Norwegian University of
Science and Technology

Simple static analysis techniques for Java

Using latent meaning to find security bugs

Edvard Kristoffer Karlsen

Master of Science in Informatics
Submission date: September 2015
Supervisor: Torbjørn Skramstad, IDI
Co-supervisor: Bjarte M. Østvold, Norsk Regnesentral

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description

The problem was given by Bjarte M. Østvold at the Norwegian Computing Center.

Software security is difficult. Every week there are reports of new and serious vulnerabilities that affect our computers or phones. Many of these vulnerabilities can be traced back to relatively simple and local causes in programs. Often the cause is that developers are unfamiliar with APIs and use these in an unintended way, leading to bugs.

Static program analysis dates back almost 40 years [17] and in theory such tools can discover many of the bugs introduced by developers. However, some issues with static program analysis remain. First, such analysis is often domain-neutral, meaning that the analysis does not know about the domain that the program under analysis is concerned with, for example, security. This means that either bugs are not reported, or they are reported together with lots of domain-irrelevant bugs, risking that the critical domain-related bugs escape developers' attention. Second, static program analysis theory is quite complex, making it harder to extend and tailor it to a particular domain.

The task of the student is to find *simple* program analysis techniques that can be used to find security bugs in Java programs, that is, techniques that are domain-specific. Much previous work has focused on being clever: find all problems (in some restricted class), find only real problems (no false positives). The goal of the this work is instead to be pragmatic: is there a simple static program analysis that can be used to find security bugs?

Abstract

Source code is rich with signs carrying meaning that is incomprehensible to a compiler, but important to the human programmer. For instance, a compiler does not understand that a variable named `privateKey` contains confidential data and therefore must be treated with extra care, or that an array populated by a cryptographically secure random number generator has properties that set it apart from other arrays. I present two static analyses that explicitly model such *latent* meaning, and use it to find bugs. Both analyses are simple; my aim is not to beat the precision of state-of-the-art techniques, but rather to argue that much can be done using simple techniques. To support this claim, I demonstrate the effectiveness of both analyses on test cases from a well-known test suite and a selection of other examples. Further, I argue that the analyses generalise to applications beyond those I investigate. I have implemented the analyses in a proof-of-concept tool, which I contribute as free and open source software.

Sammendrag

Kildekode er fylt av tegn hvis mening er uforståelig for en kompilator, men essensiell for mennesker som jobber med koden. En kompilator kan for eksempel ikke forstå at en variabel kalt `privatNøkkel` inneholder konfidensiell data, og derfor må behandles spesielt forsiktig, eller at en tabell fylt med verdier fra en kryptografisk sikker slumptallsgenerator har egenskaper som skiller den fra andre tabeller. I denne oppgaven presenterer jeg to statiske programanalyser som eksplisitt modellerer slik *latent* mening, og bruker denne for å finne feil. Begge analysene er enkle; målet mitt er ikke å overgå presisjonen til ledende teknikker fra litteraturen, men å argumentere for at man kan komme langt med enkle teknikker. For å støtte denne påstanden viser jeg at begge analysene er effektive på tester fra et velkjent testsett og et utvalg av andre eksempler. Videre argumenterer jeg for at analysene har anvendelser utenom de jeg undersøker. Jeg har implementert analysene i en prototype, som jeg publiserer som fri og åpen programvare.

Acknowledgements

I'll keep this short and free of additional cliché.

Torbjørn and Bjarte were always available for discussion and detailed feedback. It is a rare privilege to have such positive, creative, and encouraging supervisors. Thank you.

I'm grateful to my parents Kirsti and Wilhelm, my sister Ingrid, and my aunt Gerd Hilde for their love and support.

Lastly, I thank my closest friends, who have supported and encouraged me throughout my work with this thesis. Your friendship is invaluable.

