad \pi' = \pi \wedge \Pi_0(\tilde{\beta}) \wedge \neg\Pi_1(\tilde{\beta}) \qquad \mathbf{may}(\pi')}{(\texttt{if}\ \mathbf{b}\ \texttt{then}\ \mathbf{c}_0\ \texttt{else}\ \mathbf{c}_1, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\mathbf{c}_0 \bowtie \mathbf{c}_1, (\tilde{\rho}, \pi'))}$$

$$\text{SR-IF-FT}\ \frac{\rho \vdash \mathbf{b} \Downarrow \tilde{\beta} \qquad \pi' = \pi \wedge \neg\Pi_0(\tilde{\beta}) \wedge \Pi_1(\tilde{\beta}) \qquad \mathbf{may}(\pi')}{(\texttt{if}\ \mathbf{b}\ \texttt{then}\ \mathbf{c}_0\ \texttt{else}\ \mathbf{c}_1, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\mathbf{c}_1 \bowtie \mathbf{c}_0, (\tilde{\rho}, \pi'))}$$

$$\text{SR-IF-FF}\ \frac{\rho \vdash \mathbf{b} \Downarrow \tilde{\beta} \qquad \pi' = \pi \wedge \neg\Pi_0(\tilde{\beta}) \wedge \neg\Pi_1(\tilde{\beta}) \qquad \mathbf{may}(\pi')}{(\texttt{if}\ \mathbf{b}\ \texttt{then}\ \mathbf{c}_0\ \texttt{else}\ \mathbf{c}_1, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\mathbf{c}_0, (\tilde{\rho}, \pi'))}$$

$$\text{SR-LOOP-TT}\ \frac{\rho \vdash \mathbf{b} \Downarrow \tilde{\beta} \qquad \pi' = \pi \wedge \Pi_0(\tilde{\beta}) \wedge \Pi_1(\tilde{\beta}) \qquad \mathbf{may}(\pi')}{(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\mathbf{c}_0; \texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0, (\tilde{\rho}, \pi))}$$

$$\text{SR-LOOP-TF}\ \frac{\rho \vdash \mathbf{b} \Downarrow \tilde{\beta} \qquad \pi' = \pi \wedge \Pi_0(\tilde{\beta}) \wedge \neg\Pi_1(\tilde{\beta}) \qquad \mathbf{may}(\pi')}{(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} ((\mathbf{c}_0; \texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0) \bowtie \texttt{skip}, (\tilde{\rho}, \pi'))}$$

$$\text{SR-LOOP-FT}\ \frac{\rho \vdash \mathbf{b} \Downarrow \tilde{\beta} \qquad \pi' = \pi \wedge \neg\Pi_0(\tilde{\beta}) \wedge \Pi_1(\tilde{\beta}) \qquad \mathbf{may}(\pi')}{(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\texttt{skip} \bowtie (\mathbf{c}_0; \texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0), (\tilde{\rho}, \pi'))}$$

$$\text{SR-LOOP-FF}\ \frac{\rho \vdash \mathbf{b} \Downarrow \tilde{\beta} \qquad \pi' = \pi \wedge \neg\Pi_0(\tilde{\beta}) \wedge \neg\Pi_1(\tilde{\beta}) \qquad \mathbf{may}(\pi')}{(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\texttt{skip}, (\tilde{\rho}, \pi'))}$$

$$\text{SR-COMP-L}\ \frac{(\mathbf{c}_0, (\Pi_0(\tilde{\rho}), \pi)) \rightharpoonup_{s} (\mathbf{c}_0', (\rho_0', \pi'))}{(\mathbf{c}_0 \bowtie \mathbf{c}_1, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\mathbf{c}_0' \bowtie \mathbf{c}_1, (\langle\!|\rho_0' \mid \Pi_1(\tilde{\rho})|\!\rangle, \pi'))}$$

$$\text{SR-COMP-R}\ \frac{(\mathbf{c}_1, (\Pi_1(\tilde{\rho}), \pi)) \rightharpoonup_{s} (\mathbf{c}_1', (\rho_1', \pi'))}{(\texttt{skip} \bowtie \mathbf{c}_1, (\tilde{\rho}, \pi)) \rightharpoonup_{sr} (\texttt{skip} \bowtie \mathbf{c}_1', (\langle\!|\Pi_0(\tilde{\rho}) \mid \rho_1'|\!\rangle, \pi'))}$$

$$\text{SR-COMP-EXIT}\ \frac{}{(\texttt{skip} \bowtie \texttt{skip}, \tilde{\kappa}) \rightharpoonup_{sr} (\texttt{skip}, \tilde{\kappa})}$$

Figure 5.1: Rules of relational symbolic execution step relation.

2 SoundRSE: sound relational symbolic execution

To define SoundRSE, similarly to Chapter 3, we will define a few rules that make use of the standard relational symbolic execution.

2.1 Over-approximation of loops

Rules that define SoundRSE can be found in Figure 5.2.

The first rule is SR-NEXT, that just applies a normal step of execution.

For the over-approximation of loops, we have two rules: SR-APPROX-MANY and SR-APPROX-MANY-ABS. The standard rule, SR-APPROX-MANY, is defined with the use of an adapted function $\mathfrak{modif}$. This extension of $\mathfrak{modif}$ to relational symbolic states, maps modified variables to a pair of fresh symbolic variables. Rule SR-APPROX-MANY-ABS is done by also using the abstraction from RedSoundSE.

Finally, since two simultaneous traces can follow different paths, one execution might reach the iteration bound. When this happens, the over-approximation needs to be applied to just one execution. This is done by rules SR-APPROX-COMP-R and SR-APPROX-COMP-L.

2.2 SoundRSE with non-relational abstractions

At this point, rule SR-APPROX-MANY completely disregards the abstractions used in RedSoundSE, such as intervals and polyhedra domains. We will not reformulate all of the semantics rules of SoundRSE, but we will assume that:

- The symbolic state, originally a relational symbolic precise store, is now complemented by two abstract states a_0 and a_1.
- At each semantic step, the abstract semantics is applied over a_0 and a_1 with the current command, similar to RedSoundSE.
- For the loop over-approximation, rule SR-APPROX-MANY-ABS applies the reduction on both a_0 and a_1 after function $\mathfrak{modif}$ has been applied. The reduction function is similar to the one of RedSoundSE, doing the reduction of a_0 with the left side of the relational store, and the reduction of a_1 with the right side of the relational store.

We introduce the updated rule in Figure 5.3.

2.3 Soundness and refutation results

SoundRSE inherits similar soundness and refutation properties as SoundSE, as shown in the following theorems.

Theorem 7 (Soundness)**.** *Let $\tilde{\kappa} \in \mathbb{K}_2$, $w \in \mathbb{W}$, and $b \in \mathbb{B}$. We let $(\mu_0, \mu_1, \nu) \in \gamma_{\mathbb{K}_2}(\tilde{\kappa})$ and assume that stores μ_0', μ_1' are such that $(\boldsymbol{c}, \mu_0) \to^* (\texttt{skip}, \mu_0')$ and $(\boldsymbol{c}, \mu_1) \to^* (\texttt{skip}, \mu_1')$. Then, there exists $\tilde{\kappa}' \in \mathbb{K}_2$, a valuation ν', and a counter state $w' \in \mathbb{W}$ such that $\nu \preceq \nu'$, $(\mu_0', \mu_1', \nu') \in \gamma_{\mathbb{K}}(\tilde{\kappa}')$, and $(\boldsymbol{c}, \tilde{\kappa}, w, b) \rightharpoonup_{sr}^* (\texttt{skip}, \tilde{\kappa}', w', b')$.*

$$\text{SR-NEXT}\ \frac{(\mathbf{c},\tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}',\tilde{\kappa}') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w) = (\mathtt{tt},w')}{(\mathbf{c},\tilde{\kappa},w,b) \rightharpoonup_{sr} (\mathbf{c}',\tilde{\kappa}',w',b)}$$

$$\text{SR-APPROX-COMP-R}\ \frac{\begin{array}{c}(\mathbf{c},\tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}',\tilde{\kappa}') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w) = (\mathtt{ff},w') \qquad \mathfrak{deseq}(\mathbf{c}) = (\mathtt{skip} \bowtie \mathbf{c}_1,\mathbf{c}_2)\\ (\mathbf{c}_1,\Pi_1(\tilde{\kappa}),w,b) \rightharpoonup_{s} (\mathbf{c}_1',\kappa',w',\mathtt{ff}) \qquad \tilde{\kappa}'' = (\!|\Pi_0(\tilde{\kappa}) \mid \kappa'|\!)\end{array}}{(\mathbf{c},\tilde{\kappa},w,b) \rightharpoonup_{sr} (\mathtt{skip} \bowtie \mathbf{c}_1';\mathbf{c}_2,\tilde{\kappa}'',w',\mathtt{ff})}$$

$$\text{SR-APPROX-COMP-L}\ \frac{\begin{array}{c}(\mathbf{c},\tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}',\tilde{\kappa}') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w) = (\mathtt{ff},w') \qquad \mathfrak{deseq}(\mathbf{c}) = (\mathbf{c}_0 \bowtie \mathbf{c}_1,\mathbf{c}_2)\\ (\mathbf{c}_0,\Pi_0(\tilde{\kappa}),w,b) \rightharpoonup_{s} (\mathbf{c}_0',\kappa',w',\mathtt{ff}) \qquad \tilde{\kappa}'' = (\!|\kappa' \mid \Pi_1(\tilde{\kappa})|\!)\end{array}}{(\mathbf{c},\tilde{\kappa},w,b) \rightharpoonup_{sr} (\mathbf{c}_0' \bowtie \mathbf{c}_1;\mathbf{c}_2,\tilde{\kappa}'',w',\mathtt{ff})}$$

$$\text{SR-APPROX-MANY}\ \frac{\begin{array}{c}(\mathbf{c},\tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}',\tilde{\kappa}') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w) = (\mathtt{ff},w')\\ \mathfrak{deseq}(\mathbf{c}) = (\mathbf{c}_0,\mathbf{c}_1) \qquad \mathfrak{modif}(\tilde{\kappa},\mathbf{c}_0) = \tilde{\kappa}''\end{array}}{(\mathbf{c},\tilde{\kappa},w,b) \rightharpoonup_{sr} (\mathbf{c}_1,\tilde{\kappa}'',w',\mathtt{ff})}$$

Figure 5.2: Rule for loop over-approximation in SoundRSE.

$$\text{SR-APPROX-MANY-ABS}\ \frac{\begin{array}{c}(\mathbf{c},\tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}',\tilde{\kappa}') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w) = (\mathtt{ff},w')\\ \mathfrak{deseq}(\mathbf{c}) = (\mathbf{c}_0,\mathbf{c}_1) \qquad \mathfrak{modif}(\tilde{\kappa},\mathbf{c}_0) = \tilde{\kappa}''\\ i \in \{0,1\} \qquad a_i' = [\![\mathbf{c}_0]\!]^\sharp_{\mathbb{A}}(a_i) \qquad \mathfrak{reduction}(\tilde{\kappa}'',a_0',a_1') = (\tilde{\kappa}''',a_0'',a_1'')\end{array}}{(\mathbf{c},(\tilde{\kappa},a_0,a_1),w,b) \rightharpoonup_{sr} (\mathbf{c}_1,(\tilde{\kappa}'',a_0'',a_1''),w',\mathtt{ff})}$$

Figure 5.3: Rule SR-APPROX-MANY-ABS.

Theorem 8 (Refutation up to a bound)**.** *Let $\boldsymbol{c}$ be a command, $\tilde{\kappa},\tilde{\kappa}' \in \mathbb{K}_2$ be two precise stores, $w,w' \in \mathbb{W}$, such that $(\boldsymbol{c},\kappa,w,\boldsymbol{tt}) \rightharpoonup_{sr}^* (\mathtt{skip},\kappa',w',\boldsymbol{tt})$. Then, for all $(\mu_0',\mu_1',\nu') \in \gamma_{\mathbb{K}_2}(\kappa')$, it exists $(\mu_0,\mu_1,\nu) \in \gamma_{\mathbb{K}_2}(\kappa)$ such that $(\boldsymbol{c},\mu_0) \to^* (\mathtt{skip},\mu_0')$ and $(\boldsymbol{c},\mu_1) \to^* (\mathtt{skip},\mu_1')$.*

Proof. Proof follows that of Theorem 4 □

3 SoundRSE-based analysis of noninterference

We now assume a program $(\mathbf{c},L)$, and show the application of SoundRSE analysis to attempt proving noninterference. The analysis proceeds according to the following steps:

1. Construction of the initial store $\tilde{\rho}_0$ such that, for all variables $\mathtt{x}$ present in $\mathbf{c}$, $\tilde{\rho}_0(\mathtt{x}) = (\boldsymbol{x})$ (resp., $\tilde{\rho}_0(\mathtt{x}) = (\boldsymbol{x}_0 \mid \boldsymbol{x}_1)$) if $\mathtt{x} \in L$ (resp., $\mathtt{x} \notin L$), and where $\boldsymbol{x}$ is a fresh symbolic value (resp., $\boldsymbol{x}_0,\boldsymbol{x}_1$ are fresh symbolic values).

2. Exhaustive application of semantic rules from initial state $(\mathbf{c},(\tilde{\rho}_0,\mathtt{tt}),w_0,\mathtt{tt})$; we let $\mathcal{O}$ stand for the set of final precise relational stores with their precision flags: $\mathcal{O} \triangleq \{(\tilde{\kappa},b) \mid \exists w \in \mathbb{W},\ (\mathbf{c},(\tilde{\rho}_0,\mathtt{tt}),w_0,\mathtt{tt}) \rightharpoonup_{sr} (\mathtt{skip},\tilde{\kappa},w,b)\}$.

3. *Attempt to prove noninterference* for each symbolic path in $\mathcal{O}$ using an external tool, such as an SMT solver; more precisely, given $((\tilde{\rho}, \pi), b) \in \mathcal{O}$,
   - if π is not satisfiable, the path is infeasible and can be ignored;
   - if it can be proved that for all variables $\mathtt{x} \in L$, there is a unique value, i.e., $\Pi_0(\tilde{\rho})(\mathtt{x}) = \Pi_1(\tilde{\rho})(\mathtt{x})$, then the program is noninterferent;
   - if a valuation ν can be found, such that $[\![\pi]\!](\nu) = \mathbf{tt}$ (the path is satisfiable), and there exists a variable $\mathtt{x} \in L$ such that $[\![\Pi_0(\tilde{\rho})(\mathtt{x})]\!](\nu) \neq [\![\Pi_1(\tilde{\rho})(\mathtt{x})]\!](\nu)$, and $b = \mathbf{tt}$, then ν provides a counter-example refuting noninterference;
   - finally, if $b = \mathbf{ff}$ and neither of the above cases occurs, no conclusive answer can be given for this path.

To summarize, the analyser either proves noninterference (when all paths are either not satisfiable or noninterferent), or it provides a valuation that refutes noninterference (when such a valuation can be found for at least one path), or it does not conclude. When a refutation is found, this refutation actually defines a real attack.

```
1 # 𝕙 → priv ; 𝕝 → i
2 while i > priv:
3   i += 1
4   priv += 2
```

To illustrate the refutation capabilities of SoundRSE, we reintroduce Program 3.4, where the final value of `i` depends on `priv`. This program always terminates since the value of `i` is growing slower than that of `priv`.

Example 11 (Noninterference)**.** *Applying SoundRSE to Program 1.2a, goes through the four interleavings. When finishing executing the* `if`*, all the relational stores have that* `y` *maps to 5. Therefore, the program is verified noninterferent.*

Program 1.2b eventually reaches the unrolling limit and rule SR-APPROX-MANY *is applied. When the rule is applied,* `i` *is assigned* $(i_1 \mid i_2)$*, and their equality cannot be proven. Therefore, the program cannot be verified. If we instance SoundRSE with RedSoundSE, where the value of* `i` *is 10, then the program can be verified.*

The analysis of Program 3.4 computes at least one interferent path if the unrolling bound is set to any strictly positive integer.

3.1 Refutation of programs with respect to noninterference

We consider two new examples for showing the refutation of programs, found in Figure 5.4.

Example 12. *Program 5.4a copies the value of* `priv` *into* $\mathtt{y}_0$ *after three executions of the loop. Variables* $\mathtt{y}_1$ *and* $\mathtt{y}_2$ *are cleared after the loop by getting value 0 assigned, but* $\mathtt{y}_0$ *retains the value of* `priv`*.*

```
1 i = 0
2 while (i < 3):
3     y0 = y1
4     y1 = y2
5     y2 = priv
6     i += 1
7 y1, y2 = 0, 0
```

(a) Insecure

```
1 i = 0
2 while (i < 100):
3     if (priv > 0):
4         y = 5
5     i += 1
```

(b) Insecure

Figure 5.4: Unsafe programs with respect to Noninterference. Program (a) can be refuted by SoundSE, while program (b) cannot.

By using SoundRSE, from initial store

$$\tilde{\rho} = [\mathtt{i} \mapsto (i); \mathtt{y}_0 \mapsto (y_0); \mathtt{y}_1 \mapsto (y_1); \mathtt{y}_2 \mapsto (y_2); \mathtt{priv} \mapsto (p_0 \mid p_1)],$$

if the iteration bound is greater than three (meaning that SoundRSE accesses the loop the three), the loop does not need to be over approximated.

The final symbolic store is

$$\tilde{\rho} = [\mathtt{i} \mapsto (i); \mathtt{y}_0 \mapsto (p_0 \mid p_1); \mathtt{y}_1 \mapsto (0); \mathtt{y}_2 \mapsto (0); \mathtt{priv} \mapsto (p_0 \mid p_1)],$$

and then a valuation ν exists such that it satisfies this trace, and the final value of $\mathtt{y}_0$ disagrees between the two executions. For example,

$$\nu = [p_0 \mapsto 0; p_1 \mapsto 1; \dots].$$

Example 13. *Program 5.4b is also unsafe, but instead of accessing the loop 3 times, we need to access it a hundred times. We can keep raising the number of iterations, but clearly there has to be a limit, either for space explosion, or for limiting the time to analyze programs. Because of this, we assume that the bound of iterations is lower than 100.*

In this case, the analysis will have to over approximate the value of `y`*. This will cause the analyzer to not be able to prove that the program is noninterferent, neither insecure. Hence, the response is inconclusive.*

Chapter 6

RedSoundRSE: dependences and SoundRSE

Dependences analysis consists on tracking the dependency between variables in a program, ignoring the actual values. While dependences are imprecise, they can track the relation between variables, making it good for hyperproperties such as Noninterference.

Some programs like Program 1.2b, where an `while` statement with a low guard performs assignments, can be verified by dependences. However, the more precise SoundRSE cannot, steming from the broad over-approximation of loops.

In this chapter, we set up a novel form of product of abstractions, to benefit from dependences in the symbolic semantics. This notion of product is generic and does not require to fix a specific dependency abstraction. We refer to the final analysis presented in this section as RedSoundRSE (Reduced Sound Relational Symbolic Execution).

1 Dependences semantics

Although dependence abstractions may take many forms, they all characterize information flows that can be observed by comparing pairs of executions. For instance, [Assaf, 2017] uses a lattice of security levels and abstract elements map each level to a set of variables. We will use this representation for our purposes.

The general definition for dependences is as follows.

Definition 13. *Let* $\mathbb{Sec}$ *be a* security lattice*, where* $s \in \mathbb{Sec}$ *is a security level. We consider that* $\mathbb{H}$ *is always the top of the lattice, and* $\mathbb{L}$ *is the bottom. Let* $\mathbb{D}$ *be an* abstract lattice*, from security levels to variables. A* dependences state $d \in \mathbb{D}$ *is a mapping from* $\mathbb{Sec} \to \mathcal{P}(\mathbb{X})$ *(security levels to set of variables). Then, assuming for any* $d \in \mathbb{D}$*, and* $s \in \mathbb{Sec}$*, typing* $d(s)$ *returns the set of variables mapped to that security level in* d*.*

For the dependences abstraction, given a dependences state d, its concretization is a set of pairs of memories, that share the same value for all low variables.

Definition 14 (Dependences abstraction)**.** *A* dependences abstraction *is defined by an abstract*

lattice $\mathbb{D}$, and a concretization function

$$\begin{array}{rccl} \gamma_{\mathbb{D}} : & \mathbb{D} & \longrightarrow & \mathcal{P}(\mathbb{M} \times \mathbb{M}) \\ & d & \longmapsto & \{(\mu_0, \mu_1) \in \mathbb{M} \times \mathbb{M} \mid \mu_0 =_{d(\mathbb{L})} \mu_1\} \end{array}$$

Now, based on the concretization function we can define a sound dependences analysis.

Definition 15. *A* sound dependences analysis *is defined by a function $[\![\boldsymbol{c}]\!]^\sharp_{\mathbb{D}} : \mathbb{D} \to \mathbb{D}$ such that, for all $d \in \mathbb{D}$, $(\mu_0, \mu_1) \in \gamma_{\mathbb{D}}(d)$,*

$$\{(\mu_0', \mu_1') \in \mathbb{M} \times \mathbb{M} \mid \forall i \in \{0,1\},\ (\boldsymbol{c}, \mu_i) \to^* (\texttt{skip}, \mu_i')\} \subseteq \gamma_{\mathbb{D}} \circ [\![\boldsymbol{c}]\!]^\sharp_{\mathbb{D}}(d).$$

Example 14 (Standard dependence based abstraction [Assaf, 2017])**.** *The abstraction of [Assaf, 2017] is an instance of Definition 14. Let $\{\mathbb{L}, \mathbb{H}\}$ be the set of security levels. Assume an initial abstract state d that captures pairs of concrete stores that are low equal for some program $(\boldsymbol{c}, L)$. By applying the dependence analysis, if the final dependence state has a low dependency for each initially low variable, the program is noninterferent.*

In practice such information is computed by forward abstract interpretation, using syntactic dependencies for expressions and conditions, and conservatively assuming conditions may generate (implicit) flows to any operation that they guard.

We note that Definition 14 accounts not only for dependence abstractions such as that of [Assaf, 2017]. In particular, [Delmas, 2019] proposes a semantic patch analysis which can also be applied to security properties by using a relational abstract domain to relate pairs of executions; such analyses use an abstraction that also writes as in Definition 14. In the following, we assume a sound dependence analysis is fixed.

2 Product of symbolic execution and dependence analysis

Rules of SoundRSE defined in Figure 5.1 introduce no imprecision. The exception is rule SR-APPROX-MANY, from Figure 5.2, applied when the execution bound is reached. Therefore, the principle of the combined analysis is to replace this imprecise rule with another that uses dependence analysis results to strengthen relational stores.

First, we need to introduce two operations to transport information in a sound manner into and from the dependence abstract domain.

Definition 16 (Information translation)**.** *The* translation from symbolic to dependences *is defined as follows*

$$\tau_{s \to \mathbb{D}} : \overline{\mathbb{M}}_2 \to \mathbb{D}$$
$$\tilde{\rho} \mapsto \begin{cases} \mathbb{L} \mapsto \{\mathtt{x} \mid \mathbf{may}(\Pi_0(\tilde{\rho})(\mathtt{x}) = \Pi_1(\tilde{\rho})(\mathtt{x}))\} \\ \mathbb{H} \mapsto \{\mathtt{x} \mid \neg\, \mathbf{may}(\Pi_0(\tilde{\rho})(\mathtt{x}) = \Pi_1(\tilde{\rho})(\mathtt{x}))\} \end{cases}$$

Definition 16 takes a relational symbolic store, and creates an according dependences state. This dependences state can then in turn be used to execute with the dependences semantics.

Definition 17 (Dependences extraction)**.** *The* extraction of dependences information *is a function* $\lambda_{\mathbb{D}\to\mathbb{L}} : \mathbb{D} \to \mathcal{P}(\mathbb{X})$ *defined as follows:*

$$\begin{aligned}\lambda_{\mathbb{D}\to\mathbb{L}} : \mathbb{D} &\to \mathcal{P}(\mathbb{X}) \\ d &\mapsto \{\mathtt{x} \mid d(x) = \mathbb{L}\}\end{aligned}$$

Function $\lambda_{\mathbb{D}\to\mathbb{L}}$ *is sound in the following sense:*

$$\forall d \in \mathbb{D},\ \forall(\mu_0, \mu_1) \in \gamma_{\mathbb{D}}(d),\ \mu_0 =_{\lambda_{\mathbb{D}\to\mathbb{L}}(d)} \mu_1$$

Similarly, the function $\lambda_{\mathbb{D}\to\mathbb{L}}$ extracts a set of variables which are proved to remain low, boiling down to returning $d(\mathbb{L})$.

3 RedSoundRSE

We now present the combined analysis RedSoundRSE between dependences and SoundRSE. To define it we just need to replace rule sr-approx-many, with rule sr-approx-many-depby using the information translation function and the dependences extraction function.

3.1 Reduced relational semantics

The symbolic execution step SR-APPROX-MANY-DEP is shown in Figure 6.1 and replaces rule SR-APPROX-MANY (Figure 5.2). When the execution bound is reached for a loop statement, it performs the dependence analysis of the whole loop from the dependence state derived by applying $\tau_{s\to\mathbb{D}}$ to the relational symbolic store. Then, it applies $\lambda_{\mathbb{D}\to\mathbb{L}}$ to derive the set of variables that are proved to be low by the dependence analysis. Finally, it computes a new relational symbolic store by modifying the variables according to the set of variables determined low:

- if variable $\mathtt{x}$ is low based on the $\lambda_{\mathbb{D}\to\mathbb{L}}$ output, $\mathfrak{modif}_{\mathbb{D}}$ synthesizes one fresh symbolic value $\boldsymbol{x}_{\text{new}}$ and maps it to $(\boldsymbol{x}_{\text{new}})$;
- if variable $\mathtt{x}$ cannot be proved low, $\mathfrak{modif}_{\mathbb{D}}$ synthesizes two fresh symbolic values $\boldsymbol{x}_{\text{new0}}$, $\boldsymbol{x}_{\text{new1}}$ and maps $\mathtt{x}$ to $(\boldsymbol{x}_{\text{new0}} \mid \boldsymbol{x}_{\text{new1}})$.