Contents

1 Introduction **1**
1.1 Research hypothesis 3
1.2 Contributions 4
1.3 Structure of the thesis 4

2 Background **7**
2.1 Core program analysis 8
2.2 Combined techniques 10
2.3 Terminology 11

3 Type-centered analysis **15**
3.1 Domain rules as typing problems 15
3.2 Signs and latent meaning 18
3.3 Formalising the type-centered analysis 19
3.4 Latent meaning database 24
3.5 Domain rules encoded for the type-centered analysis 30

4 Solving typing problems **35**
4.1 Preliminaries 36
4.2 Typing algorithm 37
4.3 Correctness proof 38
4.4 Translating from a typing problem to a problem instance 41

5 Flow-centered analysis **43**
5.1 Introduction 43
5.2 Computing with critical values 45
5.3 Translating Java methods into the flow language 51
5.4 The analysis proper 53

6 Implementation of the proof-of-concept tool **57**
6.1 Overview of the system 57
6.2 Re-running the experimental validation 58

7 Experimental validation **59**

7.1 Experimental validation strategy 59
7.2 Experimental validation for the type-centered analysis 61
7.3 Experimental validation for the flow-centered analysis 68

8 Related work **75**
8.1 Bugs as deviant behaviour 75
8.2 Pluggable type systems 76
8.3 FindBugs 77
8.4 Naming bugs 78

9 Discussion and conclusion **79**
9.1 Discussion of the type-centered analysis 79
9.2 Discussion of the flow-centered analysis 81
9.3 Relating the flow-centered analysis and the type-centered analysis 82
9.4 Ideas for improving the precision of the type-centered analysis . . 83
9.5 Ideas for future conceptual work 85
9.6 Threats to validity 86
9.7 Conclusion 87

References **89**

CHAPTER 1

Introduction

To program is to tell a computer how to do a task. To accomplish this, programmers translate information from a domain[1] into another representation, programming language. A typical program is rich with representations of concrete or abstract entities belonging to domains existing outside of the computer.[2]

Domains have *domain rules*, which speak of what is correct within the scope of a domain. These are examples of rules from different domains:

- In a zero-based budgeting system, the balances of all accounts must sum to zero.
- The Java sparrow is neither mammal nor fish.
- The equation $a^n + b^n = c^n$ has no solution for a, b, c in $\mathbb{Z}$ when $n \geq 3$.

Some programming languages provide features that let us 'encode' certain types of domain rules so that if we violate a rule in source code, a compiler will flag it as an error. In a way, we can make it so that violations of the domain rule become semantic errors in the programming language.

Let us consider an example of how domain rules can be encoded so they are enforced by a compiler. Assume we are designing an application where we need to represent a taxonomy of animals. In Java, it is natural to encode such a taxonomy as a class hierarchy:

[1] Throughout the thesis, I use *domain* in the lenient sense given in the Collins English dictionary: 'a field or scope of knowledge or activity' [1].

[2] For an example of the distinction between a representation and that which is represented, consider the distinction between a class `JetEngine` in the source code of a flight simulator and an engine on an actual jet plane.

```
class Animal {}
class Bird extends Animal {}
...
class JavaSparrow extends Bird {}
class Mammal extends Animal {}
class Cat extends Mammal {}
class Dog extends Mammal {}
```

Using this representation, we have effectively encoded the domain rule that a Java sparrow is a bird (and neither a mammal nor a fish), so that the compiler will alert us if we violate it. For instance, if we try to compile the snippet

```
List<Mammal> mammals = new LinkedList<>();
mammals.add(new Cat());
mammals.add(new Dog());
mammals.add(new JavaSparrow());
```

the compiler returns an error message where the gist is that `JavaSparrow cannot be converted to Mammal`.

This example above illustrates how a *type system* allows us to encode certain kinds of domain rules about relations between types of entities and about how types of entities may interact. Type systems are *one* example of a feature that allows us to encode domain rules so they are enforced by a compiler. Sophisticated type systems, such as the *dependent* type systems of Agda [42] and Idris [10], allow programmers to encode very complex domain rules to be enforced by a compiler. Another example of a feature for encoding domain rules to be enforced at compile time is *static asserts*, as seen in modern C++ versions [33].

In common industry practice, however, few domain rules are encoded so that they are enforced by the compiler. There are several reasons for this. First, rules are often complicated, and it may be infeasible to specify them precisely. Second, some types of rules are practically impossible or extremely cumbersome to encode using the mechanism of a specific programming language.

Interestingly though, typical source code is rich with clues about domain rules not enforced by a compiler—clues that show up in names, comments, and other source code constructs. When human programmers read a snippet of source code they can use their domain knowledge and look to these clues to infer

- what domain rules are relevant for the snippet, and
- whether these rules are obeyed.[3]

I will say that a programmer studying code like this is uncovering *latent domain rule violations*.[4]

[3] There is nothing special about the process I describe here. This is essentially a careful way to describe what a programmer does when she reviews a program, looking for bugs.

[4] Here, I mean by *latent* that the violation 'lays latently' (or hidden) in clues in the source code—visible to a human expert but not to a compiler.

To better illustrate these ideas, let us look to an example. Consider the following method:

```
boolean isCorrectPassword(String providedPasswordRaw) {
  Account account = findAccount(this.username);
  /* ... */
  syslog.write("checking password for " + this.username +
               ". provided: " + providedPasswordRaw);
  /* ... */
}
```

How could a programmer study this method to figure out which domain rules are relevant for the method, and find potential latent domain rule violations of these rules? One possible line of reasoning goes as follows:

1. From the identifier names and the method name, one may infer that the method is security-critical.
2. In any security-related domain, the following domain rule must be obeyed: confidential data must not be exposed. This is a latent domain rule in the method, because it is not explicitly enforced by the compiler.
3. In particular, passwords are confidential. Therefore `providedPassword-Raw` should not be exposed.
4. However, writing `providedPasswordRaw` to the system log is (almost certainly) unwanted exposure. In conclusion, the method contains a latent domain rule violation.

My goal in this thesis is to design simple static analysis techniques that (largely) mimic this kind of reasoning.

1.1 Research hypothesis

A research project should be based around a falsifiable hypothesis. Further, the hypothesis should be an interesting proposition to which 'reasonable people can disagree'.

I will argue for the following hypothesis:

> **Simple intraprocedural static analyses techniques are useful for finding interesting and relevant latent domain rule violations.**

Without further qualification, this hypothesis may appear obviously true. I add two qualifications, however. First, by *simple [techniques]*, I mean that the analysis techniques should be simpler than related techniques in the literature. Second, by *[useful for finding] interesting and relevant [rules]*, I mean that the analysis techniques should be able to find violations of a range of latent domain rule violations seen in real-world code.

1.2 Contributions

The contributions of this thesis are:

1. The *type-centered analysis*, a simple static analysis built around the idea that checking whether a method violates a latent domain rule can be cast as a *typing problem*. I develop this idea and describe this analysis in Chapter 3 and Chapter 4.
2. The *flow-centered analysis*, a simple static analysis designed to assert flow-related latent domain rules, by finding *disappearing critical values*. I motivate and describe this analysis in Chapter 5.
3. Descriptions of how to specialise the analyses to assert specific security-relevant latent domain rules, such as asserting that no confidential information leaks to an insecure output channel. The primary descriptions of these specialisations are given in Section 3.5.1 and Section 3.5.2. I describe further details in Chapter 7, the experimental validation chapter.
4. A proof-of-concept implementation of the analyses, written in Scala. I contribute this implementation as free and open-source software under the MIT license. In Chapter 6, I give an overview of this implementation.
5. A discussion of ideas for applying data mining techniques to automatically learn relevant latent domain rules from a software corpus. I discuss these ideas in Section 9.5.

1.3 Structure of the thesis

The thesis may be seen as comprising three parts. The first part, *the exposition*, consists of this introductory chapter and Chapter 2. Here, I introduce the problem at hand, state my research hypothesis, and survey background literature. The second part, *the description of the solution*, consists of chapters 3 to 6. Here, I describe two simple static analyses and a proof-of-concept implementation of these. The third and final part, *the evaluation*, consists of chapters 7 to 9. Here, I begin with an experimental validation of the analyses. Then, I compare the analyses to closely related work. Further, I discuss the analyses, relate them to each other, and present ideas for future work. Finally, I discuss threats to validity and give a concluding argument.

Figure 1.1 provides another view of the thesis' structure. This figure illustrates the crucial subcomponents of the thesis (conceptual ideas, analysis techniques, and validation efforts) and the most important dependencies between these. In this figure, orange nodes (■) denote conceptual ideas, golden nodes (■) denote analysis techniques, and green nodes (■) denote validation efforts.

Note that the three longest chapters (chapters 3, 5, and 7) end in a summary with forward references, to further clarify the thesis' structure.