Remark 1 (Reduced product property)**.** *We stress the fact that the rule* SR-APPROX-MANY-DEP *may be applied multiple times during the analysis, essentially whenever a loop statement is analyzed, which is generally many times more than the number of loop commands in the program due to abstract iterations. Therefore, our analysis* cannot *be viewed as a fixed sequence of analyses. Such a decomposition (for example, where dependences analysis is ran first and symbolic execution second) would be strictly less precise than our reduced product based approach.*

$$\text{SR-APPROX-MANY-DEP}\ \frac{\begin{array}{c}(\mathbf{c},\tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}',\tilde{\kappa}') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w) = (\mathbf{ff},w') \qquad \mathfrak{deseq}(\mathbf{c}) = (\mathbf{c}_0,\mathbf{c}_1)\\ d = [\![\mathbf{c}_0]\!]^\sharp_{\mathbb{D}}(\tau_{s\to\mathbb{D}}(\tilde{\kappa})) \qquad \mathfrak{modif}_{\mathbb{D}}(\tilde{\kappa},\mathbf{c}_0,\lambda_{\mathbb{D}\to\mathbb{L}}(d)) = \tilde{\kappa}''\end{array}}{(\mathbf{c},\tilde{\kappa},w,b) \rightharpoonup_{sr\times\mathbb{D}} (\mathbf{c}_1,\tilde{\kappa}'',w',\mathbf{ff})}$$

$$\text{SR-APPROX-MANY-DEP-ABS}\ \frac{\begin{array}{c}(\mathbf{c},\tilde{\kappa}) \rightharpoonup_{sr} (\mathbf{c}',\tilde{\kappa}') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w) = (\mathbf{ff},w') \qquad \mathfrak{deseq}(\mathbf{c}) = (\mathbf{c}_0,\mathbf{c}_1)\\ d = [\![\mathbf{c}_0]\!]^\sharp_{\mathbb{D}}(\tau_{s\to\mathbb{D}}(\tilde{\kappa})) \qquad \mathfrak{modif}_{\mathbb{D}}(\tilde{\kappa},\mathbf{c}_0,\lambda_{\mathbb{D}\to\mathbb{L}}(d)) = \tilde{\kappa}'' \qquad i\in\{0,1\}\\ a'_i = [\![\mathbf{c}_0]\!]^\sharp_{\mathbb{A}}(a_i) \qquad \mathfrak{reduction}(\tilde{\kappa}'',a'_0,a'_1) = (\tilde{\kappa}''',a''_0,a''_1)\end{array}}{(\mathbf{c},(\tilde{\kappa},a_0,a_1),w,b) \rightharpoonup_{sr\times\mathbb{D}} (\mathbf{c}_1,(\tilde{\kappa}'',a''_0,a''_1),w',\mathbf{ff})}$$

Figure 6.1: RedSoundRSE: Symbolic execution approximation and product with dependence information.

3.2 Soundness and refutation properties

Under the assumption that the dependence analysis and translation operations are sound, so is the combined symbolic execution, thus Theorem 7 still holds. Moreover, the refutation property of Theorem 8 also holds.

4 RedSoundRSE-based analysis of noninterference

RedSoundRSE only deviates from SoundRSE for loop over-approximations. We present three examples of varying complexity. The first example has already been visited previously, while the last two are new, and they require dependences to be verified.

Example 15 (Combined analysis)**.** *We reintroduce Program 1.2b. This program cannot be*

```
1  # 𝕳 → priv ; 𝕷 → i,z
2  i = 0
3  while (i < z):
4      i += 1
5      priv += 2
```

verified by SoundRSE when instantiated with SoundSE (no intervals or polyhedra). By switching to RedSoundSE (either with intervals or polyhedra), the program can be analyzed successfully. Also, by using dependences and no intervals or polyhedra—that is, RedSoundRSE with SoundSE—it is also possible to verify this program.

At the stage of executing rule sr-approx-many-dep, the symbolic precise store is as follows.

$$\tilde{\rho} = [\mathtt{i} \mapsto (5); \mathtt{z} \mapsto (\boldsymbol{z}_0); \mathtt{priv} \mapsto (\boldsymbol{priv}_0 + 10 \mid \boldsymbol{priv}_1 + 10)] \qquad \pi \triangleq 5 < \boldsymbol{z}$$

The application of the information translation function returns a dependences state.

$$\tau_{s\to\mathbb{D}}(\tilde{\rho}) = d = \begin{cases} \mathbb{L} \mapsto \{\mathtt{i};\mathtt{z}\} \\ \mathbb{H} \mapsto \{\mathtt{priv}\} \end{cases}$$

```
1 if (priv > 0):
2   i = 0
3 else:
4   i = 0:
5 while (i < 10):
6   i += 1
7   priv += 5
```

(a) Secure

```
1 i = 0; w = 2
2 x = 100
3 while(i < x):
4   if (x <= 0):
5     w = priv
6   i += 2
7   x += 1
```

(b) Secure

Figure 6.2: Programs illustrating different properties of the analyzer. Variable `priv` is high.

As the guard of the loop only contains low-variables `i` *and* `z`*, applying the dependences semantics over the loop with dependences state* d *returns the same dependences state.*

$$[\![c]\!]^\sharp_{\mathbb{D}}(d) = d$$

Then, the application of the dependences extraction function returns variables `i` *and* `z`*.*

$$\lambda_{\mathbb{D}\to\mathbb{L}}(d) = \{\mathtt{i};\mathtt{z}\}$$

Finally, the application of $\mathfrak{modif}_{\mathbb{D}}$ *is as follows.*

$$\mathfrak{modif}((\tilde{\rho},\pi),(\texttt{while}\ldots),\{\mathtt{i};\mathtt{z}\}) = ([\mathtt{i}\mapsto(i_1);\mathtt{z}\mapsto(z_0);\mathtt{priv}\mapsto(priv_2 \mid priv_3)],\pi)$$

Only variables `i` *and* `priv` *are modified, making the value of* `z` *to remain the same. Variable* `i` *remained low, so it is mapped only to a single symbol. However,* $priv$ *is mapped to two values since it is a high variable.*

Thus, the program can be verified noninterferent with RedSoundRSE.

Example 16. *Program 6.2a starts with an* `if` *statement that sets the value of* `i` *to 0. This* `if` *statement would make dependences analysis to fail on its own.*

Assuming the bound of iterations is less than 10, the loop needs to be over-approximated. Both variables `i` *and* `priv` *are modified, and thanks to dependences, the analysis is able to determine that the value of* `i` *should agree between both traces, and the program is proven to be noninterferent.*

Example 17. *Program 6.2b requires more than RedSoundRSE with dependences. The loop will always terminate because* `i` *grows faster thatn* `x`*, but the amount of iterations depends both on* `i` *and* `x`*.*

At the moment before applying the over-approximation rule, the state is as follows.

$$\tilde{\kappa} = ([\mathtt{i}\mapsto 2b;\ \mathtt{x}\mapsto b+100;\ \mathtt{w}\mapsto 2;\ \mathtt{priv}\mapsto(p_0 \mid p_1)],(2b < b+100))$$
$$a_0 = a_1 = [\mathtt{i} = b;\mathtt{w} = 2;\mathtt{x} = b+100]$$

Executing the loop with the abstract semantics returns a fix-point of the loop.

$$[\![\texttt{while}\ldots]\!]^{\sharp}_{\mathbb{A}}(a_0) = [\texttt{i} \geq \texttt{b} + 100; \texttt{w} = 2; \texttt{x} \geq \texttt{b} + 100] = a'_0 = a'_1$$

While the value `i` *and* `x` *cannot be established precisely, the value of* `w` *is exactly 2, and that is agreed by both executions.*

By using dependences, even if the value of `i` *and* `x` *cannot be exact, it can established that there is no high dependency. Therefore, their values agree as well.*

Combining these abstractions allow us to end with the following relational store,

$$\tilde{\rho} = [\texttt{i} \mapsto i_1;\ \texttt{w} \mapsto 2;\ \texttt{x} \mapsto x_1;\ \texttt{priv} \mapsto (p_0 \mid p_1)] \qquad \pi = i_1 \geq x_1$$

and the program is proven noninterferent.

Chapter 7

Implementation and evaluation of Noninterference analyzer

In this section we compare our analyses among them as well as with the dependency analysis of Assaf et al. [Assaf, 2017]. To do so, we implemented prototypes of all the analyses. Our goal is not to evaluate the analyses in large code bases but to assess their differences based on programs that are small but challenging for typical noninteference analysers.

1 Implementation

We prototype the analyses proposed in this work as well as the dependency analysis, intervals and convex polyhedra analysis. The prototype is implemented in around 4k lines of OCaml code, using the Apron library [Jeannet, 2009] for the numerical domains and the Z3 SMT solver [Moura, 2008]. By defining a shared interface for SoundSE and RedSoundSE, the implementation of RedSoundRSE is parameterized by these. The analyzer can be found in github.com/ignatirabo/sound-rse.

1.1 Limitations

As RedSoundSE is sound and automatic, it necessarily fails to achieve completeness (by Rice's Theorem [Jr, 1987; Asperti, 2008]), which means that there exist secure input programs that cannot be proven so. Obvious reasons for that are the approximations performed in the dependence abstraction and in the state abstraction. In return, we provide completeness up to a bound.

Our analysis is built so that the abstractions used are parameters and can be modified or switched, for instance by switching from interval abstraction to convex polyhedra, yet each abstract domain has limited expressiveness. Likewise, the computation of the variables that may be modified in a loop (rule S-A-MANY) is also over-approximated.

Another more subtle limitation is that the numerical abstraction are applied at the level of the single symbolic execution (RedSoundSE). This means that these abstractions cannot track down relations between executions, but just local constraints. This means that more complex numerical relations across symbolic variables that stem from distinct paths will not be tracked down,

indirectly limiting the ability to discover that certain pairs of paths are infeasible. Potentially, this could be achieved by using the concept of self-composition [Barthe, 2004] together with a relational numerical abstract domain (although at a much increased cost).

In general, the set of modified command variables cannot be computed in finite time. The over-approximation provided by us makes use of SoundSE. It would be possible to better approximate the set by using RedSoundSE, but we believe that it is not cost-effective. In any case, calculating this set is always an over-approximation.

2 Evaluation

We compare the 3 different relational techniques using different single-trace analyses by evaluating them on a set of challenging examples. Our results are shown in Table 7.1. In the following, we split NI programs from non NI ones. For the latter we look at the refutation capabilities of the analysis.

Relational Analysis		$\mathbb{D}$	SoundRSE		RedSoundRSE ($\mathbb{D}$)	
relational analysis input:		None	SoundSE	RedSoundSE	SoundSE	RedSoundSE
Program	**Secure?**					
Fig. 1.2a	Yes	✗ False alarm	✓ Secure	✓ Secure **(I,P)**	✓ Secure	✓ Secure **(I,P)**
Fig. 1.2b	Yes	✓ Secure	✗ False alarm	✓ Secure **(P)**	✓ Secure	✓ Secure **(I,P)**
Fig. 4.4	Yes	✗ False alarm	✗ False alarm	✓ Secure **(I,P)**	✗ False alarm	✓ Secure **(I,P)**
Fig. 6.2a	Yes	✗ False alarm	✗ False alarm	✗ False alarm	✓ Secure	✓ Secure **(I,P)**
Fig. 6.2b	Yes	✗ False alarm	✗ False alarm	✗ False alarm	✗ False alarm	✓ Secure **(I,P)**
Fig. 3.4	No	✓ Alarm	✓ Refutation model	✓ Refutation model	✓ Refutation model	✓ Refutation model
Fig. 5.4a	No	✓ Alarm	✓ Refutation model	✓ Refutation model	✓ Refutation model	✓ Refutation model
Fig. 5.4b	No	✓ Alarm	✓ Alarm	✓ Alarm	✓ Alarm	✓ Alarm

Table 7.1: Evaluation and comparison of analyses combination. $\mathbb{D}$ denotes the dependency analysis of [Assaf, 2017]. Symbol ✓ (resp., ✗) denotes a semantically correct (resp., incorrect) analysis outcome, with either a proof of security, a (possibly false) alarm, or a refutation model. For RedSoundSE columns, when the analyses succeed to prove NI, we mark the result with **I** (resp. **P**) to indicate that the intervals (resp. polyhedra) domain is being used.

2.1 Comparison of verification capabilities

Table 7.1 synthesizes our findings, displaying all the programs that we have studied in this part.

Programs 1.2a and 1.2b display the basic differences dependences and relational symbolic execution, motivating the work we have done in this part.

Program 4.4 displays the capabilities of non-relational numerical domains such as intervals and polyhedra, by using them to exactly calculate the value of variables, thus, proving noninterference.

In Program 6.2a, the first condition renders dependence analysis useless as it will consider variable `i` high. This program will also fail to be verified by SoundRSE if the iteration bound is lower than 10: in this case, `i` will be assigned a fresh symbolic value and hence be deemed high. In contrast, RedSoundRSE can determine that the value of `i` in the loop does not depend on `priv`.

Program 6.2b is more convoluted. The analysis requires both numerical and dependence abstractions in order to prove its NI. The analysis will determine (conservatively) that three variables are modified in the loop: `x`, `i` and `w`. Dependence analysis can determine that variable `i` and `x` are low even if both are modified. However, since `w` depends on `x`, and the exact value of `x` is unknown, it is not possible to determine that `w` is low. By adding a numerical domain, it is easy to track that the value of `x` is always positive, which implies that the `if` statement can never be executed.

2.2 Comparison of refutation capabilities

Since SoundRSE and RedSoundRSE unroll loops a bounded number of times, there are insecure programs for which a refutation model can be found, and programs where this is not possible. Notice that, to refute a program with a model, it is required that the symbolic execution did not perform any over approximation, i.e. that the precision flag is set to false when the analysis finds the violation. Therefore, the results for insecure programs of SoundRSE are similar to those of the different combinations that rely on symbolic execution, as reflected on Figure 7.1. For Program 3.4, a valuation can be found by doing one iteration: $\nu(i_0) = \nu(i_1) = 1$ and $\nu(priv_0) = 0$, $\nu(priv_1) = 1$. For Program 5.4a, a model can be found if the bound of iterations is set to 4 or higher. The valuation ν just needs to map variable `priv` to two different values: $\nu(priv_0) \neq \nu(priv_1)$. In Program 5.4b, for any user-set bound lower than 100 the execution will have to overapproximate, losing refutation capabilities.

Conclusion of the evaluation. We have evaluated and compared our analyses among them and with the state-of-the-art on dependency analyses [Assaf, 2017] on a set of 8 challenging examples. Our results show that, in contrast to dependencies [Assaf, 2017], analyses inherit the capacity of providing a refutation model up to a bound from symbolic execution. Moreover, RedSoundRSE instantiated with RedSoundSE is capable of soundly verifying all the examples, in contrast to all the other compared analyses, as summarized in Table 7.1.

Chapter 8

Related work

1 Hyperproperties

Noninterference was first defined by Goguen and Meseguer [Goguen, 1982], and also generalized to more powerful attacker models under the property name of declassification. We refer the reader to a survey on declassification policies [Sabelfeld, 2005] up to 2005. As discussed in the introduction, noninterference is not a safety property but a safety hyperproperty [Clarkson, 2008], a.k.a. hypersafety. Several works in the literature have shown that hypersafety verification can be reduced to verification of safety properties [Barthe, 2004; Darvas, 2005; Terauchi, 2005; Clarkson, 2008], however this reduction is not always efficient in practice [Terauchi, 2005]. In our work, we do not reduce noninterference to verification of safety but rather apply relational analyses. We only show our results using noninterference but the methodology can be easily generalized to more relaxed declassification properties, provided sound abstract domains exist.

2 Symbolic execution

Symbolic execution is a static analysis technique that was born in the 70s [Boyer, 1975; King, 1976] and that is now deployed in several popular testing tools, such as KLEE [Cadar, 2021] and NASA's Symbolic PathFinder [Pasareanu, 2008], to name a few. A primary goal and strength of symbolic execution is to find paths leading to counter-examples to generate concrete input values exercising that path. This is of particular importance to security in order to debug and confirm the feasibility of an attack when a vulnerability is detected.

Alatawi et al. [Alatawi, 2017] use abstract interpretation to enhance the precision of a dynamic symbolic execution aimed at path coverage. Their approach consists of first doing an analysis of the program with abstraction interpretation to capture indirect dependences in order to enhance path predicates. Furthermore, their analysis does not maintain soundness (nor completeness). Meanwhile, our approach continuously alternates between abstract domains and symbolic execution, keeping soundness and completeness up to a bound. Lastly, Alatawi et al. [Alatawi, 2017] do not analyze relational properties such as noninterference but just safety

properties.

We focus the rest of the related work on static analysis techniques for relational security properties: for a broader discussion on symbolic execution we refer the interested reader to a survey [Cadar, 2011] up to 2011 and an illuminating discussion on symbolic execution challenges in practice up to 2013 [Cadar, 2013].

3 Relational symbolic execution

In order to apply symbolic execution to security properties such as noninterference, Milushev et al. [Milushev, 2012] propose a form of relational symbolic execution to use KLEE to analyze noninterference by means of a technique called self-composition [Barthe, 2004; Darvas, 2005; Terauchi, 2005] to reduce a relational property of a program p to a safety property of a transformation of p. More recently, Daniel et al. have optimized relational symbolic execution to be applicable to binary code to analyze relational properties such as constant time [Daniel, 2020] and speculative constant time [Daniel, 2021; Daniel, 2022] and discovered violations of these properties in real-world cryptographic libraries. All these approaches are based on pure (relational) symbolic execution static techniques and, as such, they are not capable of recovering soundness beyond a fixed bound as in our case. The closest work to RedSoundRSE is RelSym [Farina, 2019] which supports interactive refutation, as well as soundness. In order to recover soundness, Chong et al. [Farina, 2019] propose to use RelSym on manually annotated programs with loop invariants. Precision of refutation is guaranteed only if the invariants are strong enough, which cannot be determined by the tool itself. Precision is not guaranteed in any other cases. In contrast, our invariants are automatically generated via abstract interpretation and precision of refutation is always guaranteed up to a bound, which is automatically computed by our tool.

4 Sound static analyses for hyperproperties

As discussed in the introduction, many sound verification methods have been proposed for relational security properties. We refer the reader to an excellent survey on this topic [Sabelfeld, 2003] up to 2003. After 2003, several sound (semi-) static verification methods of noninterference-like properties have been proposed by means of type systems (e.g. [Banerjee, 2008; Fournet, 2011]), hybrid types, (e.g. [Santos, 2015]), relational logics (e.g. [Aguirre, 2017]), model checking (e.g. [Huisman, 2006; Backes, 2009]), and pure abstract interpretation [Assaf, 2017]. We expand on the ones based on abstract interpretation since they are the closest to our work. Giacobazzi and Mastroeni [Giacobazzi, 2004] define abstractions for attacker's views of program secrets and design sound automatic program analyses based on abstract interpretation for sets of executions (in contrast to relational executions). Assaf et al. [Assaf, 2017] are the first to express hyperproperties entirely within the framework of abstract interpretation by defining a Galois connection that directly approximates the hyperproperty of interest. We utilize the abstract domain of Assaf et al. [Assaf, 2017] combined with symbolic execution to obtain RedSoundSE. Notice that because the framework of Assaf et al. [Assaf, 2017] relies on incomplete abstraction,

their analysis is not capable of precise refutation nor provide refutations models. To the best of our knowledge, no previous work has combined abstract domains and symbolic execution to achieve soundness.

Part II

Static Symbolic Tainting Analysis

Chapter 9

Towards static symbolic tainting analysis

Taint-tracking is a practice that is widely used for detecting vulnerabilities. It works by tagging values and then checking whether they reach sensitive parts of the program. In this part, the challenge is designing a *complete symbolic taint-tracker* that can eliminate false alarms generated by other tools that are not so precise, and also find possible input values when the alarm is real.

1 Taint-checks versus taint-tracking

Taint-checks were first introduced in Perl in the year 1989. The Perl programming language is a scripting language aimed to executing shell commands for different tasks as system administration, which can lead to security vulnerabilities. For this reason, Perl introduced *taint-checking* (also called *taint-tracking*) in 1989 with the goal of dynamically catching possibly insecure instructions. In principle, all external information should not affect other external information. Then, the assumption is that any arguments passed to the script are possibly dangerous, and any instructions that can produce an effect on a system are inclined to be attacked. While in the implementation in Perl might be quite simple, this technique can help tackle common security issues.

Dynamic taint-tracking in Perl is extremely useful, as it can prevent countless security issues. However, Perl is mostly a scripting language, where programs are very specific. What happens when we want to ensure the security of a system composed by dozens of modules? While dynamic taint-tracking can help catch vulnerabilities on-the-fly, the system will end up failing. For that reason, we are interested in static taint-tracking, in an effort to spot the source of problems before they manifest, preventing system failures.

Performing dynamic taint-tracking is straightforward. The execution is performed over completely concrete values. However, for static taint-analysis this is not the case. Instead, values need to be abstracted, leading to several limitations.

A static taint-tracker of interest is Pysa [Meta, 2023]. This taint-tracker analyses Python code, and calculates a sound over-approximation of the program behavior via abstract interpretation. This analyzer is extremely fast, and, together with other taint-analyzers used at Meta, represents a big portion of the bugs found at their software [Meta, 2020]. In exchange for its speed, this tool

performs many over-approximations, resulting in false-alarms that have to be manually checked by engineers to determine if these are real bugs or not.

2 Symbolic taint-tracking of Python programs

Our model of taint-tracking is aimed at Python programs, where we consider sources of taints and sinks are function calls. Then, a function call might generate an observation that is not allowed. While in a real world scenario the sources can originate from different places, the formalization is simplified to reflect the meaningful mechanics that go into place for taint-tracking. We consider, like in the real world, that when a taint reaches a sink the taint-tracker raises an alarm.

In Program 1.2c, an illegal flow happens when the value of `y` is the positive root of the polynomial. This program is unsafe, but a sound taint-analyzer such as Pysa cannot assert that. This kind of limitation forces an engineer to go over the alarm and determine it is real, which is easy for the example. However, in a real system, this is not trivial. By using symbolic execution and collecting the path constraints, an SMT solver can easily determine the actual value of `y` that triggers the flow.

3 Outline

In this part, our objective is to introduce taints formally and finally design a complete symbolic taint-tracker that acts as a second-stage analysis after Pysa. Chapter 10 introduces taints and a concrete taint-tracker that we use as a reference throughout the part. Chapter 11 presents the property we focus on: *weak secrecy*. Given a set of rules that establishes which flows are illegal, this property stipulates that a program must not perform any illegal flows. In Chapter 12, we define a symbolic tainting semantics that abstract those of Chapter 10. We also provide a completeness theorem that ensures that the abstract traces have a concrete counter-part. Chapter 13 introduces a combined analysis between a sound taint-tracker and our symbolic taint-tracker. In Chapter 14, we instantiate the sound taint-tracker with Pysa, an open-source tain-tracker developed by Meta, and we instantiate the symbolic taint-tracker with our own tool Pysta. In Chapter 15, we evaluate the instantiated combined analysis in a set of examples extracted from different sources, and we answer three research questions. Finally, in Chapter 16 we present related work.

Chapter 10

Taint-tracking

In this chapter, we first present a language fit for our purposes of taint-tracking, we give a formal definition of taints, and present a concrete taint-tracker that will be the baseline for the rest of the part.

In Figure 10.1, we introduce a fictitious, unsafe-by-design, program where a user inputs its username and password to login to a system. This fictitious program has two vulnerabilities which can be found by performing taint-tracking.

We first formalize taints by defining what is a source and a sink, and how these relate through rules. A new language syntax will allow us to annotate programs to have sources and sinks handled through functions. These functions have input streams, allowing us to simulate non-determinism, and external inputs, important aspects of real-life executions. Finally, we present a concrete taint-tracker that gives insight on how illegal flows happen.

1 Unsafe program

Program from Figure 10.1 is unsafe, suffering from two vulnerabilities. The first vulnerability is an SQL injection, since string formatting in Python does ensure the proper delimitation of special characters. Then, the input stored in `user` is potentially malicious, and could lead to tampering of the database. The second vulnerability is inspired in Log4Shell (CVE-2021-44228). Similarly to vulnerability number one, we cannot ensure that input is not maliciously crafted, and we assume that function `log` can be attacked as in Log4Shell, leading to remote code execution.

2 Taints and values

In taint-tracking, the interest lies in tracking sensitive information through the program execution to ensure that it does not reach a specific vulnerable statement. What makes data sensitive is up to the user: it could be confidential information, or a value that might eventually trigger a run-time error, etcetera.

In our setup, *taint tags*, or *sources*, are generated by functions, and accompany values

```
1 user = input()
2 password = input()
3 stmt = f"SELECT␣password␣\
4 FROM␣users␣WHERE␣username␣=␣'{user}'"
5 row = db.query(stmt)
6 if row == password:
7   log(f"user␣{user}␣logged␣in")
```

Figure 10.1: Unsafe program

throughout the execution. There can be multiple sources, which are unique, and user-defined. For instance, function `input` might generate source $\mathsf{input}^{\downarrow}$, and so on. We use a down-arrow notation to denote sources, such as $t_0^{\downarrow}$, where the set of all sources is denoted by $\mathfrak{T}^{\downarrow}$. To denote a subset of $\mathfrak{T}^{\downarrow}$, we use notation $T^{\downarrow}$. We consider that the only concrete values that are handled are integers $\mathbb{V} = \mathbb{Z}$. Then, taint-values are defined as follows.

Definition 18. *A* taint-value *is a pair* $(i, T^{\downarrow}) \in \mathbb{Z} \times \mathcal{P}(\mathfrak{T}^{\downarrow})$. *We denote with* $\mathbb{VT}$ *the set of all taint-values.*

Sinks, are the counterparts of sources, and represent sensitive locations within a program where specific values may induce unwanted behaviors, ranging from system crashes to the leak of confidential information. In taint-tracking, this concept is represented by source tags, accompanying values, reaching pertinent sinks. Similarly to sources, these are unique and user-defined, but they are not carried in values. To denote them, we use an up-arrow notation to denote sinks $t_{\uparrow}$, with the set of all sinks denoted by $\mathfrak{T}_{\uparrow}$.

To define the connection between each source and sink, we define *observations* and *rules*.

Definition 19. *An* observation*, denoted by* o*, is a 3-tuple* $(T^{\downarrow}, v, t_{\uparrow}) \in \mathcal{P}(\mathfrak{T}^{\downarrow}) \times \mathbb{V} \times \mathfrak{T}_{\uparrow}$ *formed by a set of sources, a value, and a sink.*

Definition 20. *A* rule r *is a pair of* $(t^{\downarrow}, t_{\uparrow}) \in \mathfrak{T}^{\downarrow} \times \mathfrak{T}_{\uparrow}$ *that indicates an unwanted behavior.* Rules*, denoted as* $\mathfrak{R} \in \mathcal{P}(\mathfrak{T}^{\downarrow} \times \mathfrak{T}_{\uparrow})$*, are a relation between sources and sinks.*

In some cases the value carried in observations can be safely ignored. In that case, we will not write it down as part of the observations, such as $o = (t^{\downarrow}, t_{\uparrow})$. Observations are events that are observable by the attacker. These are generated during execution when a sink is reached. If the observation triggers a rule, we say it is a *relevant observation*.

Example 18. *Let* $\mathfrak{T}^{\downarrow} = \{\mathsf{input}^{\downarrow}; \mathsf{sensitive}^{\downarrow}\}$ *and* $\mathfrak{T}_{\uparrow} = \{\mathsf{log}_{\uparrow}; \mathsf{sql}_{\uparrow}\}$. *A possible set of rules is*

$$\mathfrak{R} = \{(\mathsf{input}^{\downarrow}, \mathsf{log}_{\uparrow}); (\mathsf{input}^{\downarrow}, \mathsf{sql}_{\uparrow}); (\mathsf{sensitive}^{\downarrow}, \mathsf{log}_{\uparrow}); \}$$

meaning that when a value coming from function `input` *reaches functions* `log` *or* `db.query` *an illegal behavior possibly exists. Likewise, outputs of* `db.query` *might be sensitive, and should not reach function* `log`.

$$
\begin{aligned}
\mathbf{e} &::= v \mid \mathtt{x}[i] \mid \mathbf{e} \oplus \mathbf{e} \\
\mathbf{b} &::= \mathbf{tt} \mid \mathbf{ff} \mid \mathbf{e} \oslash \mathbf{e} \\
\\
\mathbf{s} &::= \mathtt{skip} \\
&\quad \mid \mathtt{x}[i] = \mathbf{e} \\
&\quad \mid \mathtt{x} = \mathtt{y} \\
&\quad \mid \mathtt{x} = \mathtt{g}(\mathbf{e}_0, \ldots, \mathbf{e}_n) \\
&\quad \mid \mathtt{g}(\mathtt{x}) \\
&\quad \mid \mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}\ \mathtt{else}\ \mathbf{c} \\
&\quad \mid \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c} \\
&\quad \mid \mathtt{return}\ [\mathbf{e}; \ldots; \mathbf{e}] \\
\\
\mathbf{c} &::= \mathbf{s} \mid \mathbf{s}; \mathbf{c} \\
\\
\mathbf{f} &::= \mathtt{def}\ \mathtt{g}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathbf{c}; \mathtt{return}\ [\mathbf{e}_1; \ldots; \mathbf{e}_m] \\
&\quad \mid \mathtt{def\ input\ g}(\ ) : T^{\downarrow} \\
&\quad \mid \mathtt{def\ sink\ g}(\mathtt{x} : t_{\uparrow}) : \emptyset \\
\\
\mathbf{p} &::= \mathbf{f}; \mathbf{p} \mid \mathbf{c}
\end{aligned}
$$

Figure 10.2: Syntax of taint-language.

3 Language syntax

In order to perform taint-tracking, we require an extended language syntax that can describe taints, that is, sources and sinks. These aspects of the language are all expressed through functions: sources and sinks are embedded in functions.

Another particularity of the language is that variables are mapped to arrays of values, and functions always return non-empty arrays: this is very clearly reflected in expressions, where variables are indexed. This decision allows us to have functions return several values at a time, allowing us to study more complex programs.

The full syntax of this language, called *taint-language*, is presented in Figure 10.2.

3.1 Expressions

Let the set of variables be noted as $\mathbb{X}$, and binary operators $\oplus$ and $\oslash$. As noted before, in this language variables are going to be mapped to an array of values. For this reason, an expression $\mathbf{e} \in \mathbf{E}$ is either a value $v \in \mathbb{V}$, a binary operation between expressions, or a *subscript* $\mathtt{x}[i]$ which takes the i^{th} element of $\mathtt{x} \in \mathbb{X}$. Variable names are not treated as expressions since they are mapped to arrays.

Boolean expressions $\mathbf{b} \in \mathbf{B}$ are either true, false or a binary operation between expressions.

3.2 Statements, functions and programs

In Figure 10.2, we present the syntax of statements, commands, functions, and programs. Statement **s** can be a skip, an `if` statement, a `while` statement, a `return` statement, a call to a procedure, or forcibly one of three types of assignments: the first case is the assignment to an index of a variable; the second assignment allows to copy a variable to another one; the last assignment is a function call. Return statements are used to return a sequence of values from a function call. A sequence of statements is called a command, denoted by $\mathbf{c} \in \mathbf{C}$.

Symbol **f** denotes function definitions, which can be defined in three different ways: *executable functions* are composed by a name, a series of arguments, and a command that represents the body of the function that always finishes with a `return` statement. These are functions that are analyzed line-by-line by the taint-tracker, as opposed to the other types of functions. The other types of functions are *input-functions* and *sink-functions*.

1. Input-functions return an array of values that might be tainted with a source.

$$\texttt{def input } \mathbf{g}(\,) : T^{\downarrow}$$

   This function takes no argument. The sources $T^{\downarrow}$ are optional, meaning that we can model some function as **rand** (returns a random integer) where no taint is generated, or a function such as **input** where a taint is expected.

2. Sink-functions take a single value that is sunk, and return no value, denoted by $\emptyset$.

$$\texttt{def sink } \mathbf{g}(\mathtt{x} : t_{\uparrow}) : \emptyset$$

3. Executable functions take n arguments, and return an array of elements.

$$\texttt{def } \mathbf{g}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathbf{c}; \texttt{return } [\ldots]$$

   Since function calls are part of **c**, executable functions can have calls to input-functions and sink-functions, which is useful to write more interesting programs.

Example 19. *Since sink-functions cannot return values, we can define functions with sinks inside when needed. This is the case for function* **db.query**, *that sinks the argument, while still returning a tainted value.*

Let us first define an input function, and a sink function.

$$\texttt{def input } \mathbf{sensitive}(\,) : \{\mathsf{sensitive}^{\downarrow}\}$$
$$\texttt{def sink } \mathbf{sql}(\mathtt{x} : \mathsf{sql}_{\uparrow}) : \emptyset$$

While in real-life the semantics of **db.query** *are important, from the point of view of the*

taint-tracker they do not matter.

```
def db.query(x) = {
        sql(x)
        y = sensitive()
        return y
}
```

The last functions that we need to define are **input** *and* **log**.

$$\texttt{def input input}(\ ) : \{\mathsf{input}^{\downarrow}\}$$
$$\texttt{def sink log}(\mathtt{x} : \mathsf{log}_{\uparrow}) : \emptyset$$

Finally, programs are denoted by **p**: these are a sequence of function definitions, followed by a command **c**.

4 Semantics

The *concrete taint semantics* for the taint-language is described in this section. These perform a precise execution of programs but use taint-values instead of simple values.

Two main characteristics of this semantics is:

- sources and sinks are always embedded in functions;
- the usage of input channels—or input channels—to simulate external nondeterminism of functions—such as user inputs.

4.1 Concrete store

Concrete stores are mappings from variables in $\mathbb{X}$ to $\mathbb{VT}^n$, denoted by $\mu \in \mathbb{MT}$. We use $\mathtt{x} \mapsto [(v_0, T_0^{\downarrow}), \ldots, (v_n, T_n^{\downarrow})]$ to explicitly enumerate the content of a store, where $\mathtt{x}$ is mapped to an array of concrete values and sources.

4.2 Evaluation of expressions

$$\frac{}{\mu \vdash n \downarrow^{\eta} (n, \eta)} \qquad \frac{\mu \vdash \mathbf{e}_0 \downarrow^{\eta} (v_0, T_0^{\downarrow}) \qquad \mu \vdash \mathbf{e}_1 \downarrow^{\eta} (v_1, T_1^{\downarrow})}{\mu \vdash \mathbf{e}_0 \oplus \mathbf{e}_1 \downarrow^{\eta} (v_0 \oplus v_1, T_0^{\downarrow} \cup T_1^{\downarrow})}$$

$$\frac{\mu(\mathtt{x})[i] = (v, T^{\downarrow})}{\mu \vdash \mathtt{x}[i] \downarrow^{\eta} (v, \eta \cup T^{\downarrow})} \qquad \frac{\mu \vdash \mathbf{e} \downarrow^{\eta} (v, T_0^{\downarrow}) \qquad o = (T_0^{\downarrow}, v, T_1^{\downarrow})}{\mu \vdash \underline{\mathbf{e}}_{T_1^{\downarrow}} \downarrow_o^{\eta} (v, T_0^{\downarrow})}$$

Figure 10.3: Concrete evaluation of expressions

We define the concrete evaluation of expressions $\downarrow$ in big-step style in Figure 10.3. We denote $\mu \vdash \mathbf{e} \downarrow^{\eta} (v, T^{\downarrow})$ the evaluation of $\mathbf{e}$ with store μ and a *taint-context* η, reaching a taint-value

$(v, T^{\downarrow})$. While taint-tracking specifically focuses on explicit flows of information, by adding a taint-context, it is possible to use the same semantics to consider both types of flows.

Evaluation of expressions **e** and **b** is straightforward except for a special type of expressions that we call *taint-expressions*. Evaluating expression $\underline{\mathbf{e}}_{t_\uparrow}$ evaluates **e** normally, while also generating an observation o that has the sink $t_\uparrow$.

Example 20. *Let us assume* $\mu = [\mathtt{x} \mapsto (0, \{\mathsf{input}^{\downarrow}\})]$. *Let us assume that the definition of* `log` *is the following:*

$$\texttt{def sink log}(\texttt{obj} : \mathsf{log}_\uparrow) : \emptyset$$

Then, the application of `log(x)`*, results in the evaluation of expression:*

$$\mu \vdash \underline{\mathtt{x}}_{\mathsf{log}_\uparrow} \downarrow_o (0, \{\mathsf{input}^{\downarrow}\}) \quad \textit{where} \quad o = (\{\mathsf{input}^{\downarrow}\}, 0, \mathsf{log}_\uparrow)$$

4.3 Input channels

An *input channel* is a stream of inputs that can be used by functions to describe external non-determinism. As we will observe in the definition of the semantics, these input values are only used in source-functions. Each source-function name $f \in \mathcal{F}_{name}$ is associated to its own input channel.

Definition 21. *An* input pointer p *is a mapping from a function name to a non-negative integer. A* channel input queue q *is a map from non-negative integers to input values in* $\mathbb{V}$. *A* program input $\mathcal{I}$ *is a mapping from function names* f *to a channel input queue. Function* $\mathsf{read} : \mathcal{I} \to \mathbb{V}$ *is defined as follows*

$$\text{READ}\ \frac{\mathcal{I}(i) = q \qquad p(i) = n \qquad q(n) = v \qquad p' = p[i \mapsto p(i) + 1]}{\mathsf{read}(\mathcal{I}, i, p) = v, p'}$$

The set of all input channels is C_{in}. The symbol p_0 denotes an initial input pointer mapping every input channel to position 0. We assume that there is an input channel for each function name.

Example 21 (Input channels)**.** *For instance, let us define functions* `input`*,* `get_user_creds` *and* `log`*. We assume that the first two functions take no arguments, and return a "user input" and the credentials of a user, respectively. Function* `log` *takes an argument called* `obj` *and returns 0. Then, we can define them as follows.*

$$\texttt{def input input}(\ ) : \mathsf{input}^{\downarrow}$$

For `input`*, we look at two possible input channels for two different executions. Let*

$$\mathcal{I}_1(\texttt{input}) = [1, 2, 3, 4, \dots] \qquad \mathcal{I}_2(\texttt{input}) = [5, 4, 3, 2, \dots]$$

where $\mathcal{I}_1$ *and* $\mathcal{I}_2$ *are input channels, corresponding to execution 1 and execution 2, respectively.*

In execution 1, the first call of `input` *returns* $(1, \{\mathsf{input}^{\downarrow}\})$*, then* $(2, \{\mathsf{input}^{\downarrow}\})$*, and so on. However, in execution 2, a call to* `input` *returns* $(5, \{\mathsf{input}^{\downarrow}\})$ *first.*

4.4 Function definition

Based on the language syntax, programs $\mathbf{p}$ are defined as a sequence of function definitions followed by a command.

$$\mathbf{p} = \mathbf{f}_1; \ldots; \mathbf{f}_k; \mathbf{c}$$

Functions, while part of the syntax, are not defined in the semantics, but they are considered as part of the environment. To do so, we define the set of functions $\mathcal{F}$ as follows.

We assume the first n functions are executable functions—functions that have a body—then there is m input-functions, and lastly k sink-functions. Then $\mathbf{f}_i$ is either of the three:

$$\begin{aligned} \mathbf{f}_i &= \texttt{def}\ \mathbf{g}_i(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathbf{c} \quad \vee \\ \mathbf{f}_i &= \texttt{def input}\ \mathbf{g}_i(\,) : T^{\downarrow} \quad \vee \\ \mathbf{f}_i &= \texttt{def sink}\ \mathbf{g}_i(\mathtt{x}_1 : t_{\uparrow}) : \emptyset \end{aligned}$$

The set of executable functions is defined as follows.

$$\mathcal{F}_{\mathrm{e}} = \bigcup_{i=1}^{n} (\mathbf{g}_i, (\mathtt{x}_1, \ldots, \mathtt{x}_n), (\lambda(\mathtt{x}_1, \ldots, \mathtt{x}_n).\mathbf{c}))$$

The body of the program becomes a lambda-term, where the actual values can be injected.

The set of source and sink-functions is defined as follows.

$$\mathcal{F}^{\downarrow} = \bigcup_{i=n+1}^{m+n} (\mathbf{g}_i, T^{\downarrow}) \qquad \mathcal{F}_{\uparrow} = \bigcup_{i=n+k+1}^{n+m+k} (\mathbf{g}_i, t_{\uparrow})$$

Finally, the set of all functions $\mathcal{F}$ is defined as the union of these sets.

$$\mathcal{F} = \mathcal{F}_{\mathrm{e}} \cup \mathcal{F}^{\downarrow} \cup \mathcal{F}_{\uparrow}$$

4.5 Flow functions

As said previously, the semantics makes use of a taint-context η that keeps track of the implicit flows. In order to have a concise definition, we define a family of functions, called *flow-functions*.

A flow-function $\mathsf{fl} : (\mathcal{P}(\mathfrak{T}^{\downarrow}) \times \mathcal{P}(\mathfrak{T}^{\downarrow})) \to \mathcal{P}(\mathfrak{T}^{\downarrow})$ takes two sets of sources, and returns a set of sources. The first argument is the context η, and the second set is the new incoming flow context. Its output is the new taint-context after adding the incoming flow context.

We provide two different flow-functions.

Definition 22 (Implicit flow-function)**.** *The implicit flow-function is defined as follows*

$$\mathit{fl}_i(\eta, t) = \eta \cup t.$$

$$\textsc{assign-sub}\ \frac{\mu \vdash \mathbf{e} \downarrow^{\eta} v}{\mathcal{I}, \mathcal{F} \vdash (\mathtt{x}[i]^{\eta} = \mathbf{e}, \mu, p) \to (\mathtt{skip}, \mu[\mathtt{x}[i] \mapsto v], p)}$$

$$\textsc{assign-var}\ \frac{}{\mathcal{I}, \mathcal{F} \vdash (\mathtt{x}^{\eta} = \mathtt{y}, \mu, p) \to (\mathtt{skip}, \mu[\mathtt{x} \mapsto \mu(\mathtt{y})], p)}$$

$$\textsc{assign-fun}\ \frac{\begin{array}{c}(\mathbf{g}, (\mathtt{x}_1, \ldots, \mathtt{x}_n), \mathbf{c}) \in \mathcal{F} \qquad \mu \vdash \mathbf{e}_i \downarrow^{\eta} v_i \text{ for } 1 \le i \le n \\ \mathcal{I}, \mathcal{F} \vdash (\mathbf{c}(v_1, \ldots, v_n), \mu, p) \xrightarrow{o}{}^{*} (\mathtt{return}\ [\mathbf{e}'_1, \ldots, \mathbf{e}'_k], \mu', p') \\ \mu' \vdash \mathbf{e}'_i \downarrow^{\eta} v'_i \text{ for } 1 \le i \le k\end{array}}{\mathcal{I}, \mathcal{F} \vdash (\mathtt{x}^{\eta} = \mathtt{g}(\mathbf{e}_1, \ldots, \mathbf{e}_n), \mu, p) \xrightarrow{o} (\mathtt{skip}, \mu[\mathtt{x} \mapsto (v'_1, \ldots, v'_k)], p')}$$

$$\textsc{fun-input}\ \frac{(\mathbf{g}, t^{\downarrow}) \in \mathcal{F} \qquad \mathsf{read}(\mathcal{I}, \mathbf{g}, p) = ((v_1, \ldots, v_n), p') \qquad v = ((v_1, t^{\downarrow} \cup \eta), \ldots, (v_n, t^{\downarrow} \cup \eta))}{\mathcal{I}, \mathcal{F} \vdash (\mathtt{x}^{\eta} = \mathtt{g}(\ ), \mu, p) \to (\mathtt{skip}, \mu[\mathtt{x} \mapsto v], p')}$$

$$\textsc{fun-sink}\ \frac{(\mathbf{g}, t_{\uparrow}) \in \mathcal{F} \qquad \mu \vdash \mathbf{e} \downarrow^{\eta} (v, t) \qquad o = (t, v, t_{\uparrow})}{\mathcal{I}, \mathcal{F} \vdash (\mathbf{g}(\mathbf{e})^{\eta}, \mu, p) \xrightarrow{o} (\mathtt{skip}, \mu, p)}$$

Figure 10.4: Set of step semantics rules for concrete taint-tracker.

Definition 23 (Explicit flow-function)**.** *The explicit flow-function is defined as follows*

$$\mathit{fl}_e(\eta, t) = \eta.$$

The semantics are parameterized by one of these two functions.

4.6 Step relation

A *state* is a 3-tuple $(\mathbf{c}, \mu, p)$ with a command $\mathbf{c}$, a store μ, and an input pointer p. In particular, a state of the form $(\mathtt{skip}, \ldots)$ is final. We write $\mathbb{MT}$ and $\mathbb{S}$ for the set of stores and states respectively. Let $(\to) \subseteq \mathcal{F}_{name} \times C_{in} \times \mathbb{S} \times \mathbb{O}^* \times \mathbb{S}$ denote the small step operational semantics.

In Figure 10.4, we define the three assignment rules, and the function definition rule. Rule ASSIGN-SUB, evaluates an expression $\mathbf{e}$ and assigns it to an index of the variable. Rule ASSIGN-VAR, assigns the complete value of a variable to another one.

Rule ASSIGN-FUN evaluates a function call. To do so, first we look up for function $\mathtt{g} \in \mathcal{F}$, meaning that the function has been defined previously. Then, all n arguments are evaluated. All of these values can then be applied to the body of the function which is a lambda term. After a number of execution steps, the command of the state is $\mathtt{return}\ [\ldots]$. Notice that these execution steps might generate an observation. The returned expressions are evaluated and assigned, and if any observation was generated, it is annotated.

Rule FUN-INPUT returns a tainted value. The value comes from an input channel, and

Rule FUN-SINK sinks a single argument. This implies that an observation is generated. When the argument has a taint, and that taint and sink are a rule, this observation is relevant. Through this mechanism the semantics detects illegal flows.

$$\text{IF-T}\ \frac{(\mathbf{b},\mu)\downarrow^{\eta}(\mathbf{tt},t)}{\mathcal{I},\mathcal{F}\vdash(\texttt{if}^{\eta}\ \mathbf{b}\ \texttt{then}\ \mathbf{c}_0\ \texttt{else}\ \mathbf{c}_1,\mu,p)\to(\mathbf{c}_0^{\mathsf{fl}(\eta,t)},\mu,p)}$$

$$\text{IF-F}\ \frac{(\mathbf{b},\mu)\downarrow^{\eta}(\mathbf{ff},t)}{\mathcal{I},\mathcal{F}\vdash(\texttt{if}^{\eta}\ \mathbf{b}\ \texttt{then}\ \mathbf{c}_0\ \texttt{else}\ \mathbf{c}_1,\mu,p)\to(\mathbf{c}_1^{\mathsf{fl}(\eta,t)},\mu,p)}$$

$$\text{SEQ-1}\ \frac{}{\mathcal{I},\mathcal{F}\vdash((\texttt{skip}^{\eta};\mathbf{c}_1^{\eta_1}),\mu,p)\to(\mathbf{c}_1^{\eta_1},\mu,p)}$$

$$\text{SEQ-2}\ \frac{(\mathbf{c}_0^{\eta_0},\mu,p)\xrightarrow{o}(\mathbf{c}_{0'}^{\eta_0'},\mu',p')}{\mathcal{I},\mathcal{F}\vdash((\mathbf{c}_0^{\eta_0};\mathbf{c}_1^{\eta_1}),\mu,p)\xrightarrow{o}((\mathbf{c}_{0'}^{\eta_0'};\mathbf{c}_1^{\eta_1}),\mu',p')}$$

$$\text{WHILE-T}\ \frac{(\mathbf{b},\mu,\eta)\downarrow^{\eta}(\mathbf{tt},t)}{\mathcal{I},\mathcal{F}\vdash(\texttt{while}^{\eta}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0,\mu,p)\to((\mathbf{c}_0^{\mathsf{fl}(\eta,t)};\texttt{while}^{\mathsf{fl}(\eta,t)}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0),\mu,p)}$$

$$\text{WHILE-F}\ \frac{(\mathbf{b},\mu,\eta)\downarrow^{\eta}(\mathbf{ff},t)}{\mathcal{I},\mathcal{F}\vdash(\texttt{while}^{\eta}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0,\mu,p)\to(\texttt{skip},\mu,p)}$$

Figure 10.5: Set of step semantics rules for concrete taint-tracker.

In rules IF-T and IF-F, either $\mathbf{c}_0$ or $\mathbf{c}_1$ are chosen based on the value of the guard. The context is updated by fl with the taint calculated from the guard. The same logic applies for WHILE-T and WHILE-F.

Example 22. *In this example we execute Program 10.1 with the taint-tracker semantics. We denote the program with* $\mathbf{c}$*, and we assume all functions used are defined in set* $\mathcal{F}$*. We specify the input channels for input-functions as in Example 21.*

$$\begin{aligned}\mathcal{I}(\texttt{input}) &= [1;2;\dots]\\ \mathcal{I}(\texttt{sensitive}) &= [2;4;\dots]\end{aligned}$$

When we write $\mathbf{c}_i$*, that represents the command from "line* i *to the end of the program". We are going to display the execution of the program line-by-line, writing on the left side "Line* i*", meaning that the command on the right is yet to be executed. Starting from the initial*

configuration is $(\boldsymbol{c}, [\,], p_0)$*, let us follow the execution of the program.*

Line 1: $((\texttt{user} = \textbf{input}(\,);\, \boldsymbol{c}_2), [\,], p_0)$

$\downarrow$ FUN-INPUT

Line 2: $((\texttt{password} = \textbf{input}(\,);\, \boldsymbol{c}_3), [\texttt{user} \mapsto (1, \{\mathsf{input}^{\downarrow}\})], p_1)$

$\downarrow$ FUN-INPUT

Line 3: $((\texttt{stmt} = \texttt{user};\, \boldsymbol{c}_4), [\texttt{password} \mapsto (2, \{\mathsf{input}^{\downarrow}\}); \texttt{username} \mapsto (1, \{\mathsf{input}^{\downarrow}\})], p_2)$

$\downarrow$ ASSIGN-VAR

Line 5: $((\texttt{row} = \textbf{db.query}(\texttt{stmt});\, \boldsymbol{c}_5), [\texttt{stmt} \mapsto (1, \{\mathsf{input}^{\downarrow}\}); \ldots], p_2)$

$\downarrow$ ASSIGN-FUN, $o_1 = (\{\mathsf{input}^{\downarrow}\}, 1, \mathsf{sql}_{\uparrow})$

Line 6: $((\texttt{if row} = \texttt{password then log}(\texttt{user}) \texttt{ else skip}), [\texttt{row} \mapsto (2, \{\mathsf{sensitive}^{\downarrow}\}); \ldots], p_3)$

First and second step apply rule FUN-INPUT*, assigning value* 1 *and* 2 *from the input channel to variables* `user` *and* `password`*, respectively. Based on the definition of* **input***, the value also carries a taint* $\{\mathsf{input}^{\downarrow}\}$*.*

For the third step, we assume that the string formatting is equal to an assignment between variables. Notice that line 3 takes two lines of code, so the next configuration holds line 5.