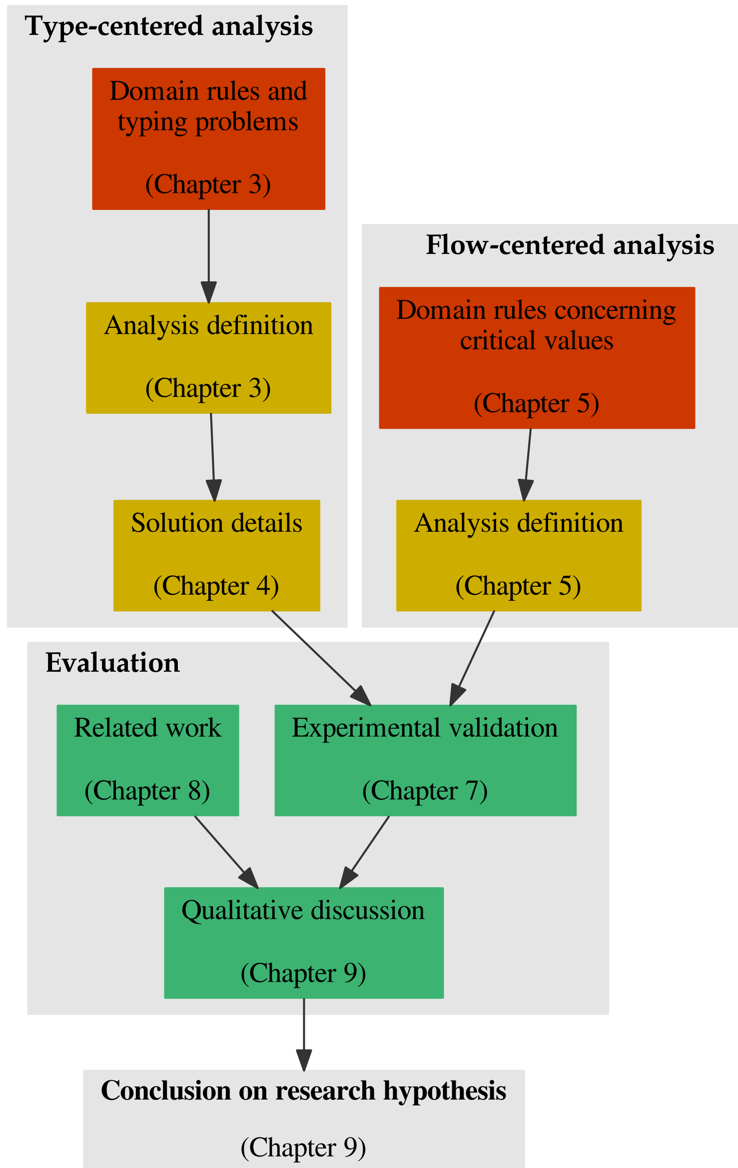

Figure 1.1: Roadmap for the thesis.

CHAPTER 2

Background

What is the research context of this thesis? Broadly speaking, the answer is *program analysis*, as I rely heavily on theory and tools developed by program analysis researchers. I should qualify this answer, however, as the term program analysis has strong connotations to specific formal techniques for statically optimising or verifying properties of programs, and it would be imprecise to constrict myself only to this interpretation. Rather, the relevant context is all rigorous[1] analysis of programs, aiming to answer interesting questions about programs, with the goal of improving the quality of these programs, or in other ways benefit software engineers. To better illustrate this context, I make a distinction between *core* program analysis and *combined* techniques—which apply theory from core program analysis in combination with techniques such as machine learning. I should caution, though, that this distinction is somewhat artificial; there is no clear distinction between core and combined techniques, and much research falls somewhere in the middle.

One crucial aim in this text (the whole thesis) is to discuss my contributions in context of related research. In this chapter, I lay the foundation of my effort towards this goal. Concretely, in Section 2.1, I give an overview of important work in the core program analysis field, and after that, in Section 2.2, I discuss some of the combined techniques that have surfaced the last 20 years.[2] However, this chapter is only a foundation, and I delay discussing the works most relevant to my thesis until Chapter 8. The rationale for this is that I want to be able to refer to the details of the techniques I present, without resorting to forward references.

I have a secondary, minor goal with the present chapter: to introduce and clarify some general program analysis terminology, which I use throughout the

[1]Here I mean by rigorous that analysis is or could be done mechanically.

[2]My aim with these two sections is not to lay the groundwork for a (bad) reference text, but to show that I have a good overview of the most important work in the field.

rest of the thesis. This is the topic of Section 2.3.