The fourth step applies rule ASSIGN-FUN*. In this step, function* **db.query***, based on the definition of Example 19, performs a sink in its body, and returns a new value from function* **sensitive***. For that reason, observation* o_1 *is generated, and* `row` *maps to 2 with a taint* $\{\mathsf{sensitive}^{\downarrow}\}$*.*

Next, we inspect the last steps.

Line 6: $((\texttt{if row} = \texttt{password then log}(\texttt{user}) \texttt{ else skip}), [\texttt{row} \mapsto (2, \{\mathsf{sensitive}^{\downarrow}\}); \ldots], p_3)$

$\downarrow$ IF-T

Line 7: $(\texttt{log}(\texttt{user}), [\texttt{user} \mapsto (1, \{\mathsf{input}^{\downarrow}\}); \ldots], p_3)$

$\downarrow$ FUN-SINK, $o_2 = (\{\mathsf{input}^{\downarrow}\}, 1, \mathsf{log}_{\uparrow})$

Line 8: $(\texttt{skip}, [\texttt{user} \mapsto (1, \{\mathsf{input}^{\downarrow}\}); \ldots], p_3)$

Condition `row` = `password` *holds, since both are equal to 2. Then, rule* IF-T *can be applied, and the body of the* `if` *gets chosen to execute. Lastly, the call* `log` *generates observation* o_2 *that, as observation* o_1*, is relevant.*

Finally, the program generated two relevant observations, leading us to determine that it is unsafe with respect to rules $\mathfrak{R}$*.*

Chapter 11

Observations and Weak Secrecy

Given the definition of the concrete taint-tracker, we now require a property of interest that gives a meaning to these concrete traces with their respective observations. This property is called *weak secrecy* [Volpano, 1999], and it stipulates which explicit flows are allowed in a program execution given a pre-defined set of rules.

In order to introduce these properties, we first present a series of definitions that allow us to compare traces based on the observations generated in each of them.

To gain intuition about these different properties, in Figure 11.1 we introduce five programs that illustrate weak secrecy. We will reason about these programs to determine if they satisfy or not the property.

1 Relevant-equal observations

To prove anything about the semantics, we focus on the events generated during the execution—the *observations*, specifically, the *relevant* ones. An observation is relevant when the source and sink connecting are a rule in the rules relation $\mathfrak{R}$.

Definition 24 (Relevant observation)**.** *Let an observation* $(\{t_1^{\downarrow}, \ldots, t_n^{\downarrow}\}, v, t_{\uparrow}) \in \mathbb{O}$*. We say* $(\{t_1^{\downarrow}, \ldots, t_n^{\downarrow}\}, v, t_{\uparrow})$ *is* relevant*, denoted* $(\{t_1^{\downarrow}, \ldots, t_n^{\downarrow}\}, v, t_{\uparrow}) \in \mathfrak{R}$*, if and only if*

$$\exists\, i : 1 \leq i \leq n \ \wedge\ (t_i^{\downarrow}, t_{\uparrow}) \in \mathfrak{R}.$$

Since traces generate a sequence of observations, we are interested in filtering such sequences to only keep relevant observations. Non-relevant observations can be safely removed.

Definition 25 (Relevant filtering)**.** *Let* $o = [o_1;\ \ldots;\ o_n]$ *be a sequence of observations. Observations* $o' = [o_{i_1};\ \ldots;\ o_{i_m}]$*, with* $1 \leq i_1 < \cdots < i_m \leq n$*, are a* relevant filtering *of* o*, written* $o/\mathfrak{R}$*, if and only if* o' *is the result of removing every* $o_j \in o$ *such that* $o_i \notin \mathfrak{R}$ *while not altering the order of the elements.*

Now that sequences of observations can be filtered, we need to compare them to establish a relationship between traces. To do so we define the *relevant-equal observations* relation.

```
1  x = f1()
2  g = g1(x)
```
(a)

```
1  x = f1()
2  if b:
3      if not b:
4          y = g1(x)
```
(b)

```
1  r = r1()
2  x = f1()
3  if x > 0:
4      y = g1(r)
```
(c)

```
1  x = f1()
2  y = mult(0,x)
3  z = g1(y)
```
(d)

Figure 11.1: Abstract programs to provide intuition about weak secrecy. In these programs, we assume that functions that are called $\mathtt{f}_i$ return a value with a source $\mathsf{f}_i^{\downarrow}$. Functions called $\mathtt{g}_i$ have sinks $\mathsf{g}_{\uparrow}^i$. Pairs $(\mathsf{f}_i^{\downarrow}, \mathsf{g}_{\uparrow}^i)$ are part of the set of rules $\mathfrak{R}$.

Definition 26 (Relevant-equal observations)**.** *We define the relevant equal observations as the equality of relevant filterings.*

Let $o_1, o_2 \in \mathbb{O}^*$ be two sequences of observations. We say o_1 is relevant-equal *to o_2, denoted $o_1 \overset{\mathfrak{R}}{=} o_2$, if and only if,*

$$o_1/\mathfrak{R} = o_2/\mathfrak{R}$$

2 Weak secrecy

The property we present is *weak secrecy* [Volpano, 1999]. This property is of interest to us because it models closely taint-trackers, stating that no relevant observation should happen in an execution.

Definition 27 (Weak secrecy)**.** *Assuming the semantics are parameterized by the explicit flow-function fl_e. Given an initial memory μ_0, a set of functions $\mathcal{F}$, and an input channels $\mathcal{I}$. A program $\boldsymbol{c}$ satisfies* weak secrecy*, written $WS \vDash^{t^{\downarrow}} \boldsymbol{c}$, if and only if,*

$$\exists\ \mu_1, p_1 : \mathcal{I}, \mathcal{F} \vdash (\boldsymbol{c}, \mu_0, p_0)(\xrightarrow{o})^*(\mathtt{skip}, \mu_1, p_1) \implies o \overset{\mathfrak{R}}{=} [\,]$$

The nature of weak secrecy is that a program is secure if no relevant observations are generated.

Example 23. *Program 11.1a generates a relevant observation in every execution. Therefore, it is not weak secret.*

In Program 11.1b, the sink is unreachable, therefore, no observation is generated. Thus, the program is weak secret.

In Program 11.1c the sink is reachable when the value of `x` *is greater than 0, but the value passed to $\mathtt{g}_1$ is not tainted. Hence, the observation generated is $(\emptyset, r, \mathsf{g}_{\uparrow}^1)$, and it is not relevant. Then, the program is weak secret.*

Program 11.1d always generates relevant observations with value 0. Hence, it is not weak secret.

Chapter 12

Symbolic execution taint-tracker

The concrete taint-tracker presented in Chapter 10 acts as a dynamic taint-tracker, that can catch illegal flows on-the-fly (during an execution). However, finding an illegal flow on-the-fly leads to an abrupt termination, possibly causing the system to fail. Instead, in this chapter we aim at analyzing the program statically, allowing to detect bugs previous to the execution of the system. In some cases, it is also possible to prove the absence of bugs. To do so, we introduce our *symbolic taint-tracker semantics*, built to emulate the concrete taint-tracker. We also provide a completeness theorem, and a soundness up-to-a-bound theorem, proving that the symbolic taint-tracker abstracts the concrete taint-tracker precisely.

1 Concretization

To ensure the new taint-tracker abstracts the concrete one, we define the abstraction and its concretization, connecting concrete and symbolic stores.

1.1 Symbolic store and symbolic path

The counterpart of concrete values are *symbolic expressions* $\varepsilon \in \mathbb{E}$. These can either be an integer n, a numerical symbol $\boldsymbol{x}$, or a binary operation $\oplus$ between symbolic expressions.

Symbolic boolean expressions are denoted by $\beta \in \mathbb{B}$: a symbolic boolean expression is either true, false, a boolean symbol $\boldsymbol{b}$, the negation of a symbolic boolean expression, the comparison of two symbolic expressions, or a binary operation $\oslash$ between symbolic boolean expressions.

$$\begin{array}{rcl} \varepsilon & ::= & n \mid \boldsymbol{x} \mid \varepsilon \oplus \varepsilon \\ \beta & ::= & \mathbf{tt} \mid \mathbf{ff} \mid \boldsymbol{b} \mid \neg\beta \mid \varepsilon \oslash \varepsilon \mid \beta \oslash \beta \end{array}$$

Our symbolic execution, given that it abstracts the concrete semantics, abstracts taint-values by replacing concrete values for symbolic expressions.

$$\mathbb{ET} = \mathbb{E} \times \mathcal{P}(\mathfrak{T}^{\downarrow})$$

Symbolic taint store A *symbolic taint store* is a mapping from variables in $\mathbb{X}$ to $\mathbb{ET}^n$, and are denoted by $\rho \in \overline{\mathbb{MT}}$. We use $\mathtt{x} \mapsto [(\varepsilon_0, T_0), \ldots, (\varepsilon_n, T_n)]$ to explicitly enumerate the contents of a store, where $\mathtt{x}$ is mapped to a list of symbolic expressions and sources.

Symbolic taint path For defining constraints, we define the set of taint boolean expressions.

$$\mathbb{BT} = \mathbb{B} \times \mathcal{P}(\mathfrak{T}^{\downarrow})$$

Then, a *symbolic taint path* is a set of constraints denoted by $\pi \in \mathcal{P}(\mathbb{BT})$. The symbolic taint path collects information about the path taken by the trace, and constraints the symbolic map.

Symbolic precise store A *symbolic precise store* is a pair (ρ, π) of a symbolic taint store and a symbolic taint path, denoted by $\kappa \in \mathbb{KT}$.

1.2 Concretization

To concretize a symbolic precise store, we not only require a concrete store, but a *valuation* as well. We let a valuation be a function $\nu : \overline{\mathbb{V}} \longrightarrow \mathbb{V}$, mapping every symbol present in the symbolic store to a concrete value. By having a valuation, we can precisely characterize which concrete stores are a concretization of the symbolic store.

To exploit valuations, we define a substitution function. Given a valuation, the substitution function takes a symbolic expression and returns a concrete value.

Definition 28 (Symbolic substitution)**.** *Given a symbolic expression ε, we let $[\![\varepsilon]\!]$ be a partial function that maps a valuation ν to the value obtained by replacing each symbolic value $\boldsymbol{x}$ in ε with $\nu(\boldsymbol{x})$, defined as follows:*

$$\begin{aligned}
[\![n]\!](\nu) &= n \\
[\![\boldsymbol{x}]\!](\nu) &= \nu(\boldsymbol{x}) \\
[\![\varepsilon_0 \oplus \varepsilon]\!](\nu) &= [\![\varepsilon_0]\!](\nu) \oplus [\![\varepsilon_1]\!](\nu)
\end{aligned}$$

In the case of a taint symbolic expression, we define it in the following way.

$$[\![(\varepsilon, t)]\!](\nu) = ([\![\varepsilon]\!](\nu), t)$$

We extend this notation to symbolic maps and symbolic paths

$$\begin{aligned}
[\![\rho]\!](\nu) &= [\ \mathtt{x} \in \mathbb{D}(\rho) : \mathtt{x} \mapsto [\![\rho[\mathtt{x}]]\!](\nu)\ ] \\
[\![\pi]\!](\nu) &= \bigwedge_{i=1}^{m} [\![\phi_i]\!](\nu)
\end{aligned}$$

where $\mathbb{D}$ is the domain function, returning the set of variables that are mapped.

To define the concretization function, we require auxiliary functions π_1 and π_2, which are the left and right projection of a taint-value, respectively.

Definition 29 (Symbolic store concretization)**.** *The* symbolic store concretization, $\gamma_{\overline{\mathbb{MT}}} : \overline{\mathbb{MT}} \longrightarrow \mathcal{P}(\mathbb{MT} \times (\overline{\mathbb{V}} \to \mathbb{V}))$*, maps a symbolic store to the set of pairs made of a store and a valuation that realize it, i.e.,*

$$\gamma_{\overline{\mathbb{MT}}}(\rho) = \{(\mu, \nu) \mid \forall\, \mathtt{x} \in \mathbb{D}(\rho) :\ \pi_1(\mu(\mathtt{x})) = \pi_1([\![\rho(\mathtt{x})]\!](\nu)) \ \wedge\ \pi_2(\mu(\mathtt{x})) = \pi_2(\rho(\mathtt{x}))\}$$

A *symbolic precise store* is a pair $\kappa = (\rho, \pi)$ where $\rho \in \overline{\mathbb{MT}}$ and π is a symbolic path. We write $\mathbb{K}$ for the set of symbolic precise stores. Their meaning is defined as follows:

Definition 30 (Symbolic precise store concretization)**.** *The* symbolic precise store concretization*, $\gamma_{\mathbb{KT}} : \mathbb{KT} \longrightarrow \mathcal{P}(\mathbb{MT} \times (\overline{\mathbb{V}} \to \mathbb{V}))$, is defined by $\gamma_{\mathbb{KT}}(\rho, \pi) = \{(\mu, \nu) \in \gamma_{\overline{\mathbb{MT}}}(\rho) \mid [\![\pi]\!]\nu = \boldsymbol{tt}\}$.*

For example, the most general symbolic state $\top$ maps each variable to an unconstrained symbol. Its precise store concretization is an infinite set of pairs $(\mu, \nu) \in \gamma_{\mathbb{KT}}(\top)$ with all possible combinations, since there are no constraints.

```
1  user = input()
2  pwd = input()
3  if 0 == pwd**2 + pwd*2 - 8:
4      log(f"user␣{user}␣logged␣in")
```

Program 1.2d adapted to fit more closely with the context of taint-trackers.

Example 24. *We reintroduce Program 1.2d. If the password is the root of a polynomial (*$\mathtt{pwd} = 2$*), the user is logged. Assuming that the input value coming from* **input** *is 2, the execution is as follows, and there exists a symbolic precise store, and a valuation such that* $(\mu_0, \nu_0) \in \gamma_{\mathbb{KT}}(\kappa_0)$*:*

$$\mathcal{I}, \mathcal{F} \vdash (\boldsymbol{c}, [\,], p_0)(\xrightarrow{o_0})^*(\mathtt{skip}, \mu, p_0') \quad \textit{where} \quad \mu_0 = [\mathtt{pwd} \mapsto (2, \{\mathsf{input}^{\downarrow}\}); \mathtt{user} \mapsto (0, \{\mathsf{input}^{\downarrow}\})]$$
$$\textit{and} \quad o_0 = [(\mathsf{input}^{\downarrow}, \mathsf{log}_{\uparrow}); (\emptyset, \mathsf{log}_{\uparrow})]$$

$$\kappa_0 = ([\mathtt{pwd} \mapsto (\boldsymbol{p}, \{\mathsf{input}^{\downarrow}\}); \mathtt{user} \mapsto (\boldsymbol{u}, \{\mathsf{input}^{\downarrow}\})], \boldsymbol{p}^2 + 2\boldsymbol{p} = 8)$$

Notice that the constraints in κ_0 are such that any concretization of κ_0 maps the value of `pwd` *to 2. The observation o_0 is relevant with respect to rules $\mathfrak{R}$, implying that the program is not secure.*

Another possible scenario is that the password taken from **input** *is not 2. Then,*

$$\mathcal{I}, \mathcal{F} \vdash (\boldsymbol{c}, [\,], p_0)(\xrightarrow{o_0})^*(\mathtt{skip}, \mu_1, p_1') \quad \textit{where} \quad \mu_1 = [\mathtt{pwd} \mapsto (1, \{\mathsf{input}^{\downarrow}\}); \mathtt{user} \mapsto (0, \{\mathsf{input}^{\downarrow}\})]$$
$$\textit{and} \quad o_1 = [\,]$$

$$\kappa_1 = ([\mathtt{pwd} \mapsto (\boldsymbol{p}, \{\mathsf{input}^{\downarrow}\}); \mathtt{user} \mapsto (\boldsymbol{u}, \{\mathsf{input}^{\downarrow}\})], \boldsymbol{p}^2 + 2\boldsymbol{p} \neq 8)$$

This other execution does not generate any relevant observations.

2 Formal semantics for symbolic taint-tracker

The semantics presented in Chapter 10 might not terminate. Instead, the semantics presented in this section always terminates, and performs no over-approximation. This adaptation requires the addition of a counter to keep tracks of loop iterations.

2.1 Abstract input channels

The input channels from Chapter 10 are replaced by *abstract input channels*, denoted by $\mathcal{I}^\sharp$. These input channels only vary in that they contain symbols.

Example 25 (Abstract input channels)**.** *For function* `rand`*, a possible input channel is*

$$\mathcal{I}^\sharp(\texttt{rand}) = [r_0, r_1, \dots]$$

2.2 Evaluation of expressions

We define the evaluation of symbolic expressions as follows.

$$\frac{}{\rho \vdash n \Downarrow^\eta n} \qquad \frac{\rho \vdash \mathbf{e}_0 \Downarrow^\eta (\varepsilon_0, T_0^\downarrow) \qquad \rho \vdash \mathbf{e}_1 \Downarrow^\eta (\varepsilon_1, T_1^\downarrow)}{\rho \vdash \mathbf{e}_0 \oplus \mathbf{e}_1 \Downarrow^\eta (\varepsilon_0 \oplus \varepsilon_1, T_0^\downarrow \cup T_1^\downarrow)}$$

$$\frac{\rho(\mathtt{x})[i] = (\varepsilon, T^\downarrow)}{\rho \vdash \mathtt{x}[i] \Downarrow^\eta (\varepsilon, \eta \cup T^\downarrow)} \qquad \frac{\rho \vdash \mathbf{e} \Downarrow^\eta (\varepsilon, T_0^\downarrow) \qquad o = (T^\downarrow, t_\uparrow)}{\rho \vdash \underline{\mathbf{e}}_{t_\uparrow} \Downarrow_o^\eta (\varepsilon, T_0^\downarrow)}$$

This is very similar to the concrete evaluation of expressions, but the output are symbolic expressions that keep the relation between the different symbols. For instance, given $\rho = [\mathtt{x} \mapsto (\boldsymbol{x}, \emptyset); \mathtt{y} \mapsto (\boldsymbol{y}, \emptyset)]$, the evaluation of $\mathtt{x} + \mathtt{y}$ returns $(\boldsymbol{x} + \boldsymbol{y}, \emptyset)$. Then, the concretization function provides a valuation that specifies the value of each of these symbols.

2.3 Symbolic step semantics

The symbolic step semantics for taint-tracking are presented in the following subsection. In contrast to the concrete taint-tracker, these semantics are designed to perform static taint-tracking. For that reason, ensuring termination is a key aspect. This is done similarly to the symbolic semantics of Part I, where a counter keeps track of loops, and eventually forces the exit of the loop.

We denote counters with $w \in \mathbb{W}$, and these are operated abstractly through a *step function* $\mathfrak{step}$ that takes three arguments: the initial command, the following command, an a counter, and it returns a boolean indicating if the iteration is allowed, and the new counter. When the boolean is false, the iteration cannot be executed.

Let us see how configurations are defined.

Definition 31. *A* configuration *is a 4-tuple* $(\boldsymbol{c}, \kappa, p, w)$ *composed by: a command* $\boldsymbol{c}$*, a symbolic precise store* κ*, a pointer* p*, and a counter* w*.*

$$\text{S-ASSIGN-SUB}\ \frac{\rho \vdash \mathbf{e} \Downarrow^{\eta} v}{\mathcal{I}^{\natural}, \mathcal{F} \vdash (\mathtt{x}[i]^{\eta} = \mathbf{e}, (\rho, \pi), p, w) \rightharpoonup (\mathtt{skip}, (\rho[\mathtt{x}[i] \mapsto v], \pi), p, w)}$$

$$\text{S-ASSIGN-VAR}\ \frac{}{\mathcal{I}^{\natural}, \mathcal{F} \vdash (\mathtt{x}^{\eta} = \mathtt{y}, (\rho, \pi), p, w) \rightharpoonup (\mathtt{skip}, (\rho[\mathtt{x} \mapsto \rho(\mathtt{y})], \pi), p, w)}$$

$$\text{S-SEQ-1}\ \frac{\mathcal{I}^{\natural}, \mathcal{F} \vdash (\mathbf{c}_0^{\eta_0}, (\rho, \pi), p, w) \xrightarrow{o} (\mathbf{c}_{0'}^{\eta'_0}, (\rho', \pi'), p', w')}{\mathcal{I}^{\natural}, \mathcal{F} \vdash ((\mathbf{c}_0^{\eta_0}; \mathbf{c}_1^{\eta_1}), (\rho, \pi), p, w) \xrightarrow{o} ((\mathbf{c}_{0'}^{\eta'_0}; \mathbf{c}_1^{\eta_1}), (\rho', \pi'), p', w')}$$

$$\text{S-SEQ-2}\ \frac{}{\mathcal{I}^{\natural}, \mathcal{F} \vdash ((\mathtt{skip}^{\eta}; \mathbf{c}_1^{\eta_1}), (\rho, \pi), p, w) \rightharpoonup (\mathbf{c}_1^{\eta_1}, (\rho, \pi), p, w)}$$

Figure 12.1: Simple assignments and sequences of statements of symbolic taint-tracker.

$$\text{S-IF-T}\ \frac{\rho \vdash \mathbf{b} \Downarrow^{\eta} (\beta, T^{\downarrow}) \qquad \pi' \triangleq \pi \wedge (\beta, T^{\downarrow}) \qquad \mathbf{may}(\pi')}{\mathcal{I}^{\natural}, \mathcal{F} \vdash (\mathtt{if}^{\eta}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1, (\rho, \pi), p, w) \rightharpoonup (\mathbf{c}_0^{\mathsf{fl}(\eta, T^{\downarrow})}, (\rho, \pi'), p, w)}$$

$$\text{S-IF-F}\ \frac{\rho \vdash \mathbf{b} \Downarrow^{\eta} (\beta, T^{\downarrow}) \qquad \pi' \triangleq \pi \wedge (\neg\beta, T^{\downarrow}) \qquad \mathbf{may}(\pi')}{\mathcal{I}^{\natural}, \mathcal{F} \vdash (\mathtt{if}^{\eta}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1, (\rho, \pi), p, w) \rightharpoonup (\mathbf{c}_1^{\mathsf{fl}(\eta, T^{\downarrow})}, (\rho, \pi'), p, w)}$$

Figure 12.2: Branching rules of symbolic taint-tracker.

To study this semantics, we split the rules into different categories, starting with the simplest rules.

2.3.1 Assignments and sequences of statements

For assignments, either a variable is copied to another variable, or an expression is evaluated and then assigned to the index of a variable.

Sequence rules are recursively defined, executing the head of the sequence until eventually it reaches `skip`.

2.3.2 Branching statements

For branching statements, we use proposition **may**, which was introduced previously in Chapter 3. Proposition **may** states the possible truth value of the symbolic path, meaning that such path is feasible. More specifically, proposition $\mathbf{may}(\pi)$ holds when there exists a valuation ν where $[\![\pi]\!]\nu = \mathbf{tt}$. These two rules alter the configuration by constraining the symbolic path and updating the command to execute.

2.3.3 Loops

Loops need to be carefully managed since it is possible that they never terminate, in which case, we want to stop the taint-tracker forcefully. Loops are handled via a counter w with a $\mathfrak{step}$ function that determines when the analysis needs to be stopped. If function $\mathfrak{step}$ returns

$$\text{S-WHILE-F}\ \frac{\mathfrak{step}(\mathbf{c}, \mathbf{c}', w) = (\mathbf{tt}, w') \qquad \rho \vdash \mathbf{b} \Downarrow^{\eta} (\beta, T^{\downarrow}) \qquad \pi' \triangleq \pi \wedge (\neg\beta, T^{\downarrow}) \qquad \mathbf{may}(\pi')}{\mathcal{I}^{\natural}, \mathcal{F} \vdash (\texttt{while}^{\eta}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0, (\rho, \pi), p, w) \rightharpoonup (\texttt{skip}, (\rho, \pi'), p, w')}$$

$$\text{S-WHILE-T}\ \frac{\mathfrak{step}(\mathbf{c}, \mathbf{c}', w) = (\mathbf{tt}, w') \qquad \rho \vdash \mathbf{b} \Downarrow^{\eta} (\beta, T^{\downarrow}) \qquad \pi' \triangleq \pi \wedge (\beta, T^{\downarrow}) \qquad \mathbf{may}(\pi')}{\mathcal{I}^{\natural}, \mathcal{F} \vdash (\texttt{while}^{\eta}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0, (\rho, \pi), p, w) \rightharpoonup ((\mathbf{c}_0^{\mathsf{fl}(\eta,t)}; \texttt{while}^{\mathsf{fl}(\eta,t)}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0), (\rho, \pi'), p, w')}$$

$$\text{S-WHILE-END}\ \frac{\mathfrak{step}(\mathbf{c}, \mathbf{c}', w) = (\mathbf{ff}, w')}{\mathcal{I}^{\natural}, \mathcal{F} \vdash (\texttt{while}^{\eta}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}_0, (\rho, \pi), p, w) \rightharpoonup (\texttt{end}, (\rho, \pi), p, w')}$$

Figure 12.3: Rules for loop iteration in symbolic taint-tracker.

true, indicating that an execution step must happen, two rules can be applied: rule S-WHILE-T accesses the loop, unrolling it once. Meanwhile, S-WHILE-F exits the loop directly. However, if $\mathfrak{step}$ returns false, the command is replaced with a new command `end`, indicating that the execution ended abruptly. This special statement has no corresponding rule to apply, meaning that the configuration is a final configuration, similarly to `skip`.

2.3.4 Function calls

Last four rules regard function calls, and are defined in Figure 12.4. Rule S-ASSIGN-FUN remains very similar to its concrete version. When the function call is made, and the function reaches a `return` statement, this rule can be applied succesfully, and the execution may continue. However, if the execution of the function body has a loop that reached the bound of iterations, rule S-ASSIGN-FUN-END is applied. This causes the function evaluation to end abruptly, setting the configuration statement to `end`, but keeping the observations generated. The last two rules are S-FUN-INPUT and S-FUN-SINK. Rule S-FUN-INPUT reads a value from an abstract input channel and adds the corresping source. Rule S-FUN-SINK generates an observation with the symbolic value that was passed as an argument.

In the following example we show different cases to provide intuition on the symbolic taint-tracker.

Example 26. *Program 12.5a defines a function that takes an argument, sinks it, and then returns the input value plus a randomly generated number. The function call is located inside the body of the* `if` *statement.*

We specify the input channels for input-functions.

$$\begin{aligned} \mathcal{I}^{\natural}(\mathbf{input}) &= [\boldsymbol{i}; \ldots] \\ \mathcal{I}^{\natural}(\mathbf{rand}) &= [\boldsymbol{r}; \ldots] \end{aligned}$$

Since we are looking at the symbolic taint-tracker, the input channels are abstract, that is, they hold symbols. These symbols represent external inputs that are initially not constrained. This makes taking different paths possible during an execution. For instance, both branches of an `if` *statement may be satisfiable.*

$$\text{S-ASSIGN-FUN}\ \frac{\begin{array}{c}(\mathbf{g},(\mathtt{x}_1,\ldots,\mathtt{x}_n),\mathbf{c})\in\mathcal{F}\qquad \rho\vdash \mathbf{e}_i\Downarrow^{\eta}(\varepsilon_i,T_i^{\downarrow})\text{ for }1\le i\le n\\ \mathcal{I}^{\natural},\mathcal{F}\vdash(\mathbf{c}((\varepsilon_1,T_1^{\downarrow}),\ldots,(\varepsilon_n,T_n^{\downarrow})),(\rho,\pi),p,w)\xrightarrow{o}{}^{*}(\texttt{return}\ [\mathbf{e}'_1,\ldots,\mathbf{e}'_k],(\rho',\pi'),p',w')\\ \rho'\vdash\mathbf{e}'_i\Downarrow^{\eta}(\varepsilon_{i'},T_{i'}^{\downarrow})\text{ for }1\le i\le k\end{array}}{\mathcal{I}^{\natural},\mathcal{F}\vdash(\mathtt{x}^{\eta}=\mathbf{g}(\mathbf{e}_1,\ldots,\mathbf{e}_n),(\rho,\pi),p,w)\xrightarrow{o}(\texttt{skip},(\rho[\mathtt{x}\mapsto((\varepsilon_{1'},T_{1'}^{\downarrow}),\ldots,(\varepsilon_{k'},T_{k'}^{\downarrow}))],\pi'),p',w)}$$

$$\text{S-ASSIGN-FUN-END}\ \frac{\begin{array}{c}(\mathbf{g},(\mathtt{x}_1,\ldots,\mathtt{x}_n),\mathbf{c})\in\mathcal{F}\qquad \rho\vdash \mathbf{e}_i\Downarrow^{\eta}(\varepsilon_i,T_i^{\downarrow})\text{ for }1\le i\le n\\ \mathcal{I}^{\natural},\mathcal{F}\vdash(\mathbf{c}((\varepsilon_1,T_1^{\downarrow}),\ldots,(\varepsilon_n,T_n^{\downarrow})),(\rho,\pi),p,w)\xrightarrow{o}{}^{*}(\texttt{end},(\rho',\pi'),p',w')\end{array}}{\mathcal{I}^{\natural},\mathcal{F}\vdash(\mathtt{x}^{\eta}=\mathbf{g}(\mathbf{e}_1,\ldots,\mathbf{e}_n),(\rho,\pi),p,w)\xrightarrow{o}(\texttt{end},(\rho',\pi'),p',w')}$$

$$\text{S-FUN-INPUT}\ \frac{\begin{array}{c}(\mathbf{g},T^{\downarrow})\in\mathcal{F}\\ \mathsf{read}(\mathcal{I}^{\natural},\mathbf{g},p)=((\varepsilon_1,\ldots,\varepsilon_n),p')\qquad \varepsilon t=((\varepsilon_1,\eta\cup T^{\downarrow}),\ldots,(\varepsilon_n,\eta\cup T^{\downarrow}))\end{array}}{\mathcal{I}^{\natural},\mathcal{F}\vdash(\mathtt{x}^{\eta}=\mathbf{g}(\ ),(\rho,\pi),p,w)\rightharpoonup(\texttt{skip},(\rho[\mathtt{x}\mapsto\varepsilon T^{\downarrow}],\pi),p',w)}$$

$$\text{S-FUN-SINK}\ \frac{(\mathbf{g},t_{\uparrow})\in\mathcal{F}\qquad \rho\vdash\mathbf{e}\Downarrow^{\eta}(\varepsilon,T^{\downarrow})\qquad o=(T^{\downarrow},\varepsilon,t_{\uparrow})}{\mathcal{I}^{\natural},\mathcal{F}\vdash(\mathbf{g}(\mathbf{e})^{\eta},(\rho,\pi),p,w)\xrightarrow{o}(\texttt{skip},(\rho,\pi),p,w)}$$

Figure 12.4: Rules for function evaluation in symbolic taint-tracker.

```
1  def g(x):
2      y = rand()
3      print(x)
4      return (x + y)
5  x = input()
6  if (x > 0):
7      y = g(x)
```

(a) Intricate function call.

```
1  i = rand()
2  y = rand()
3  while (i < y):
4      i++
```

(b) Non terminating program.

Figure 12.5: Two example programs to display the symbolic taint-tracker semantics.

Execution starts at line 5. In line 5, `x` *is assigned the value* $\boldsymbol{i}$ *plus the incoming taint from function* `input`*. Since* $\boldsymbol{i}$ *has no constraints, it is possible for this execution to take either paths: the guard may hold, and the guard may not hold.*

First, we look at the case where the semantics chooses the false branch.

$$\begin{array}{ll}\textit{Line 5:} & ((\mathtt{x}=\texttt{input}(\ );\texttt{if}\ldots),([\,],\boldsymbol{tt}),p_0,w_0)\\ & \quad\downarrow\ \text{S-FUN-INPUT}\\ \textit{Line 6:} & (\texttt{if}\ \mathtt{x}>0\ \texttt{then}\ \mathbf{g}(\mathtt{x})\ \texttt{else}\ \texttt{skip},([\mathtt{x}\mapsto(\boldsymbol{i},\mathsf{input}^{\downarrow})],\boldsymbol{tt}),p_1,w_0)\\ & \quad\downarrow\ \text{S-IF-F}\\ \textit{Line 8:} & (\texttt{skip},([\mathtt{x}\mapsto(\boldsymbol{i},\mathsf{input}^{\downarrow})],\boldsymbol{i}\le 0),p_1,w_0)\end{array}$$

Since the body of the `else` *is* `skip`*, the execution ends there, and no observations are generated.*

Instead, if the true branch is taken, the body of the `if` *is not empty.*

$$\begin{aligned}
&\textit{Line 5:}\quad (\texttt{x} = \textbf{input}(\ ); \texttt{if}\ldots), [\ ], p_0)\\
&\qquad\quad \lfloor\ \textsc{s-fun-input}\\
&\textit{Line 6:}\quad (\texttt{if x} > 0\ \texttt{then g(x) else skip}, ([\texttt{x} \mapsto (i, \mathsf{input}^{\downarrow})], \boldsymbol{tt}), p_1, w_0)\\
&\qquad\quad \lfloor\ \textsc{s-if-t}\\
&\textit{Line 7:}\quad (\texttt{y} = \texttt{g(x)}, ([\texttt{x} \mapsto (i, \mathsf{input}^{\downarrow})], i > 0), p_1, w_0)
\end{aligned}$$

From line 7, the next rule that applies is S-ASSIGN-FUN*. To gain insight, we show the execution of the body of the function and how the value is then returned.*

It is important to notice at this stage that the symbolic path remains the same, as the symbolic path does not constrain variables, but symbols. These constraints still apply inside the function call, and new constraints that arise in the function call should also be considered after the function finishes. Variable `x` *is assigned the value of the argument passed, in this case* `x`*, making the symbolic precise store remain equal.*

$$\begin{aligned}
&\textit{Line 2:}\quad (\texttt{y} = \textbf{rand}(); \ldots, ([\texttt{x} \mapsto (i, \mathsf{input}^{\downarrow})], i > 0), p_1, w_0)\\
&\qquad\quad \lfloor\ \textsc{s-fun-input}\\
&\textit{Line 3:}\quad (\textbf{print}(\texttt{x}); \ldots, ([\texttt{x} \mapsto (i, \mathsf{input}^{\downarrow}); \texttt{y} \mapsto (r, \emptyset)], i > 0), p_1, w_0)\\
&\qquad\quad \lfloor\ \textsc{s-fun-sink},\ o = (\mathsf{input}^{\downarrow}, i, \mathsf{print}_{\uparrow})\\
&\textit{Line 4:}\quad (\texttt{return x} + \texttt{y}, ([\texttt{x} \mapsto (i, \mathsf{input}^{\downarrow}); \texttt{y} \mapsto (r, \emptyset)], i > 0), p_1, w_0)
\end{aligned}$$

Rule S-FUN-ASSIGN *dictates that the expression in the* `return` *statement must now be evaluated, and then assigned to* `y` *in the top-level of the program.*

$$[\texttt{x} \mapsto (i, \mathsf{input}^{\downarrow}); \texttt{y} \mapsto (r, \emptyset)] \vdash \texttt{x} + \texttt{y} \Downarrow (i + r, \mathsf{input}^{\downarrow})$$

Finally, the end of this trace is as follows:

$$\begin{aligned}
&\textit{Line 7:}\quad (\texttt{y} = \texttt{g(x)}, ([\texttt{x} \mapsto (i, \mathsf{input}^{\downarrow})], i > 0), p_1, w_0)\\
&\qquad\quad \lfloor\ \textsc{s-assign-fun},\ o = (\mathsf{input}^{\downarrow}, i, \mathsf{print}_{\uparrow})\\
&\textit{Line 8:}\quad (\texttt{skip}, ([\texttt{x} \mapsto (i, \mathsf{input}^{\downarrow}); \texttt{y} \mapsto (i + r, \mathsf{input}^{\downarrow})], i > 0), p_1, w_0)
\end{aligned}$$

Then, observation o is relevant, and the program is not secure with respect to weak secrecy.

Example 27. *Program 12.5b takes two random values and iterates, adding one to the value of* `i` *until* $\texttt{i} \geq \texttt{y}$*. This program is safe with respect to weak secrecy by design, since there are no sinks. However, since the values of* `rand` *depend on its input channel, it might be the case that* `y` *is sufficiently greater than* `i` *to reach the bound of iterations in a concrete execution. For the symbolic taint-tracker, the abstract input channel of* `rand` *is as follows:*

$$\mathcal{I}^{\natural}(\textbf{rand}) \quad = \quad [r_0; r_1; \ldots]$$

The execution of the first assignments is straightforward.

$$
\begin{array}{ll}
\textit{Line 1:} & (\texttt{i} = \texttt{rand}(\,);\ldots,([\,],\boldsymbol{tt}),p_0,w_0) \\
& \quad \downarrow \ \textsc{s-fun-input} \\
\textit{Line 2:} & (\texttt{y} = \texttt{rand}(\,);\ldots,([\texttt{i} \mapsto (r_0,\emptyset)],\boldsymbol{tt}),p_1,w_0) \\
& \quad \downarrow \ \textsc{s-fun-input} \\
\textit{Line 3:} & (\texttt{while i} < \texttt{y do i} = \texttt{i} + 1,([\texttt{i} \mapsto (r_0,\emptyset);\texttt{y} \mapsto (r_1,\emptyset)],\boldsymbol{tt}),p_2,w_0)
\end{array}
$$

Since we have no constraint over r_0 or r_1, we may assume that the loop is never executed, applying rule S-WHILE-F*. In that case, the trace is as follows:*

$$
\begin{array}{ll}
\textit{Line 3:} & (\texttt{while i} < \texttt{y do i} = \texttt{i} + 1,([\texttt{i} \mapsto (r_0,\emptyset);\texttt{y} \mapsto (r_1,\emptyset)],\boldsymbol{tt}),p_2,w_0) \\
& \quad \downarrow \ \textsc{s-while-f} \\
\textit{Line 5:} & (\texttt{skip},([\texttt{i} \mapsto (r_0,\emptyset);\texttt{y} \mapsto (r_1,\emptyset)],r_0 \geq r_1),p_2,w_0)
\end{array}
$$

Instead, applying rule S-WHILE-T*, acceses the loop, updating the symbolic path and the counter.*

$$
\begin{array}{ll}
\textit{Line 3:} & (\texttt{while i} < \texttt{y do i} = \texttt{i} + 1,([\texttt{i} \mapsto (r_0,\emptyset);\texttt{y} \mapsto (r_1,\emptyset)],\boldsymbol{tt}),p_2,w_0) \\
& \quad \downarrow \ \textsc{s-while-t} \\
\textit{Line 4:} & ((\texttt{i} = \texttt{i} + 1;\texttt{while}\ldots),([\texttt{i} \mapsto (r_0,\emptyset);\texttt{y} \mapsto (r_1,\emptyset)],r_0 < r_1),p_2,w_1) \\
& \quad \downarrow \ \textsc{s-assign-sub} \\
\textit{Line 3:} & (\texttt{while i} < \texttt{y do i} = \texttt{i} + 1,([\texttt{i} \mapsto (r_0+1,\emptyset);\texttt{y} \mapsto (r_1,\emptyset)],r_0 < r_1),p_2,w_1) \\
& \quad \vdots \\
\textit{Line 3:} & (\texttt{while i} < \texttt{y do i} = \texttt{i} + 1,([\texttt{i} \mapsto (r_0+n,\emptyset);\texttt{y} \mapsto (r_1,\emptyset)],r_0+n-1 < r_1),p_2,w_n) \\
& \quad \downarrow \ \textsc{s-while-end} \\
\textit{Line 5:} & (\texttt{end},([\texttt{i} \mapsto (r_0+n,\emptyset);\texttt{y} \mapsto (r_1,\emptyset)],r_0+n-1 < r_1),p_2,w_{n+1})
\end{array}
$$

After n steps, the bound of iterations is reached, as determined by

$$\mathfrak{step}((\texttt{while}\ldots),\ldots,w_n) = (\boldsymbol{ff},w_{n+1}).$$

At said point, the only option is to set the command `end` *and finish the execution abruptly. Notice that any observations that happen before reaching* `end` *are* real *observations that can be observed in the concrete execution.*

3 Completeness and soundness properties

The symbolic taint-tracker presented does not perform any over-approximations, but in some situations it does the opposite: when the bound of iterations is exceeded, the analysis halts, under-approximating the program semantics. This is visible in Program 1.2c, where we cannot assure that the program is safe. However, not every program has to be under-approximated: some programs can be fully explored, meaning that the analyzer is sound for those programs. Thanks to the concretization function and that the semantics have been crafted to imitate that of

the concrete taint-tracker of Chapter 10, in this chapter we prove that the symbolic taint-tracker is *complete*, and *partially-sound* with respect to the concrete taint-tracker.

3.1 Completeness

In the following lemma, for each possible step of the symbolic taint-tracker, there exists a memory and a valuation that corresponds to it. Between these two executions, the observations are equal. So is the case for the pointer, and the counter.

Lemma 2 (Completeness in 1 step)**.** *For any program $\boldsymbol{c}$, any symbolic precise store κ, any pointer p, any abstract input channels set $\mathcal{I}^\sharp$, and any set of functions $\mathcal{F}$. Let $\mathbb{K}'$ denote the set of symbolic precise stores when executing $\boldsymbol{c}$ one step from κ.*

$$\mathbb{K}' = \{(\kappa', o') \mid \mathcal{I}^\sharp, \mathcal{F} \vdash (\boldsymbol{c}, \kappa, p, w) \xrightarrow{o'} (\boldsymbol{c}', \kappa', p', w')\}$$

For any $(\kappa', o') \in \mathbb{K}'$, for each $(\mu', \nu') \in \gamma_{\mathbb{K}}(\kappa')$,

$$\begin{aligned} \exists\, o, \mu, \nu, \mathcal{I} : \quad & (\mu, \nu) \in \gamma_{\mathbb{K}}(\kappa) \ \wedge\ \mathcal{I} = [\![\mathcal{I}^\sharp]\!](\nu') \ \wedge \\ & \mathcal{I}, \mathcal{F} \vdash (\boldsymbol{c}, \mu, p) \xrightarrow{o} (\boldsymbol{c}', \mu', p') \ \wedge \\ & \nu \preceq \nu' \wedge o = o' \end{aligned}$$

Then, we expand Lemma 2 for n steps.

Lemma 3 (Completeness)**.** *For any programs $\boldsymbol{c}, \boldsymbol{c}'$, any symbolic precise store κ, any pointer p, any abstract input channels set $\mathcal{I}^\sharp$, and any set of functions $\mathcal{F}$. Let $\mathbb{K}^f$ denote the set of final symbolic precise stores when executing $\boldsymbol{c}$ from κ.*

$$\mathbb{K}^f = \{(\kappa^f, o') \mid \mathcal{I}^\sharp, \mathcal{F} \vdash (\boldsymbol{c}, \kappa, p, w)\ (\xrightarrow{o_1})^n\ (\boldsymbol{c}', \kappa^f, p^f, w^f)\}$$

For any $(\kappa^f, o') \in \mathbb{K}^f$, for each $(\mu^f, \nu^f) \in \gamma_{\mathbb{K}}(\kappa^f)$,

$$\begin{aligned} \exists\, o \in \mathbb{O}^*,\ (\mu, \nu) \in \gamma_{\mathbb{K}}(\kappa) : \quad & \mathcal{I} = [\![\mathcal{I}^\sharp]\!](\nu) \wedge \mathcal{I}, \mathcal{F} \vdash (\boldsymbol{c}, \mu, p, w)\ (\xrightarrow{o_2})^n\ (\boldsymbol{c}', \mu^f, p^f, w^f) \ \wedge \\ & \nu \preceq \nu^f \ \wedge\ o_1 = o_2. \end{aligned}$$

Theorem 9 states that given a collecting execution of the symbolic taint-tracker, if no relevant observations happen, and no `end` statement is reached, the program is weak secret.

Theorem 9 (Symbolic taint-tracker implies weak secrecy)**.** *For any program $\boldsymbol{c}$, any pointer p, any abstract input channels set $\mathcal{I}^\sharp$, and any set of functions $\mathcal{F}$. Let*

$\mathbb{K}^{\texttt{skip}} = \{(\kappa', o') \mid \mathcal{I}^\sharp, \mathcal{F} \vdash (\boldsymbol{c}, ([\,], \boldsymbol{tt}), p_0, w_0,)\ (\xrightarrow{o})^*\ (\texttt{skip}, \kappa', p', w')\ \}$ *(execution finished succesfully)*

$\mathbb{K}^{\texttt{end}} = \{(\kappa', o') \mid \mathcal{I}^\sharp, \mathcal{F} \vdash (\boldsymbol{c}, ([\,], \boldsymbol{tt}), p_0, w_0,)\ (\xrightarrow{o})^*\ (\texttt{end}, \kappa', p', w')\ \}$ *(execution reached unfolding bound)*

Then,

$$\forall (\kappa', o) \in \mathbb{K}^{\texttt{skip}} : o \overset{\mathfrak{R}}{=} [\,] \ \wedge\ \mathbb{K}^{\texttt{end}} = \emptyset \Rightarrow WS \vDash \boldsymbol{c}.$$

Note that the double implication does not hold: let $\mathbf{c} = \texttt{while}\ \mathbf{tt}\ \texttt{do skip}$, then $WS \vDash \mathbf{c}$ but $\mathbb{K}^{\texttt{end}} \neq \emptyset$. Also, the assumption $\mathbb{K}^{\texttt{end}} = \emptyset$ is important. Let us assume there is a program where the initial value of `i` is zero. Then, there is a loop that adds one to `i` in each iteration, and an `if` statement that has a vulnerability, but the guard of the `if` is $\texttt{i} > n$ where n is a large enough number to be greater than the iteration limit. This program is not weak secret, given that after n iterations, the attack happens. However, given that no symbolic trace reaches the body of the `if`, the illegal flow is never explored. Such executions belong in set $\mathbb{K}^{\texttt{end}}$.

Lastly, Theorem 10 states that given a symbolic trace that annotates a relevant observation, a concrete trace behaves similarly. Thus, the program is not weak secret and a counter-example can be generated.

Theorem 10 (SE gives counter-example)**.** *For any program $\mathbf{c}$, any pointer p, any abstract input channels set $\mathcal{I}^\sharp$, and any set of functions $\mathcal{F}$. Let*

$$\mathbb{K}^{\texttt{skip}} = \{(\kappa', o') \mid \mathcal{I}^\sharp, \mathcal{F} \vdash (\mathbf{c}, ([\,], \mathbf{tt}), p_0)\ (\xrightarrow{o})^*\ (\texttt{skip}, \kappa', p')\ \}\ \textit{(execution finished successfully)}$$
$$\mathbb{K}^{\texttt{end}} = \{(\kappa', o') \mid \mathcal{I}^\sharp, \mathcal{F} \vdash (\mathbf{c}, ([\,], \mathbf{tt}), p_0)\ (\xrightarrow{o})^*\ (\texttt{end}, \kappa', p')\ \}\ \textit{(execution reached unfolding limit)}$$

If $\exists \mu', \nu', \kappa', o_1 : (\kappa', o_1) \in \mathbb{K}^{\texttt{skip}} \cup \mathbb{K}^{\texttt{end}} \ \wedge\ o_1 \overset{\mathfrak{N}}{\neq} [\,] \ \wedge\ (\mu', \nu') \in \gamma_{\mathbb{K}}(\kappa')$, *then*

$$\mathcal{I}, \mathcal{F} \vdash (\mathbf{c}, [\,], p)\ (\xrightarrow{o_2})^*\ (\texttt{skip}, \mu', p') \textit{ with } \mathcal{I} = [\![\mathcal{I}^\sharp]\!](\nu') \ \wedge\ o_1 = o_2.$$

Proof is done by applying Lemma 3.

3.2 Soundness up-to-a-bound

If the symbolic taint-tracker reaches the bound of iterations, the execution is ended abruptly, discarding a valid trace. When this happens, we cannot consider the symbolic taint-tracker sound. However, if set $\mathbb{K}^{\texttt{end}}$ is empty, it means that no traces were disregarded, and the symbolic taint-tracker covered the whole semantics of the program with respect to the concrete taint-tracker semantics.

Theorem 11. *For any program $\mathbf{c}$, abstract input-channel $\mathcal{I}^\sharp$, set of functions $\mathcal{F}$. Let $\mathbb{K}^{\texttt{end}}$ denote the set*

$$\mathbb{K}^{\texttt{end}} = \{\kappa' \mid \exists p', w' : \mathcal{I}^\sharp, \mathcal{F} \vdash (\mathbf{c}, ([\,], \mathbf{tt}), p_0, w_0)(\xrightarrow{o})^*(\texttt{end}, \kappa', p', w')\}.$$

Then, $\mathbb{K}^{\texttt{end}} = \emptyset$ if and only if

$$\forall \mu, o : (\mathbf{c}, [\,], p_0)(\xrightarrow{o})^*(\texttt{skip}, \mu, p) \ \Rightarrow\ \exists \nu, \rho, \pi, w : \begin{array}{l} (\mathbf{c}, [\,], p_0, w_0)(\xrightarrow{o})^*(\texttt{skip}, (\rho, \pi), p, w) \\ \wedge\ (\mu, \nu) \in \gamma_{\mathbb{KT}}(\rho, \pi). \end{array}$$

Given $\mathcal{I}^\sharp$, and $\mathcal{F}$, when a program $\mathbf{c}$ is such that $\mathbb{K}^{\texttt{end}} = \emptyset$, we say that the semantics is sound with respect to $\mathbf{c}$.

The proof of this theorem is done by structural induction over statements. Based on the assumption that $\mathbb{K}^{\texttt{end}}$ is empty, we know that at no point the symbolic semantics can reach the

bound of iterations. Then, we show that each step taken by the concrete semantics corresponds to a step in the symbolic semantics.

Chapter 13

A combined taint-tracker

In Chapter 12, we presented a symbolic taint-tracking semantics. This symbolic taint-tracker is language-agnostic and complete. The downside of it, is that it does not scale: it explores programs exhaustively, approximately duplicating the amount of states for each branching statement. In this chapter, we assume the existence of a sound taint-tracker analyzer, and we propose a method to combine both the sound and the complete taint-trackers. Through the combination of the analyzers, we aim to achieve a more encompassing result than when using the tools on their own.

1 Sound taint-tracker

For the combined analysis we require an alternative taint-tracker than the symbolic one presented in Chapter 12. Instead of defining it ourselves, we allow any third-party taint-tracker to be used, as long as it satisfies two conditions:

1. the taint-tracker must be sound with respect to the concrete semantics;
2. it must return an over-approximation of the set of observations generated by the concrete semantics.

We assume the existence of some abstract domain $\mathbb{A}$, with a concretization function $\gamma_{\mathbb{A}}$ from an abstract element to a concrete store $\mu \in \mathbb{M}$. We assume the existence of a *sound taint-tracker* analyzer denoted by $[\![\_]\!]^\sharp : \mathbb{A} \to (\mathbb{A} \times \mathcal{P}(\mathbb{O}))$. We write a_0 for the initial state in $\mathbb{A}$. This sound taint-tracker is parameterized by a program $\mathbf{c}$. Given an abstract state, it generates a new abstract state that over-approximates the concrete semantics of $\mathbf{c}$, plus an over-approximated set of observations.

Assumption 1 (Sound taint-tracker)**.** *For a given command $\boldsymbol{c}_0$, and a set of functions $\mathcal{F}$,*

$$\begin{aligned}\forall a_0, \mu_0, \mu, \boldsymbol{c}, p: \quad & \mu_0 \in \gamma_{\mathbb{A}}(a_0) \ \wedge \ (\boldsymbol{c}_0, \mu_0, p_0)(\xrightarrow{o})^*(\boldsymbol{c}, \mu, p) \implies \\ & \exists a : [\![\boldsymbol{c}_0]\!]^\sharp_{\mathcal{F}} a_0 = (a, o_s) \ \wedge \ \mu \in \gamma_{\mathbb{A}}(a) \ \wedge \ o \subseteq o_s.\end{aligned}$$

While in the symbolic taint-tracker, the semantics split at branching statements, the sound taint-tracker joins different paths producing a single abstract state as output. Together with the

```
1 def f():
2     return input()
3 def g(x):
4     print(x)
5 x = f()
6 if (x > 0):
7     y = g(x)
```

Figure 13.1: A program with a sink that is accessed only if x is bigger than zero.

abstract state, a set of observations is outputted. When no relevant observations were generated, we assume the program is secure with respect to weak secrecy.

Theorem 12. *For a given command $\boldsymbol{c}_0$, and a set of functions $\mathcal{F}$,*

$$\left(\exists a, o_s : [\![\boldsymbol{c}_0]\!]^\sharp_{\mathcal{F}} a_0 = (a, o_s) \wedge o_s \overset{\mathfrak{R}}{=} [\,]\right) \Longrightarrow WS \vDash \boldsymbol{c}$$

Proof outline. Assuming that a_0 comprehends all initial memories, and assuming the hypothesis of the theorem. Let us take any initial memory such that $\mu_0 \in \gamma_{\mathbb{A}}(a_0)$. Then, by Assumption 1, we have that any concrete execution $(\mathbf{c}_0, \mu_0, p_0)(\xrightarrow{o})^*(\mathbf{c}, \mu, p)$ is such that $o \subset o_s$, where o_s is the observations generated by the abstract taint-tracker: $[\![\mathbf{c}_0]\!]^\sharp_{\mathcal{F}} a_0 = (a, o_s)$. Then, this plus the hypothesis of the theorem implies that, no concrete execution generated observations. Therefore, the program is weak secret.

Example 28. *We focus on the program found in Figure 13.1. This program is insecure, since the tainted value flows to function* g *when the value is positive.*

The sound taint-tracker over-approximates the concrete semantics, meaning that it reports the illegal trace. We have that the set o_s of observations is such that

$$(\mathsf{input}^{\downarrow}, \mathsf{print}_{\uparrow}) \subseteq o_s.$$

2 Theoretical basis for combined taint-tracker

Based on previous theorems, we can write a new theorem that indicates in which conditions a program is weak secret, or not.

Theorem 13. *Given a program $\boldsymbol{c}$, a set of functions $\mathcal{F}$. For any abstract input channel $\mathcal{I}^\sharp$. Let*

$$\mathbb{K}^{\texttt{skip}} = \{\ (\kappa, o) \mid \mathcal{I}^\sharp \vdash (\boldsymbol{c}, \kappa_0, p_0, w_0)(\xrightarrow{o})^*(\texttt{skip}, \kappa, p, w)\ \}$$
$$\mathbb{K}^{\texttt{end}} = \{\ (\kappa, o) \mid \mathcal{I}^\sharp \vdash (\boldsymbol{c}, \kappa_0, p_0, w_0)(\xrightarrow{o})^*(\texttt{end}, \kappa, p, w)\ \}.$$

Then,

1. $$\Big(\exists a, o_s : [\![\boldsymbol{c}]\!]^\sharp_{\mathcal{F}} a_0 = (a, o_s) \wedge o_s \overset{\mathfrak{R}}{=} [\,]\Big) \;\vee\; \Big(\forall (\kappa, o) \in \mathbb{K}^{\texttt{skip}} : o \overset{\mathfrak{R}}{=} [\,] \wedge \mathbb{K}^{\texttt{end}} = \emptyset\Big)$$
$$\Downarrow$$
$$WS \vDash \boldsymbol{c}$$

2. $$\Big(\exists (\kappa, o) \in \mathbb{K}^{\texttt{skip}} \cup \mathbb{K}^{\texttt{end}} : o \overset{\mathfrak{R}}{\neq} [\,]\Big)$$
$$\Downarrow$$
$$WS \nvDash \boldsymbol{c}$$

To prove this, we require previous theorems from this chapter and Chapter 12.

Proof. To prove this theorem, we are going to consider the two main cases.

Case 1. Let us assume that

$$\Big(\exists a, o_s : [\![\mathbf{c}]\!]^\sharp_{\mathcal{F}} a_0 = (a, o_s) \wedge o_s \overset{\mathfrak{R}}{=} [\,]\Big) \;\vee\; \Big(\forall (\kappa, o) \in \mathbb{K}^{\texttt{skip}} : o \overset{\mathfrak{R}}{=} [\,] \wedge \mathbb{K}^{\texttt{end}} = \emptyset\Big).$$

We split the disjunction in two, and check each case separately.

Case 1.1. Taking the left side of the disjunction: $\Big(\exists a, o_s : [\![\mathbf{c}]\!]^\sharp_{\mathcal{F}} a_0 = (a, o_s) \wedge o_s \overset{\mathfrak{R}}{=} [\,]\Big)$. Directly from Theorem 13, we have that $WS \vDash \mathbf{c}$.

Case 1.2. Taking the right side of the disjunction: $\Big(\forall (\kappa, o) \in \mathbb{K}^{\texttt{skip}} : o \overset{\mathfrak{R}}{=} [\,] \wedge \mathbb{K}^{\texttt{end}} = \emptyset\Big)$. Directly from Theorem 9, we have that $WS \vDash \mathbf{c}$.

Case 2. Let us assume that

$$\Big(\exists (\kappa, o) \in \mathbb{K}^{\texttt{skip}} \cup \mathbb{K}^{\texttt{end}} : o \overset{\mathfrak{R}}{\neq} [\,]\Big).$$

Directly from Theorem 10, we have that $WS \nvDash \mathbf{c}$.

□

3 A combined taint-tracker algorithm

Given a command **c** and a set of functions $\mathcal{F}$, we would like to check if this program is weak secret. If the program is not weak secret, we would like to generate a counter-example. To do so, we can use both the sound and the symbolic taint-trackers to craft a combined analysis, as illustrated in Figure 13.2. This algorithm consists of two main steps.

1. Execution of sound taint-tracker analysis. If there are no issues found, the analysis finishes successfully—that is, the program is secure.

2. When issues are found in step 1, execution of symbolic taint-tracker analysis. Given than an issue was raised, we want to check if it is a false-positive (program is secure) or if it is a real issue (program is insecure) providing a counter-example.

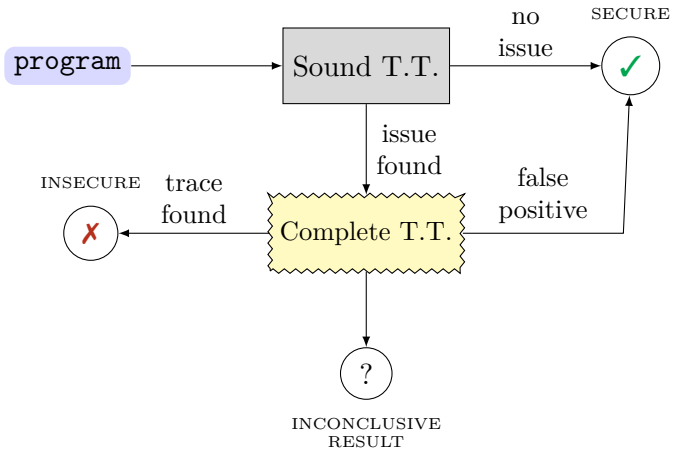


Figure 13.2: Overview of combined taint-tracker analysis. "T.T." referst to "taint-tracker".

If neither the sound taint-tracker, nor the symbolic taint-tracker, can provide an answer, it is not possible to conclude whether the program is weak secret.

- **Step 1: Application of sound taint semantics.** If no relevant observations are generated, the program is weak secret.

$$[\![\mathbf{c}]\!]^\sharp_{\mathcal{F}} a_0 = (a, o_s) \wedge o_s \overset{\mathfrak{R}}{=} [\,]$$

  If relevant observations were generated, the analysis cannot conclude whether the program is weak explicit secret or not. Thus, we go to the second step.

- **Step 2: Application of symbolic taint semantics.** In the symbolic taint-tracker, we start executing with an initial state $\kappa_0 = ([\,], \mathbf{tt})$, from command $\mathbf{c}$. Let us assume that

$$\mathbb{K}^{\texttt{skip}} = \{\ (\kappa, o) \mid \mathcal{I}^\sharp \vdash (\mathbf{c}, \kappa_0, p_0, w_0)(\xrightarrow{o})^*(\texttt{skip}, \kappa, p, w)\ \}$$
$$\mathbb{K}^{\texttt{end}} = \{\ (\kappa, o) \mid \mathcal{I}^\sharp \vdash (\mathbf{c}, \kappa_0, p_0, w_0)(\xrightarrow{o})^*(\texttt{end}, \kappa, p, w)\ \}$$

  - **Case 2.1:** If there is no final configuration in $\mathbb{K}^{\texttt{skip}}$ that generated relevant observations, and $\mathbb{K}^{\texttt{end}} = \emptyset$, then the program is weak secret.
  - **Case 2.2:** If there was an element $(\kappa, o) \in \mathbb{K}^{\texttt{skip}} \cup \mathbb{K}^{\texttt{end}}$ such that $o \overset{\mathfrak{R}}{\neq} [\,]$, the program $\mathbf{c}$ is not weak secret and there is a concrete execution that is a counter-example, that is, a concrete execution that generates a relevant observation.

- **Inconclusive result.** If neither cases **2.1** and **2.2** yield a positive answer, that implies two things:

  1. $\mathbb{K}^{\texttt{end}} \neq \emptyset$ (by case **2.1**) and,
  2. $\nexists\ (\kappa, o) \in \mathbb{K}^{\texttt{skip}} \cup \mathbb{K}^{\texttt{end}} : o \overset{\mathfrak{R}}{\neq} [\,]$ (by case **2.2**)

  Elements in the set $\mathbb{K}^{\texttt{end}}$ imply that there are executions of the symbolic semantics that were stopped at the loop-unrolling bound. These elements represent concrete traces that

are not covered in the symbolic execution, thus, not allowing us to prove that the program is weak secret. Simultaneously, all the traces lack any relevant observation.

Therefore, the program may or may not be weak secret, and the combined analysis is inconclusive.

Example 29. *We again focus on the program found in Figure 13.1. As we have seen in Example 28, the sound taint-tracker reports that there is an illegal observation.*

Then, we need to use the symbolic taint-tracker to check whether the program is secure with respect to weak secrecy.

Since the program has no loops, set $\mathbb{K}^{\texttt{end}}$ *is empty. Therefore, either all traces in* $\mathbb{K}^{\texttt{skip}}$ *generated no relevant observations, or at least one trace did.*

For any input channel $\mathcal{I}$ *such that the first input value of* `input` *is positive, the concrete execution triggers the relevant observation. By the completeness of the symbolic taint-tracker, there is a symbolic trace in* $\mathbb{K}^{\texttt{skip}}$ *that abstracts the concrete trace. Therefore, this corresponds to case 2 of Theorem 13.*

Chapter 14

An Implementation of Pysta

In the previous chapter, we presented a combined analysis to test weak secrecy. The analysis works by first applying a sound taint-tracker to prove weak secrecy. If the sound taint-tracker fails, we recur to the symbolic taint-tracker from Chapter 12. Since the symbolic taint-tracker is more precise, it can potentially prove weak secrecy, or, if not, provide a counter-example to refute the program.

The sound taint-tracker assumed in the previous chapter is completely abstracted. In this chapter our aim is to present a specific sound taint-tracker for Python named Pysa, and to instantiate it to the algorithm, exploiting further its output. Coincidentally, we instantiate the symbolic taint-tracker with Pysta, an OCaml-implemented taint-tracker, that can communicate with Pysa to raise its effectiveness. We end by discussing the limitations of this approach, and the design choices taken to develop Pysta.

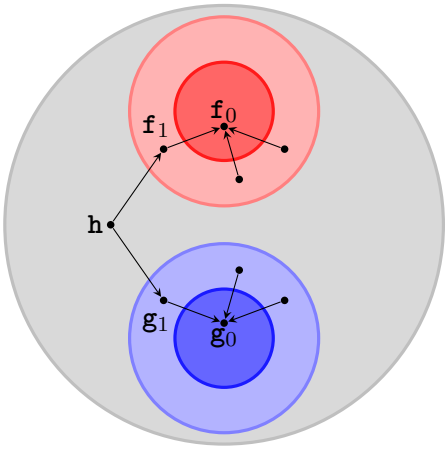
1 Pysa

Pysa is an open source python taint-tracker developed by Meta [Meta, 2023]. This taint-tracker uses abstract interpretation to perform a sound over-approximation of unwanted flows in Python code repository.

To use this tool, the user must define special *stubs*—Python signatures—that establish which functions or methods are sources, and which ones are sinks. By knowing which are the sources and sinks beforehand, the analysis can progressively build a call graph. Then, with the call graph, it must find a path from a source to a sink. This concept is illustrated in Figure 14.1.

In Figure 14.1, we assume there is function $\texttt{f}_0$. Each circle in the graph, represents the distance of the function call. For instance, $\texttt{f}_1$ calls $\texttt{f}_0$ directly. Similarly to $\texttt{f}_1$, there are other functions that call $\texttt{f}_0$, but then these functions are not called by other functions. On a higher layer of the graph, function `h` calls $\texttt{f}_1$, meaning that it has distance two to $\texttt{f}_0$ in the call graph. Function $\texttt{g}_0$ is in a similar situation: it is called by $\texttt{g}_1$, which is then called by `h`.

Then, Pysa can find a possible issue when function `h` is called. However, Pysa performs an over-approximation, meaning that this issue may be a false-alarm.




```
def f1():
    return f0()

def h():
    x = f1()
    g1(x)

def g1(x):
    return g0(x)
```

Figure 14.1: Hypothetical function-call graph where function `h` calls $\mathtt{f}_1$, and $\mathtt{f}_1$ calls $\mathtt{f}_0$. Then, function `h` calls $\mathtt{g}_1$, which then calls function $\mathtt{g}_0$.

1.1 Example

In order to get a grasp of Pysa, we propose to inspect an example using the formalism presented previously in this part. We focus on the program found in Figure 13.1. This program is insecure, since the tainted value flows to function `g` when the value is positive.

We define functions **`input`** and **`print`** as previously done.

$$\texttt{def input input}(\,) : \mathsf{input}^{\downarrow}$$
$$\texttt{def sink print}(\texttt{obj} : \mathsf{print}_{\uparrow}) : \emptyset$$

We assume the set of sources, sinks, and rules are defined as follows:

$$\mathfrak{T}^{\downarrow} = \{\mathsf{input}^{\downarrow}\} \qquad \mathfrak{T}_{\uparrow} = \{\mathsf{print}_{\uparrow}\}$$
$$\mathfrak{R} = \{(\mathsf{input}^{\downarrow}, \mathsf{print}_{\uparrow})\}$$

This program is unsafe with respect to weak secrecy based on rules $\mathfrak{R}$.

1.2 Instantiation

We now adapt the example to work on Pysa. Pysa uses two confuration files are `taint.config` and `general.pysa`. Rules from $\mathfrak{R}$ are defined in `taint.config`. Meanwhile, `general.pysa` defines the signature of functions.

Taints. Sources, sinks and rules are defined in `taint.config`, as illustrated in Figure 14.2, in JSON format. These are completely decided by the user.

Taint-signatures. For defining taint-signatures, we need to define stubs in `general.pysa`. These determine which functions or methods are sources, and which functions or methods are sinks. For the purpose of our example, we provide the signatures for **`input`** and **`print`**. Notice that, while in the formal model we provide a definition for these two functions, in Pysa, we

```
"sources": [
  { "name": "UserControlled",
    "comment": "use␣to␣annotate␣user␣input" }
],
"sinks": [
  { "name": "Print",
    "comment": "use␣to␣annotate␣printing" }
],
"rules": [
  { "name": "Possible␣data␣leakeage",
    "sources": [ "UserControlled" ],
    "sinks": [ "Print" ]
    }
]
```

Figure 14.2: Extract of `taint.config`.

```
def input(__prompt) -> TaintSource[UserControlled]: ...
def print(*objects: TaintSink[Print], sep, end, file, flush): ...
```

Figure 14.3: Signature definition in Pysa for **`input`** and **`print`**.

only provide the function signature. This is because these functions are defined in the standard library of Python already.

1.3 Execution of Pysa

Pysa is included in Pyre. Once the configurations are done, and the source directory is specified, to execute the analysis over the code we execute the command

```
pyre analyze --save-results-to output_dir
```

This executes the analyzer and save all the output in `output_dir`.

The execution of Pysa generates several JSON files. In file `errors.json` a summary of all found issues can be found, stating when sources met with a sink. In `taint-output.json` there is highly detailed information about the traces that generated these issues. However, this information needs to be processed first to be interpreted by humans. Processing `taint-output.json` can be done with *Static Analysis Post Processor* (or SAPP), generating a SQLite database that concisely contains every issue with details about the sequence of calls that generated it. Then, SAPP also allows us to explore this issues in the command line.

1. To process `taint-output.json` we use the command

   ```
   sapp --database-name sapp.db analyze output_dir/taint-output.json
   ```

where `sapp.db` is the name of the database created.

2. Then, command

```
sapp --database-name sapp.db explore
```

allows to interactively explore the issues and their respective traces.

3. Once we are exploring the database, we can request to check the issue generated:

```
>>> issues
Issue 1
            Code: 5001
        Callable: prog_stt_1.toplevel
         Sources: UserControlled
           Sinks: Print
        Location: source/prog_stt_1.py:7|11|11
Min Trace Length: Source (0) | Sink (1)
```

Here, we can observe that the issue is a $\mathsf{input}^{\downarrow}$ taint reaching a sink $\mathsf{print}_{\uparrow}$, as expected.

4. Finally, we can inspect the trace from that issue.

```
>>> trace
     #    [callable]     [port]    [location]
     1  [Missing trace frame: prog.f:result]
     2    prog.f         result    source/prog.py:5|5|7
 --> 3    prog.toplevel  root      source/prog.py:7|11|11
     4    prog.g         formal(x) source/prog.py:7|11|11
     5  [Missing trace frame: prog.g:formal(x)]
```

We can observe that the trace is composed of five levels. At level 3, there is `toplevel`, which is the root where functions `f` and `g` are called. As we can observe, at level 2, there is a call to function `f`, at line 5. And a call to `g` happens at line 7. This trace shows us which source and sink meet, and what is the flow of the taint. However, this trace is missing information about the execution of the program, that is, we have no certainty that the trace is possible, since Pysa performs an over-approximation.

Normally, at this stage, a developer has to manually check the issue and determine whether the issue is real, or if it is a false alarm.

Based on Theorem 13, since Program 13.1 generates relevant observations in Pysa, we need to apply a symbolic taint-tracker to refine the result and verify if the alarm is real or a false positive.

2 Pysta

Pysta is our symbolic taint-tracker implementation for Python, hosted at github.com/ignatirabo/pysta. Pysta is implemented in OCaml, consisting of approximately 4000 lines of code. Much of the code is reused from the symbolic executor from Part I. The Python parsing is done with `pyre-ast`, which features full-fidelity to the official Python spec. The tool does not provide full coverage of the Python language. It accepts programs with: assignments, functions, `if` conditions, `while` loops, `for` loops, a subset of classes, and dictionaries and lists without their operations.

Pysta can be used as a stand-alone tool, but the main goal of Pysta is to use output of Pysa to refute or validate alarms. Currently, when an issue is found using Pysa, the only option is to have a developer manually check the issue. All the examples that have been used previously are fairly short and simple. However, what happens when the code being analyzed is potentially thousands of lines, coming from different Python modules? The goal of Pysta is to help sort out alarms automatically, reducing the workload of developers.

Pysta performs taint-tracking, following closely the semantics presented in Chapter 12. The main differences to the formal semantics, is that it is specifically for Python, thus, it admits various types of instructions not in the syntax of the taint-language that was used in the formalism. The configuration of Pysta is done by reusing the configuration files of Pysta.

2.1 Implementation

In Pysta, executions are performed with *configurations*. Configurations represent the state of a trace at a certain point of the execution, and hold all variable information. These are a 5-tuple,

$$(\mathbf{c}, \kappa, f, i, w)$$

composed of:

1. a command $\mathbf{c}$;
2. a symbolic precise store κ;
3. a *flag* f;
4. a *ID* i;
5. and a counter w.

Two of these elements are used for debugging, so we skip their details. However, we can briefly summarize them. A flag f is used to store comments and different statuses. An ID i is unique, and it is utilized to differentiate traces and check details about the steps it went through.

To display these different elements that conform a configuration, we go through each one of them, decomposing how they are defined.

2.1.1 Commands, statements and expressions

Command, similarly to the formal definition, are a sequence of statements. These statements vary since Pysta performs analysis over the Python language. We define the set of language statements purely syntactically at this stage, with these consisting of:

- assignment;
- conditional `if` statements;
- `while` statements;
- `for` statements;
- an expression statement;
- and a `return` statement.

Assignments are straightforward, so as `if` and `while` statements. Expression statements are used for function calls that are not modifying the memory, such as **`print`**. The return statement is used for returning a value from a function call.

All of these statements make use of *language expressions*. We call language expressions to expressions that can be found in the source code that is being executed, since later we describe a different kind of expressions.

Expressions consist of the following:

- a constant—either integer, boolean or string;
- a unary or a binary operation;
- a function call;
- a subscript—that is, expressions of the form `x['a']`;
- an attribute—for example, `x.a` where `x` is a variable and `a` is an attribute;
- and joined strings.

The binary operations allowed are comparison between integers and arithmetic operations.

2.1.2 Symbolic precise store and taint expressions

In the concrete setting, evaluating language expressions returns values. Thus, in the symbolic setting, values turn into symbolic expressions, and since Pysta is a taint-tracker, these symbolic expressions must also hold taints. We call this type of expressions *taint expressions*.

The symbolic store maps variables to taint expressions, and the symbolic path is a set of taint expressions. Taint expressions are composed of:

- a constant with a taint;

- a symbolic variable with a taint;
- a unary or binary taint expression;
- a list of taint expressions;
- a map to taint expressions;
- and an object—similar to a map.

To illustrate how these expressions are composed, we can inspect a few of them. For instance, a taint expression constant is a constant plus a possible source and sink.

```
(10,SO[{Input,1}],SI[])
```

A taint expression list is a sequence of tainted expressions.

```
[(x,SO[{Input,1}],SI[{Print,2}]), (x + y,SO[],SI[])]
```

Symbolic precise store The symbolic precise store is defined as a pair of a symbolic store—also called a *symbolic variable map*—and a symbolic path.

$$\kappa = (\rho, \pi)$$

A symbolic store is a mapping from program variables to taint expressions, and a symbolic path is a set of taint expressions that can evaluate to true or false.

2.1.3 Counter

In Part I, Chapter 3, we define classic counters in Definition 3. This same definition is used for the implementation of counters in Pysta. The counter is now implemented as a stack of integers, adding one element for every new loop entered. Once the bound of iterations—a constant—is reached, the configuration goes to statement `end`.

2.1.4 Semantics

The symbolic taint-tracker starts from an initial configuration with a command **c**, and an empty symbolic state. Rule evaluation is similar to that of the formal symbolic semantics presented in Chapter 12. However, this is a *collecting semantics*. These can be seen in Program 12.5b, as illustrated in Figure 14.4. The left-hand graph is a control-flow graph, showing how the program normally behaves. In this graph, number 5 indicates the program exited, after the `while` statement. Meanwhile, the graph on the right-hand side shows the possible paths taken by the symbolic execution analysis, where f indicates the false branch, e indicates that the program reached the iteration bound and finished abruptly, and t indicating that the true branch was taken.

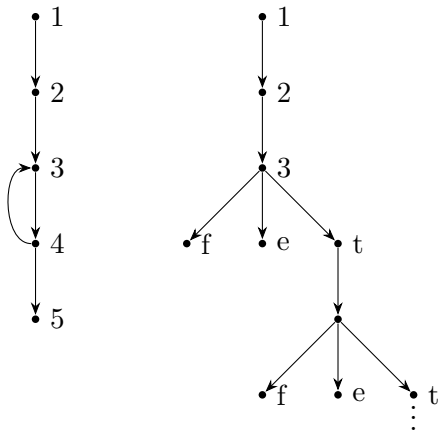


Figure 14.4: Control-flow graph for Program 12.5b on the left, and path graph for Program 12.5b on the right. Numbers to the right of the label indicate line of code to be executed next. In trace graph, label f indicates *false branch* taken, label e indicated *end branch* taken, and label t indicates *true branch* taken. The trace graph can continue infinitely until iteration bound is reached.

Inspecting the right-hand graph, each leaf in this graph represents a final trace generated by Pysta. Then, Pysta outputs the collection of all final traces and present that to the user.

In the formal semantics, we use the primitive **may**, which now is implemented through a satisfiability query with the SMT solver. When the query returns "unsatisfiable", the trace is dropped. When a relevant observation is generated, the trace is automatically terminated to eliminate overhead created by extra exploration.

2.2 Analyzer usage

Given a program of interest `program.py` to analyze, Pysta is executed as follows:

```
pysta.exe program.py
```

Pysta tries to access the database generated by SAPP based on the output of Pysa. If it succeeds, that is, if there is information about the execution of that program in the database, Pysta attempts to execute the analysis starting from the root of the program. That way, unimportant sections of the program are pruned. If the program has not been analyzed by Pysa, it will not appear in the database. Thus, no starting location will be available. In such a case, the analyzer will analyze the code from top to bottom, similarly to the Python interpreter.

Coming back to Program 13.1, where function `f` calls `input`, flowing a value to `print` through `g`. Since the call to `g` is guarded by a condition `x` > 0, there are two possible symbolic traces, and Pysta outputs the following information. This can be seen in Figure 14.5. In this figure, both branches are final, meaning that they reached `skip`. The symbolic store, also called symbolic variable map, is denoted by `SVM`. Elements in `SVM` are tuples with: the symbolic value of the variable, the set of sources, and the set of sinks. In the first trace, `x` maps to value $symb_0$ with only a source UserControlled, and the symbolic path, denoted by `SP`, only constraints $symb_0$ to

```
Final branches for programs/thesis/prog_14_1.py: 2
FINAL STATE:
   id: (2, Final)
  SVM: x -> (symb_0,SO[{UserControlled,2}]);
   SP: (symb_0 <= 0,SO[{UserControlled,2}])
No return value.

FINAL STATE, RULE TRIGGERED:
   id: (1, Final)
  SVM: objects -> (symb_0,SO[{UserControlled,2}],SI[{Print,4}]);
       x       -> (symb_0,SO[{UserControlled,2}]);
   SP: (symb_0 > 0,SO[{UserControlled,2}])
Rule 5001 triggered. 'Possible␣data␣leakage'
MODEL:
(define-fun symb_0 () Int 1)
No return value.
```

Figure 14.5: Output of Pysta for Program 13.1

be less than 1. For that reason, the program does not enter the `if` statement and the execution finishes with no observations generated.

The second trace is also final, and it triggers a rule. The symbolic path is complementary to that of the previous trace and this causes the execution to access the body of the `if`. For that reason, a call to **`g`** is done, and finally the sink is met. This interrupts the execution of that trace and returns the current symbolic store, where `objects` is the name of the argument of **`print`**.

When Pysta finds a rule is triggered, by using the SMT solver, a model can be generated to provide concrete values that lead the execution. For this example, since there is only one input value, the only requirement is that $symb_0 = 1$. In the formal model, this would imply the following: the input channel $\mathcal{I}(\texttt{input}) = [symb_0; \ldots]$, and there is a valuation ν such that $\nu(symb_0) = 1$.

3 Limitations

Neither Pysa nor Pysta are free of limitations. While some limitations might be theoretical, also design choices impact on which limitations appear.

Language subset. Pysta does not implement the full syntax of Python. However, some Python statements can be split into several simpler statements that do not alter the taint analysis. For instance, the following statement can be transformed in two statements.

```
data_json = json.dumps(charge_cash).encode('utf-8')
    |||
data_json = json.dumps(charge_cash)
data_json = encode(data_json,'utf-8')
```

Since the functionality of `encode` is not important, this transformation is admissible. However, each case must be treated carefully, as it might not be the case. Currently, this changes have to be made to the source code of the program manually which is not optimal. Ideally, Pysta should have an intermediate step where it spots cases such as the one above, transforming the program automatically and without altering the semantics of the program. The problem associated with these transformations is that if Pysa was executed on the origianl program with no modifications, its output might correspond the altered program. For instance, a source was originally in line 5 of the program, but due to these alterations, it is now in like 10, and the information extracted is no longer useful.

In Pysa, all Python programs can be analyzed, which is a desirable feature. However, Pysa performs over-approximations. This makes the work of covering the language easier. In contrast, implementing language statements in Pysta is extremely tricky as performing over-approximations completely defeats the goal of the analysis. Features implemented at the moment are still rudimentary. For instance, not all features of classes have been implemented.

Regarding program transformations, exceptions and imports are not implemented, thus, they are just commented. Shortly, we explain further why imports are not implemented. Basic operations on data structures are not implemented either, for instance, `pop` in a list. Also, assignments to labelled values in functions are not implemented.

The consequence of not covering all the Python language is that, programs of interest might not be analyzable by Pysta for containing unimplemented statements.

Taint signatures (Primitives and modules) A logical limitation shared between Pysa and Pysta is that primitives, such as `input`, need to be implemented manually. This limitation cannot be avoided.

Regarding taint-signatures, Pysa makes use of module signatures in `.pyi` format. By using Python interface files, it can automatically extract information about classes, functions, and values coming from modules. This is very important for making the process of executing Pysa as automatic as possible. While we are parsing the taint-signature file from Pysa automatically, the rest of the modules are not know to Pysta. This could be easily fixed.

Symbolic objects and functions As explained in the previous paragraph, there are possibly primitives, functions from a module, or classes from a module, whose code is unknown to Pysta. Instead of crashing, the analyzer can continue executing with a placeholder symbolic value. For instance, let us assume that there is a function `f` that is unknown. When called, it returns a new symbol with the taint of its input arguments—a pass-through function. For functions that are mentioned in the taint-signature file, we can expand this behavior. If the function is sinking an argument, not knowing the function does not alter the fact that the argument can be sunk. Similarly, if a function is a source, producing a tainted value, the placeholder symbolic value can be tainted. This allows Pysta to successfully perform taint analysis without having a perfectly precise representation of values. This approach has a limitation: if the placeholder is used as part of a guard in a control-flow statement, the analyzer loses completeness since the value,

technically, is an over approximation.

Lists and "for" loops For simplicity, the implementation of lists is not as general as it should be. We assume that every function that returns lists, returns a singleton list. This assumption is imprecise, but it is necessary to use `for` loops easily, which are heavily used in Python.

The imprecision can affect negatively certain programs, for instance, a guard asking if a list is empty, or if it has more than one element.

Sanitizers *Sanitizers* are used to model functions that clear taints from a value. Sanitizers are important in taint analysis as they can be used to model functions such as hashing functions in cryptography. For instance, an unhashed password is confidential, but hashing it makes it more secure. In this context, the hashing function is modeled as a sanitizer, clearing the taint of the original value. In Pysta, only the simplest sanitizers can be defined. This consist of removing all the taint of the input argument. This would be enough to implement the example of the hashing function.

SMT solvers are slow The most common problem with symbolic execution is that SMT solvers are computationally expensive. Thus, we want to minimize the amount of queries performed by the SMT solver to minimize the overhead. While there is pruning in Pysta, the tool is still a prototype and there are several parts of the analysis that can be optimized.

Besides that limitation, another problem is that, being very precise adds overhead by design, even when a statement is not relevant for taint-tracking. Let us assume there is a function `cmplx` that has a great computational complexity. Already executing this function concretely is not optimal. Now, symbolically executing such function could potentially crash the analysis, or drop a trace. The question is then, is the semantics of this function important for taint-tracking? In Pysa, such problems are greatly avoided by the lack of numerical computations. But in Pysta, we care about numerical precision. This situation imposes a limitation because we can choose two sub-optimal paths: the first path is to leave the analysis as it is, and let the trace be dropped, losing coverage. The other solution is to manually spot this functions and give a special definition for them such that their semantics are lost, but the "spirit of the program" is preserved.

Chapter 15

Evaluation of Pysta

In this chapter, we try to answer whether Pysta can effectively reduce the amount of human work required to check Pysa alarms, and reflect on what aspects of Pysta are limiting the integration of the two tools. To do so, we inspect three different research questions.

RQ1. Validation of issues. For an unsafe Python program, in the case where Pysa raises an alarm, is Pysta able to find a trace triggering the bug?

RQ2. Analysis refinement. For a safe Python program, in the case where Pysa raises an alarm, is Pysta able to disregard the false positive?

RQ3. Scalability. Given that Pysa is used to analyze immense codebases, what is the largest program that can be analyzed by Pysta, and what is limiting Pysta to cope with larger programs?

To answer these questions, we have split them into three sections, each one answering an individual question.

Setup. Experiments were performed on a laptop with an Apple M1 Pro with 16GB of RAM. The initial configuration has an empty symbolic state.

1 Validation of issues (RQ1)

When a real issue is raised by Pysa, we want Pysta to attest that the issue is real—that is, to find a trace that triggers the unwanted flow.

We start by detailing different examples extracted from different sources. One of them comes from a CVE regarding CPython. Two of the examples are from Cloudgoat. Last one is taken from the tutorial of Pysa. For performing the analysis of these examples, we have adapted them to be compatible with both Pysa and Pysta, while keeping the behavior of the code that is relevant to us.

All the examples are unsafe programs that raise alarms in Pysa.

1.1 Example "CPython"

Context CPython is the reference implementation of the Python interpreter, and it is written in C and Python. In 2018, a vulnerability was found in CPython, version 2.7, that can result in denial of service and information gain by exploiting the incorrect neutralization of special elements in a command constructed by externally-influenced input.

The code analyzed is 23 lines long, and can be found in Appendix 1. The vulnerability is registered as CVE-2018-1000802, and it was closed in a commit [Pierce, 2018]. This CVE had a CVSS score of 7.5 and 9.8.

Attack Module `shutil.py` of CPython contains utility functions for copying and archiving files and directory trees. The problem specifically arises from `_call_external_zip` function, which compresses a directory by using the CLI command `zip`. To do so, function `spawn` was being used, which does not neutralize shell injection attacks. The solution is to replace `spawn` with `subprocess.check_call(cmd)`.

Configuration of analyzers Since the problematic function is `spawn`, we consider it the sink, while the source is the input strings provided to the function. For that matter, we assume the source is coming from `input` function, and the configuration for `spawn` is as follows.

```
def distutils.spawn.spawn(cmd: TaintSink[RemoteCodeExecution],
        search_path, verbose, dry_run): ...
```

Pysa response Analyzing the code found in Appendix 1, Pysa is able to determine that the second argument of `_call_external_zip` is a sink, given that it flows to `spawn`. And since the value provided in the call comes directly from `input`, an issue is raised.

Pysta response The output from Pysa is taken by Pysta. Thanks to the output, Pysta can determine that the execution starts at line 21, the origin of the source. It executes, producing a single trace, until it reaches `spawn`, causing a relevant observation to be generated. Since function `spawn` is inside a `try` block, and Pysta does not implement exceptions, we need to adapt the program. To do so, we have commented the `try`, and left the `spawn`.

In this example, Pysa raised an issue, and Pysta was able to corroborate the veracity of the issue, informing the user that there is a bug.

1.2 Example "Vulnerable-λ"

Context This example, consists on 38 lines of code, and it can be found in Appendix 2, comes from a "vulnerable by design" AWS (Amazon Web Services) repository called CloudGoat. This repository has AWS deployment code that is vulnerable on purpose, and it is used as training for people working in cybersecurity.

In AWS, a feature is *AWS lambda functions*. These functions are event-driven, serverless functions provided by Amazon. These run in Amazon servers and can be triggered by different events, and can be used for multiple purposes, such as file processing, stream processing, IoT backends, and more.

Attack In this specific scenario, the AWS bucket has a lambda function `policy_applier` that is "vulnerable", meaning that it can be exploited. This lambda function is used to apply a set of policies to a user. To do so, a SQL statement is crafted to get all the policies passed as an input. However, the SQL statement is created using one of the standard ways to join strings in Python.

```
statement = f"select␣policy_name␣from␣policies
where␣policy_name='{policy}'␣and␣public='True'"
```

Using these type of string concatenation is exploitable by injecting SQL command through variable `policy`. Through this exploit, it is possible, for instance, to give administrator rights to a user.

Configuration of analyzers To perform taint-tracking on this example, we have to consider the payload as a source. Meanwhile, the sink is the execution of the SQL query. Its taint signature is as follows.

```
def sqlite_utils.db.Database.query(sql: TaintSink[SqlInjection],
        params): ...
```

Pysa response For this example, Pysa is able to determine that the first argument of function `handler` is a sink, because of the call to `db.query` in its body. This raises an issue pointing to an SQL injection, informed in the output of the tool.

Pysta response Pysta analyzes the output of Pysa, and it determines that the execution must start from the input statement, and goes through the body of `handler` until it reaches line 27, generating the relevant observation and stopping the execution.

In this example, Pysa raised an issue, and Pysta was able to corroborate the veracity of the issue, informing the user that there is a bug.

1.3 Example "SQS-λ"

Context This example, similarly to example "Vulnerable-λ", comes from the CloudGoat repository. The code is 29 lines long, and can be found in Appendix 3. In this example, the application is an online shop where the user can buy items. The secret is hidden in the most expensive item that costs 100 million credits, and the user starts with 3000 credits. Credits can be charged through the webpage but only at fixed amounts of 1, 5, and 10. Hence, to buy the secret the user should click around 10 million times.

This example is also deployed on AWS, and clicking one of the buttons to charge money, calls a Python function `charge_cash` with the amount chosen by the user. This Python function, then uses function `sqs.sqs_client.send_message` to send a *Simple Queue Service (SQS)* message, which ends up calling an AWS Lambda function.

Attack Part of the attack lies in doing privilege escalation to be able to send forged SQS messages. While technically the first part of the attack lies on the privilege escalation, the other security issue is that the lambda function does not check the input. By this, we mean that the amount to charge, coming from the input, is not checked. This type of vulnerability is different from the ones presented previously, since the flow from the input must be allowed, but only when the input is "valid".

Configuration of analyzers With this reasoning, we assume that the source is function `input`, and the sink is function `urrlib.request.urlopen` that takes has been crafted with the payload. The configuration of the sink is as follows.

```
def urllib.request.urlopen(url: TaintSink[AuthorizedChannel],
        data, timeout, cafile, capath, cadefault, context): ...
```

Pysa response Pysta is able to report the issue, pointing to the call to `input`, reaching `urlopen`.

Pysta response First, the output of Pysa is analyzed. Pysta determines that the execution must start at line 28. The analysis spans into two traces. One of the traces does not raise an alarm, concluding that the trace is safe. The other trace does generate a relevant observation. This is reported by Pysa, together with a counter-example providing a possible value for `charge_cash["charge_amount"]`. In the original program, Python checks for the existence of the field `"charge_amount"`. In Pysta, we simplify this by changing the `if` guard to check whether `"charge_amount"` is greater than 0. We also removed the exceptions, as this are not handled in Pysta.

1.4 Example "Django Application"

Context Django is a Python framework for developing web applications. It is widely used for its ease of use.

Applications in Django consist on a series of *views*. Views are functions that represent different states of the web application. For instance, there is a view for the index of the webpage, meaning that, every time the index is requested, the index view function is executed, and the webpage is generated.

The example is 12 lines long, and can be found in Appendix 4. In the example, we have the definition of function `operate_on_twos` that gets an `HttpRequest`. In this request, it expects to get two values: `"operator"` and `"range"`. Then, depending on the value of `range` that is stored in `r`, the program might return 0, or calculate an arithmetic operation.

Attack To calculate the arithmetic operation, function **eval** is used. This function takes a string input, and it evaluates that input in the Python interpreter. If the request brings numbers as expected, this program works fine. But the way the input is being passed to **eval** can be maliciously targeted because of the incorrect neutralization of special elements.

Configuration of analyzers We assume that the source is the `HttpRequest` object that the function takes as an argument, and the sink is function **eval**. The configuration is as follows.

```
django.http.HttpRequest.GET: TaintSource[UserControlled] = ...
def eval(__source: TaintSink[CodeExecution], __globals, __locals): ...
```

Pysa response Pysa reports an issue as expected.

Pysta response In this example, the output of Pysa is crucial, since there is no code at the top-level of the program. The analysis of the output of Pysa determines that the execution must begin at line 3: the definition of function `operate_on_twos`. Pysa assumes the existence of the input of the function, which has type `HttpRequest`. Based on the configuration files, the argument has a taint. Line 4 and 5, assigns values from the input argument into variables `operator` and `r`, making these variables tainted as well. To traces are generated: when `r` is zero, the trace is safe. However, if `r` is different than zero, sink function **eval** is executed, and a relevant observation is generated.

The ouput from Pysta is as follows.

```
Final branches for programs/thesis/rq1/django/views.py: 2
FINAL STATE, RULE TRIGGERED:
   id: (-1, Temporary, Final; Function evaluation)
  SVM: __source->(joined_0,SO[{UserControlled,}],SI[{CodeExecution,10}]);
       operator->(operator_0,SO[{UserControlled,}]);
              r->(range_0 + 1);
          range->(range_1);
        request->{ 'GET' : { 'operator' : (operator_0,SO[{UserControlled,}]) } };
   SP: (range_1 != 0)
  Rule 5001 triggered. 'Possible␣shell␣injection'
  MODEL: range_1 = 1
  No return value.

FINAL STATE:
   id: (1, Final; IF true)
  SVM: operator->(operator_0,SO[{UserControlled,}]);
              r->(range_0 + 1);
          range->(range_1);
        request->{ 'GET' : { 'operator' : (operator_0,SO[{UserControlled,}]) } };
         result->(0);
   SP: (range_1 = 0)
  Return value: (0)
```

1.5 Example "Billion Laughs"

The following example varies from the previous ones as it does not focus on a specific attack but on a type of attack. It is a denial-of-services attack called "Billion Laughs" that targets XML parsers.

Context Extensible Markup Language (XML) is a markup language used widely for storing and transmitting data over the internet. At the core of XML, is an XML parser that transforms XML strings to objects in memory, and attacks to the parser exist as studied in [Späth, 2016].

One important feature of XML is *entities*: these bind to XML values. An entity is defined as follows: `<!ENTITY entity-name "entity-value">`, where `entity-name` is a binding to value `entity-value`. Entities can be used for making more maintainable XML files. However, entities must be resolved at run-time, meaning that XML files have to be *inflated*. Inflating an XML file translates to parsing the XML file and creating an instance in the memory. This creates the possibility of denial-of-service attacks.

Attack A famously known attack is the *Billion Laughs* attack, where a well-formed XML string expands exponentially in the memory when inflated. While this problem was first reported in 2002, there are still vulnerable applications. For instance, in Python, module `xml.etree.ElementTree` is not secure against maliciously constructed data, meaning that it is the obligation of the developer to prevent the attack.

Configuration of analyzers Since the attack only consists of an XML input flowing to the XML parser, we do not provide an interesting example on an application. We have the attack encoded in example `xml.unsafe.py`. The input XML string must be tainted, while function `xml.etree.ElementTree.XML` is the sink. To remove the vulnerability in `xml.safe.py`, there is a sanitization function that removes the source from the value.

This example can be analyzed both by Pysa and Pysta, and it counts as a different type of problem that we can aim to model.

1.6 Discussion

These five examples display realistic levels of complexity of Python programs, while also considering limitations of our tool Pysta regarding language coverage. Considering this, the examples are unsafe and Pysa, a sound tool, reports the issues. Likewise, Pysta is able to find a trace for each of the examples, asserting that the programs are not safe.

At the moment, some of the examples have been modified manually to be able to be accepted by Pysta, but this limitation can be easily resolved.

2 Analysis refinement (RQ2)

Pysa is a sound tool, and as such, it performs over approximations. This causes the generation of false-positives—issues reporting flows that are actually unrealizable. Meanwhile, Pysta is

```
1 if b:
2   x = rand()
3 else:
4   x = input()
5 if b:
6   print(x)
```

(a)

```
1 def foo():
2   return [int(input()),
3     int(input()),
4     int(input()),
5     3]
6 ls = foo()
7 print(ls[3])
```

(b)

Figure 15.1: Safe programs that raises an alarm in Pysa. Program 15.1a, on the left, belongs to the first category of false-positives. Program 15.1b, on the right, belongs to the second category of false-positives.

complete, making it well-fitted for rebutting such false-positives. If Pysta cannot find a feasible path for the reported flow, then Pysta is refining Pysa, raising the precision of the analysis.

We call false-positives issues raised by Pysa that are not reporting an actual unwanted flow. We assume that sources and sinks are correctly set, and that the programs are indeed safe—that is, sanitizers are well set, and the tainted information does not reach the sink. Then, false-positives are only caused by the imprecision of Pysa. We focus on two, very comprehensive, categories:

1. issues caused by imprecision on expression evaluations;
2. issues caused by collapsing of structures.

The first kind of false-positive happens when Pysa, by not keeping track of values, assumes all possible paths of branching statements feasible.

The second kind of false-positive happens when structures, such as lists and dictionaries have tainted values. At a certain threshold, Pysa *collapses* the structure—that is, spreading the taint to all the elements of the structure. This mechanism is used to minimize the memory usage of the analysis.

2.1 Imprecise evaluation of expressions

Academic example. Program 15.1a is safe with respect to weak secrecy, assuming `input` is a source, and `print` is a sink. We denote this program **c**. We assume that `b` is valid guard. It is clear that the execution of `input` can only happen if guard `b` does not hold. Then the program has to be safe.

- Tool Pysa, by ignoring the guard, raises an alarm.
- Tool Pysta, in the other hand, determines the program is safe.

Example "SQS-λ". The first scenario comes by fixing the "SQS-λ" example from RQ1. The problem that we want to model is that of the lack of verification of the message. The amount of money to add to the account should be either 1, 5 or 10, but the attacker sends 100 million as

the amount. To fix the program, we can modify the code to check that the amount is correct, making the sink only reachable at certain amounts.

The cleanest solution is to define a function called `valid` that takes the input and returns true or false. If the function returns false, the execution stops. Otherwise, the function continues executing as normal. In Pysa, it would be possible to configure function `valid` as a sanitizer, removing the taints. Thus, removing the alarm.

Currently, to make this example work in our analyzer, we force the input to have either value 1, 5 or 10.

- Tool Pysa continues to raise alarms since it ignores the validation of the input. This can potentially be fixed by marking the input validation as a sanitizer.
- Tool Pysta is able to check that the input must be one of the allowed values (1, 5, 10). If the values is not 1, 5 or 10, the path will never be taken. The fixed version of the program does not raise an alarm.

2.2 Collapsing of structures

Academic example. Program 15.1b is safe with respect to weak secrecy, assuming `input` is a source, and `print` is a sink.

By configuring Pysa with setting `maximum_model_source_tree_width` lower than 4, Pysa collapses any structure larger than that.

In Program 15.1b, list `ls` has four elements, and assuming that the tree width option is 3 or lower, Pysa collapses the list, tainting every element. In Pysta, this is not the case, and taints are fully tracked, allowing the analyzer to determine that the program is safe.

- Tool Pysa raises an alarm, indicating an unwanted flow.
- Tool Pysta does not raise and alarm and considers the program safe.

Real-life example.

3 Scalability (RQ3)

The largest programs that we have performed analysis on are those of RQ1, specifically example "Vulnerable-λ" with 38 lines.. Currently, the main limitation is the lack of coverage of Python instructions, hence, we cannot test the scalability of the tool properly.

Instead, two scripts have been implemented to at least test two aspects of the analyzer. The first script checks the maximum length of the stack. The second one checks roughly how fast the analyzer explodes when multiplying its branches.

Program Name	User time	System time
`long_if1.py`	0.08s	0.01s
`long_if2.py`	0.16s	0.02s
`long_if3.py`	0.30s	0.04s
`long_if4.py`	0.61s	0.03s
`long_if5.py`	1.25s	0.06s
`long_if6.py`	2.46s	0.09s

Table 15.1: Execution times for different programs.

3.1 Stack length (function call depth)

Script `long_call_generator.py` takes an integer n as input, and generates a python program that performs n nested function calls. The output program is stored, and then the analyzer can be executed.

At the moment of writing this work, we have found that the depth limit of function calls is 24999. By trying with 25000 function calls, the analyzer terminates with a segmentation fault.

We think that this is more than enough to accommodate real world scenarios given that with the information taken from the PYSA execution, we are starting PYSTA from the closest point possible to the source.

3.2 Branching limit

For every `if` statement in the program, there are at-most two possible traces, and symbolic execution is know for *space explosion*.

Script `long_call_if.generator` takes an integer n as input, and generates a python program that performs n nested `if` statements. The output program is stored, and then the analyzer can be executed.

As expected, the complexity of the analyzer, with respect to branching statements, is n^2, where n is the amount of nested `if` statements. Results can be seen in Table 15.1.

3.3 Discussion

While we do not have large examples to inspect, where we can approximate the amount of branches, the examples and situations we have observed in RQ1, seem to indicate that the amount of `if` in these applications is not great, and the guards of the `if` statements are not highly complex, lowering the difficulty for the SMT solver.

We also believe that the length of the stack call in larger programs will never be as large as 25 thousand.

Finally, we think that it is possible in the future to perform the analysis of larger code bases.

Chapter 16

Related work

1 Semantic properties

In [Volpano, 1999], the authors introduce the security property of *weak secrecy*. This property aims to identify instances where a program discloses information through explicit flows, deviating from the traditional noninterference concept. Weak secrecy is important because it formalizes the behavior displayed by taint-trackers. However, weak secrecy only explores simple languages, assuming that the only statemets that create explicit flows are assignments.

Later, [Schoepe, 2016] introduces *explicit secrecy*, a more generic property that aims to generalize weak secrecy to other languages, including high-level languages and machine code. The problem that arises with these languages is that there are semantic statements, other than assignments, that can also modify the program state. Their framework is adaptable to various languages, and it uses a special statement **out** that generates observations. The attacker can solely observe information through such outputs. Their framework can be adapted to work with functions such as we do. However, they separate the memory in high ($\mathbb{H}$) and low ($\mathbb{L}$). This translates to assuming variables are either tainted or not tainted, while we provide customized taints, and which combinations of sources and sinks trigger alarms. Also, variables are high from the beginning, similarly to [Volpano, 1999]. Instead, taints are introduced by functions that work as souces.

2 Taint analysis

Much have been done for performing dynamic taint analysis.

[Busse, 2022] is a recent work that examines the combination of dynamic symbolic execution and static analyzers, focusing on the automation of the process of confirming potential bugs found by static analyzers for C and C++ programs. The bugs considered are memory safety bugs (such as use-after-free), memory leaks, and runtime errors. The static analyzers used in this work are Clang Static Analyzer [LLVM Projec,], Infer [Calcagno, 2011] and Klee [Cadar, 2008]. Traces generated by these static analyzers are used to create constraints, guiding the execution in

dynamic symbolic execution. However, it is noted that the real-time nature of dynamic symbolic execution leads to reduced coverage. In the best-case scenario, a bug is identified, but in the worst case, manual verification is required if no issue is detected.

In [Corin, 2012], the authors develop a dynamic symbolic execution analysis with tainting mechanics, using KLEE [Cadar, 2008], for the C language. This analysis does not only consider explicit flows, but also implicit flows. The tainting mechanics are applied on top of the checks already done by KLEE, while we only perform taint analysis for weak secrecy.

Frama-C [Kirchner, 2015], which stands for Framework for Modular Analysis of C programs, is an open-source platform designed for the static analysis of C code. Since release 24.0 of Frama-C there is support for a taint domain, that can also generate graphs of dependencies. Other work has been done on Frama-C to implement taint analysis [Ceara, 2010].

Part III

Conclusion

Chapter 17

Conclusion

1 Relational sound symbolic execution for noninterference

In Part I, we proposed a series of symbolic execution analyses, summarized in Figure 2.1. The goal of these analyses is to perform verification of programs with respect to the security property noninterference. To do so, the analyses must provide full coverage over the concrete execution semantics, in other words, be sound analyses. When executing loops, these might iterate for a great amount of times, or even infinitely from the perspective of a symbolic executor. To address this limitation, the analyses perform an over approximation of the loops. One of our main contributions is the combination of symbolic execution with abstract interpretation to perform these over approximations, in order to retain precision.

The first sound symbolic semantics presented is SoundSE. In SoundSE, when the bound of iterations is met, we perform a trivial over approximation. The over approximation consists in replacing the value of all variables that might have been modified to unused symbolic values. This over approximation makes the semantics sound. However, it is not precise, causing SoundSE to not be able to infer interesting properties of programs. However, when no over approximation is done, this semantics, and all the following semantics that are presented, can generate counter-examples.

To make SoundSE more precise, we define RedSoundSE. RedSoundSE swaps the over approximation mechanism to use either intervals or convex polyhedra analysis. When a loop bound is met, a reduction function is used to extract the constraints from the abstract states, and insert them into the symbolic path of the symbolic state. This raises the precision of the analysis, but it is still not a relational semantics, meaning that we cannot verify noninterference.

The next symbolic execution semantics presented is SoundRSE. SoundRSE is a relational symbolic execution semantics, mapping variables to either a single, or a pair of symbolic values. It is the first semantics that we present that can be used for the verification of noninterference. We offer two flavors: the first one uses the trivial over approximation from SoundSE. The second flavor, uses the over approximation of RedSoundSE. However, the over approximation applied in the RedSoundSE cannot convey information about two executions, making it not enough to prove a variety of examples.

The last symbolic semantics we present is RedSoundRSE. This semantics uses a dependences analysis to perform loop over approximation. This method works by: first, when a loop bound is met, it generates a dependences state based on the symbolic state. Then, the loop is analyzed with said dependences state. Finally, the symbolic state assigns one unused symbolic value per variable mapped to low in the final dependences state. On top of using dependences to perform the over approximation, we also offer a combination using dependences and one of the non-relational abstract domains (intervals or convex polyhedra). This is done by first performing the dependences over approximation, and then using the reduction function. This raises the precision of the analysis even further.

Our results, summarized in Table 7.1, show that on a set of challenging examples for noninterference, our analysis performs better than the dependency analysis, and is able to precisely conclude whether the programs are noninterferent, providing counter-examples when possible.

2 Static symbolic tainting analysis

In Part II, we explored static symbolic taint analysis. This is done by adapting the security property weak secrecy [Volpano, 1999] to a more modern language that has functions, and defining relevant observations as the illegal flows that can be observed in a program execution. Functions work as both sources and sinks in the language.

Next, we present a concrete taint semantics that work with programs in this language. This concrete taint semantics can be seen as a dynamic taint-tracker, tracking taints while also performing concrete execution of the program. This semantics does not time-out, and behave exactly like standard concrete semantics plus taint-tracking.

Then, we contribute our symbolic taint semantics, meant to work as a static taint analysis for the taint language. The symbolic taint semantics does not perform over approximations, making it complete. To do so, it uses counters that allow for a bound of iterations of loops. When the threshold is passed, the semantics stops the execution. However, the observations generated are still valid. This way, it allows us to get counter-examples when a relevant observation is generated, in other words, when an illegal flow happens.

By assuming the existence of a sound taint semantics, we design a combined taint analysis that uses both the sound and symbolic taint semantics, to provide a more precise response. In the best case scenarios, the symbolic taint semantics filters false alarms, or find a valid trace displaying the illegal flow. We took the combined taint analysis and instantiate to the Python programming language by using two analyzers: Pysa [Meta, 2023], a sound taint analyzer by Meta, and Pysta, a symbolic taint analyzer created by us. Pysta, while still a prototype, illustrates the symbolic taint-tracker semantics, and the limitations of such an approach, which we see as future opportunities to enhance it. In the instantiation of the combined analysis, Pysta utilizes the output of Pysa to refine the starting point of the analysis and minimize the overhead of symbolic execution. The combination of Pysta with Pysa was displayed in several cases of interest, allowing to gain insight in some of the problems that could be solved with these tools.

3 Future work

Given these encouraging results in both parts, we have thought about different ways to expand the work presented.

The first point of interest would involve picking a different language for both the sound symbolic execution, and the symbolic taint analysis. Languages such as C, where memory can be modified through pointers, and guards of `if` statements can have effects, make this task not trivial. In the case of the taint analysis, a property such as weak secrecy would not work anymore when working with low-level languages. For that reason, in the work of Schoepe et al. [Schoepe, 2016], the authors introduce property *explicit secrecy* with this goal.

For the sound symbolic execution, we have not explored replacing dependences for a different abstraction, such as the cardinality domain presented in [Assaf, 2017].

Another line of work would be replacing noninterference for a different property, even non-relational. For instance, weak secrecy, as used in Part II, could potentially be an interesting property to study. This adaptation would turn SoundRSE to a sound symbolic execution taint analysis.

Regarding the tools, both the implementation of RedSoundRSE and PYSTA are prototypes that could be optimized to be faster, and sturdier. Specifically, we are interested in covering a bigger portion of Python with PYSTA, and using more information of the output of PYSA. Potentially, we could define a new analyzer that uses the abstract representation of PYSA directly, instead of using its output. Another upgrade for PYSTA would be mixing it with concrete values to evaluate expensive functions while not losing completeness. For instance, in the context of cryptography, many functions are designed to avoid being invertible, which symbolic execution cannot deal with. By using concrete values, we could accelerate the execution while not losing completeness. A step further would be to use a fuzzer with deferred concretization, introduced in [Pandey, 2019], to handle the instances where symbolic objects and functions are causing imprecisions. This would allow to mitigate the loss of completeness introduces by this mechanic.

Appendix A

Evaluation examples for PYSA

1 Example "CPython"

```
def _call_external_zip(base_dir, zip_filename, verbose=False, dry_run=False):
    # XXX see if we want to keep an external call here
    if verbose:
        zipoptions = "-r"
    else:
        zipoptions = "-rq"
    from distutils.errors import DistutilsExecError
    from distutils.spawn import spawn
    cmd = ["zip", zipoptions, zip_filename, base_dir]
    try:
        spawn(cmd, dry_run=dry_run)
    except DistutilsExecError:
        # XXX really should distinguish between "couldn't find
        # external 'zip' command" and "zip failed".
        raise ExecError \
            ("unable to create zip file '%s': "
            "could neither import the 'zipfile' module nor "
            "find a standalone zip utility") % zip_filename

base_dir = "/Users/ignacio"
zip_filename = input("Enter zip name: ")

_call_external_zip(base_dir, zip_filename)
```

2 Example "Vulnerable-λ"

```
import boto3
from sqlite_utils.db import Database

db = sqlite_utils.db.Database("my_database.db")
iam_client = boto3.client('iam')

def handler(event, context):
    target_policies: list[int] = event['policy_names']
    user_name = event['user_name']
    print(f"target policies are : {target_policies}")

    for policy in target_policies:
        statement_returns_valid_policy = False
        statement = f"select policy_name from policies where" \
                    f"policy_name='{policy}' and public='True'"
        q: list[int] = db.query(statement)
        for row in q:
            statement_returns_valid_policy = True
            print(f"applying {row['policy_name']} to {user_name}")
            response = iam_client.attach_user_policy(
                UserName=user_name,
                PolicyArn=f"arn:aws:iam::aws:policy/{row['policy_name']}"
            )
            print("result: " +
                str(response['ResponseMetadata']['HTTPStatusCode']))

        if not statement_returns_valid_policy:
            invalid_policy_statement = f"{policy} is not an approved policy, " \
                                       f"please only choose from approved " \
                                       f"policies and don't cheat. :) "
            print(invalid_policy_statement)
            return invalid_policy_statement

    return "All managed policies were applied as expected."

payload = input()
ret = handler(payload, 'uselessinfo')
print(ret)
```

3 Example “SQS-λ”

```
import json
import os
import urllib.request

def lambda_handler(event, context):
    print("event : ", event)
    url = os.environ['web_url']
    print("url : ", url)

    charge_cash = event["Records"][0]["body"]
    charge_cash = json.loads(charge_cash)

    if charge_cash["charge_amount"]:
        try:
            charge_cash['auth'] = os.environ['auth']
            charge_cash['sqs_request'] = charge_cash.pop('charge_amount')
            data_json = json.dumps(charge_cash).encode('utf-8')
            req = urllib.request.Request(url, data=data_json,
                headers={'content-type': 'application/json'})
            res = urllib.request.urlopen(req)
            return "Message sent to EC2 server successfully!"

        except Exception as e:
            return "Error sending request to EC2 server"
    else:
        return "another request"

event = input()
lambda_handler(event, "something")
```

4 Example “Django”

```
1  from django.http import HttpRequest, HttpResponse
2
3  def operate_on_twos(request: HttpRequest) -> HttpResponse:
4      operator = request.GET["operator"]
5      r = abs(request.GET["range"]) + 1
6
7      if r == 0:
8          result = 0
9      else:
10         result = eval(f"(2 {operator} 2) * {r}")
11
12     return result
```

5 Example “Billion Laughs”

```
import xml.etree.ElementTree

billion_laughs = input()
billion_laughs = library.funs.sanitize(billion_laughs)
root = xml.etree.ElementTree.XML(billion_laughs)
```

```
import xml.etree.ElementTree

billion_laughs = input()
root = xml.etree.ElementTree.XML(billion_laughs)
```

List of publications

International conferences (first author)

Ignacio Tiraboschi, Tamara Rezk, Xavier Rival. "*Sound Symbolic Execution via Abstract Interpretation and Its Application to Security*". Verification, Model Checking, and Abstract Interpretation: 24th International Conference, VMCAI 2023, Boston, MA, USA.

Bibliographie

[Aguirre, 2017] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. "A Relational Logic for Higher-Order Programs". *Proc. ACM Program. Lang.* ICFP (2017), 21:1–21:29 (cit. on p. 58).

[Alatawi, 2017] Eman Alatawi, Harald Søndergaard, and Tim Miller. "Leveraging abstract interpretation for efficient dynamic symbolic execution". *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. Ed. by Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen. IEEE Computer Society, 2017, pp. 619–624 (cit. on p. 57).

[Asperti, 2008] Andrea Asperti and Cristian Armentano. "A Page in Number Theory". *J. Formaliz. Reason.* 1.1 (2008), pp. 1–23 (cit. on p. 53).

[Assaf, 2017] Mounir Assaf, David A. Naumann, Julien Signoles, Éric Totel, and Frédéric Tronel. "Hypercollecting semantics and its application to static analysis of information flow". *Symposium on Principles of Programming Languages (POPL)*. ACM, 2017, pp. 874–887 (cit. on pp. 8, 47, 48, 53–55, 58, 125).

[Backes, 2009] Michael Backes, Boris Köpf, and Andrey Rybalchenko. "Automatic Discovery and Quantification of Information Leaks". *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. 2009, pp. 141–153 (cit. on p. 58).

[Banerjee, 2008] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. "Expressive Declassification Policies and Modular Static Enforcement". *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society, 2008 (cit. on p. 58).

[Barthe, 2004] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. "Secure information flow by self-composition". *Proceedings of the IEEE Computer Security Foundations Workshop (CSF)*. Vol. 17. 2004, pp. 100–114 (cit. on pp. 54, 57, 58).

[Boyer, 1975] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT - a formal system for testing and debugging programs by symbolic execution". *Proceedings of the International Conference on Reliable Software 1975, Los Angeles, California, USA, April 21-23, 1975*. ACM, 1975, pp. 234–245 (cit. on pp. 4, 14, 57).

[Busse, 2022] Frank Busse, Pritam Gharat, Cristian Cadar, and Alastair F. Donaldson. "Combining static analysis error traces with dynamic symbolic execution (experience paper)". *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. , Virtual, South Korea, Association for Computing Machinery, 2022, pp. 568–579 (cit. on p. 119).

[Cadar, 2008] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224 (cit. on pp. 3, 5, 119, 120).

[Cadar, 2011] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, et al. "Symbolic execution for software testing in practice: preliminary assessment". *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 2011, pp. 1066–1071 (cit. on p. 58).

[Cadar, 2021] Cristian Cadar and Martin Nowack. "KLEE symbolic execution engine in 2019". *International Journal of Software Tools Technol. Transf.* (2021) (cit. on pp. 3, 57).

[Cadar, 2013] Cristian Cadar and Koushik Sen. "Symbolic execution for software testing: three decades later". *Communications of the ACM* (2013), pp. 82–90 (cit. on pp. 4, 58).

[Calcagno, 2011] Cristiano Calcagno and Dino Distefano. "Infer: An Automatic Program Verifier for Memory Safety of C Programs". *NASA Formal Methods*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 459–465 (cit. on p. 119).

[Ceara, 2010] Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet. “Taint Dependency Sequences: A Characterization of Insecure Execution Paths Based on Input-Sensitive Cause Sequences”. *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. 2010, pp. 371–380 (cit. on p. 120).

[Clarkson, 2008] Michael R. Clarkson and Fred B. Schneider. “Hyperproperties”. *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 2008, pp. 51–65 (cit. on pp. 3, 57).

[Corin, 2012] Ricardo Corin and Felipe Andrés Manzano. “Taint Analysis of Security Code in the KLEE Symbolic Execution Engine”. *Information and Communications Security*. Ed. by Tat Wing Chim and Tsz Hon Yuen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 264–275 (cit. on p. 120).

[Cousot, 1977] Patrick Cousot and Radhia Cousot. “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints”. *Symposium on Principles of Programming Languages (POPL)*. ACM, 1977, pp. 238–252 (cit. on pp. 4, 5, 27–29).

[Cousot, 1979] Patrick Cousot and Radhia Cousot. “Systematic Design of Program Analysis Frameworks”. *Symposium on Principles of Programming Languages (POPL)*. ACM, 1979 (cit. on p. 31).

[Cousot, 1978] Patrick Cousot and Nicolas Halbwachs. “Automatic discovery of linear restraints among variables of a program”. *Symposium on Principles of Programming Languages (POPL)*. ACM, 1978, pp. 84–97 (cit. on p. 28).

[Daniel, 2020] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level”. *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. 2020, pp. 1021–1038 (cit. on p. 58).

[Daniel, 2021] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE”. *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021 (cit. on p. 58).

[Daniel, 2022] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Reflections on the Experimental Evaluation of a Binary-Level Symbolic Analyzer for Spectre”. *Post-proceedings of the LASER@NDSS 2021*. The Internet Society, 2022 (cit. on p. 58).

[Darvas, 2005] Ádám Darvas, Reiner Hähnle, and David Sands. “A Theorem Proving Approach to Analysis of Secure Information Flow”. *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*. Ed. by Dieter Hutter and Markus Ullmann. Lecture Notes in Computer Science. Springer, 2005, pp. 193–209 (cit. on pp. 57, 58).

[Delmas, 2019] David Delmas and Antoine Miné. “Analysis of Software Patches Using Numerical Abstract Interpretation”. *Static Analysis Symposium (SAS)*. Ed. by Bor-Yuh Evan Chang. Vol. 11822. LNCS. Springer, 2019, pp. 225–246 (cit. on p. 48).

[Farina, 2019] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. “Relational Symbolic Execution”. *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*. Ed. by Ekaterina Komendantskaya. ACM, 2019, 10:1–10:14 (cit. on pp. 5, 15, 58).

[Fournet, 2011] Cédric Fournet, Jérémy Planul, and Tamara Rezk. “Information-flow types for homomorphic encryptions”. *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. 2011, pp. 351–360 (cit. on p. 58).

[Giacobazzi, 2004] Roberto Giacobazzi and Isabella Mastroeni. “Abstract non-interference: parameterizing non-interference by abstract interpretation”. *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. ACM, 2004, pp. 186–197 (cit. on p. 58).

[Goguen, 1982] Joseph A. Goguen and José Meseguer. “Security Policies and Security Models”. *IEEE Symposium on Security and Privacy, Oakland*. IEEE Computer Society, 1982, pp. 11–20 (cit. on pp. 2, 57).

[Huisman, 2006] M. Huisman, P. Worah, and K. Sunesen. “A temporal logic characterisation of observational determinism”. *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. 2006 (cit. on p. 58).

[Hunt, 2006] Sebastian Hunt and David Sands. “On flow-sensitive security types”. *Symposium on Principles of Programming Languages (POPL)*. ACM, 2006, pp. 79–90 (cit. on p. 8).

[Jeannet, 2009] Bertrand Jeannet and Antoine Miné. “Apron: A Library of Numerical Abstract Domains for Static Analysis”. *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 661–667 (cit. on p. 53).

[Jr, 1987] Hartley Rogers Jr. *Theory of recursive functions and effective computability (Reprint from 1967)*. MIT Press, 1987 (cit. on p. 53).

[King, 1976] James C. King. "Symbolic Execution and Program Testing". *Commununications of the ACM* 19.7 (1976), pp. 385–394 (cit. on pp. 4, 57).

[Kirchner, 2015] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. "Frama-C: A software analysis perspective". *Formal Aspects of Computing* 27 (3 2015) (cit. on p. 120).

[LLVM Projec,] LLVM Projec. *Clang* (cit. on p. 119).

[Merz, 2001] Stephan Merz. "Model Checking: A Tutorial Overview". *Modeling and Verification of Parallel Processes: 4th Summer School, MOVEP 2000 Nantes, France, June 19–23, 2000 Revised Tutorial Lectures*. Ed. by Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Dermot Ryan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 3–38 (cit. on p. 3).

[Meta, 2020] Meta. *Pysa: An open source static analysis tool to detect and prevent security issues in Python code*. 2020. URL: https://engineering.fb.com/2020/08/07/security/pysa/ (visited on 06/25/2024) (cit. on p. 63).

[Meta, 2023] Meta. *Pyre*. Version 0.9.18. 2023 (cit. on pp. 9, 63, 97, 124).

[Milushev, 2012] Dimiter Milushev, Wim Beck, and Dave Clarke. "Noninterference via Symbolic Execution". *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*. Lecture Notes in Computer Science. Springer, 2012, pp. 152–168 (cit. on pp. 15, 58).

[Moura, 2008] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340 (cit. on p. 53).

[Palikareva, 2016] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. "Shadow of a doubt: testing for divergences between software versions". *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 2016, pp. 1181–1192 (cit. on p. 15).

[Pandey, 2019] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. "Deferred concretization in symbolic execution via fuzzing". *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 228–238 (cit. on p. 125).

[Pasareanu, 2008] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, et al. "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software". *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*. Ed. by Barbara G. Ryder and Andreas Zeller. ACM, 2008, pp. 15–26 (cit. on p. 57).

[Pierce, 2018] Benjamin Pierce. *Commit closing bpo-34540 (CVE-2018-1000802)*. 2018. URL: https://github.com/python/cpython/pull/8985/commits/add531a1e55b0a739b0f42582f1c9747e5649ace (visited on 07/07/2024) (cit. on p. 110).

[Rice, 1953] H. Gordon Rice. "Classes of recursively enumerable sets and their decision problems". *Transactions of the American Mathematical Society* 74 (1953), pp. 358–366 (cit. on p. 3).

[Sabelfeld, 2003] Andrei Sabelfeld and Andrew C. Myers. "Language-based information-flow security". *IEEE Journal Sel. Areas Commun.* 21.1 (2003), pp. 5–19 (cit. on p. 58).

[Sabelfeld, 2005] Andrei Sabelfeld and David Sands. "Dimensions and Principles of Declassification". *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*. IEEE Computer Society, 2005, pp. 255–269 (cit. on p. 57).

[Santos, 2015] José Fragoso Santos, Thomas P. Jensen, Tamara Rezk, and Alan Schmitt. "Hybrid Typing of Secure Information Flow in a JavaScript-Like Language". *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Madrid, Spain, August 31 - September 1, 2015 Revised Selected Papers*. Lecture Notes in Computer Science. Springer, 2015, pp. 63–78 (cit. on p. 58).

[Schoepe, 2016] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce, and Andrei Sabelfeld. "Explicit secrecy: A policy for taint tracking". 2016 (cit. on pp. 119, 125).

[Späth, 2016] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. "SoK: XML Parser Vulnerabilities". *Workshop on Offensive Technologies*. 2016 (cit. on p. 114).

[Terauchi, 2005] Tachio Terauchi and Alexander Aiken. “Secure Information Flow as a Safety Problem”. *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*. Vol. 3672. Lecture Notes in Computer Science. Springer, 2005, pp. 352–367 (cit. on pp. 57, 58).

[Volpano, 1999] Dennis Volpano. “Safety versus Secrecy”. *Static Analysis*. Ed. by Agostino Cortesi and Gilberto Filé. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 303–311 (cit. on pp. 2, 75, 76, 119, 124).

RÉSUMÉ

Ce travail se concentre sur l'application de *l'analyse statique* pour la vérification ou la réfutation automatique de *propriétés de flux d'information*, en se concentrant sur l'interprétation abstraite et l'exécution symbolique. et l'exécution symbolique. Plus précisément, nous nous concentrons sur deux propriétés de flux d'informations : *non-interférence*, et *secret faible*.
La thèse est divisée en deux parties. Dans la première partie de la thèse, nous explorons une analyse statique basée sur des symboles qui communique avec une analyse des dépendances pour la vérification de l'intégrité du système. avec une analyse des dépendances pour la vérification de la non-interférence. Notre contribution est un domaine de produit réduit entre un domaine symbolique et un domaine de dépendances pour l'analyse solide de la non-interférence dans un langage impératif simple. Nous proposons également un produit réduit entre une exécution symbolique non relationnelle et des domaines numériques tels que les intervalles et les polyèdres convexes.
La deuxième partie consiste à explorer le concept d'un taint-tracker symbolique statique. Nous développons une sémantique formelle pour un traqueur de taches symbolique, ainsi que l'adaptation du secret faible pour les cas d'utilisation modernes. Ensuite, en supposant l'existence d'un taint-tracker sain mais imprécis, nous proposons une analyse combinée qui utilise à la fois le taint-tracker sain et le taint-tracker imprécis. analyse combinée qui utilise à la fois les traqueurs de taches sonores et symboliques. Enfin, nous instancions l'analyse combinée avec PYSA, un taint-tracker sonore développé par Meta, et notre outil PYSTA. Notre outil affine les résultats de l'analyseur de taches sonores, réduisant ainsi la charge de travail des développeurs pour l'examen manuel des alarmes. d'examiner manuellement les alarmes.

MOTS CLÉS

analyse statique, flux d'informations, interprétation abstraite, exécution symbolique

ABSTRACT

This work focuses on the application of *static analysis* for the automatic verification or refutation of *information flow properties*, focusing on abstract interpretation and symbolic execution. More specifically, we focus on two information flow properties: *noninterference*, and *weak secrecy*.
The thesis is split in two parts. In the first part of the thesis we explore a symbolically driven static analysis that communicates with a dependences analysis for the verification of noninterference. Our contribution is a reduced product domain between a symbolic domain and a dependences domain for the sound analysis of Noninterference in a simple imperative language. We also offer a reduced product between a non-relational symbolic execution and numerical domains such as intervals and convex polyhedra.
The second part consists on exploring the concept of a static symbolic taint-tracker. We develop a formal semantics for a symbolic taint-tracker, together with the adaptation of weak secrecy for modern use-cases. Then, by assuming the existence of a sound-but-imprecise taint-tracker, we contribute a combined analysis that uses both the sound and the symbolic taint-trackers. Finally, we instantiate the combined analysis with PYSA, a sound taint-tracker developed by Meta, and our tool PYSTA. Our tool refines the output of the sound taint-tracker, lowering the load of developers to manually review alarms.

KEYWORDS

static analysis, information flow, abstract interpretation, symbolic execution