2.1 Core program analysis

Fundamentally, program analysis is about examining the dynamic behavior of programs. In the traditional view, it concerns two main problems: First, *program optimisation*, that is, how to rewrite programs so that they do the same task, but perform better with respect to some metric. Second, *program verification*, which is about checking whether programs satisfy certain properties, most commonly whether a program behaves in accordance with a given specification.

Program analysis has a long history; engineers and scientists have been interested in optimising programs, verifying the correctness of programs, and systematically finding bugs, since the first programming languages were implemented.[3] With that said, it concretised as a self-standing subfield in the late seventies, when Cousot and Cousot and others formalised its principal problems and described the first systematic techniques for attacking them [17, 18]. The theory developed in this period are mainstays in the field today.[4] In addition to the Cousots, one should, at the least, mention the seminal contributions of Hoare [27], Floyd [22], and Dijkstra [20].

In the next sections, I discuss program optimisation and verification, in turn, with focus on recent research trends. Following that, I describe various *representations* that are important in program analysis.

Program optimisation

Program optimisation concerns techniques for transforming programs into programs that have the same behavior but perform better with respect to some relevant metric. The classic metric to optimise for is (reduced) runtime, but there are other important metrics, such as executable size, memory footprint, and energy consumption.

Much of the work in modern program analysis concerns how to do standard optimisations techniques—such as dead code elimination, common subexpression optimisation, constant propagation, and procedure inlining[5]—within increasingly challenging contexts. Examples of techniques that allow more advanced optimisation in challenging contexts are alias analysis [13] and shape analysis [34].

Traditionally, research in program optimisation has been about purely imperative programming languages, such as C, or about languages whose semantics are largely similar to purely imperative languages, such as C++ and Java. Further,

[3]One example of early work on program optimisation, is the FORTRAN I compiler, developed at IBM between 1954 and 1957 [45]. To the surprise of its authors and users, the compiler often output programs that ran faster than hand-crafted assembly [45], and 'in some cases, [...] produced code which was so good that users thought it was wrong' [3].

[4]This is evident in modern reference texts, such as Nielson et al.'s well-known monograph [41].

[5]These techniques are discussed in any reference text, such as the 'Dragon book' [2].

it has typically focused on statically typed languages. In the last twenty years, however, there has been substantial work done on optimisation for non-imperative languages, such as Haskell and ML, and dynamically typed languages, such as JavaScript, Python, and Scheme.

Functional programming languages present unique challenges for optimisation. For one thing, it is harder to do interprocedural control flow analysis—a necessary first step for much aggressive optimisation—for these languages, because of the extensive use of function calls, and the blurred line between functions and data. Shivers presented the first powerful techniques for doing interprocedural control flow analysis for higher-order languages, the CFA family of analyses [57].[6] Because of the extensive use of function calls in functional programming, aggressive and precise inlining is necessary for efficient execution. An example of an advanced inliner is the one used in the Glasgow Haskell Compiler [48].

Much recent work in program optimisation has been about techniques for optimising programs by exploiting opportunities for parallellisation. (This, of course, echoes a general trend in computer science.) For example, Radoi et al. 'present an approach for automatic translation of sequential, imperative code into a parallel MapReduce framework' [50].

Program verification

Program verification is about asserting the truth of certain properties of a program.[7] Some textbook examples of such properties are i) whether a concurrent program may deadlock, and ii) whether a C program may crash with a *segfault*. Key techniques in program verification are model checking [16, 65], abstract interpretation [18], Hoare logic [27], and separation logic [53].

Many of the domain rules I study later in the thesis may be seen as security-related verification problems. For instance, I look at one domain rule which, fundamentally, concerns how to 'correctly treat' cryptographically secure random numbers, and one domain rule which concerns how to protect confidential information so that it does not 'leak'. For the latter topic, relevant literature is grouped under the umbrella term *language-based information flow security*, on which Sabelfeld and Myers have written an extensive survey [55].

Representations

An essential aspect of attacking problems in program optimisation and verification is developing useful custom languages for representing programs.

[6] These techniques were later refined by Might [40], under Shiver's supervision, and are still being improved. See, for instance, Prabhu et al.'s work on implementing CFA analyses on GPUs [49].

[7] Note that this is just one attempt at defining *program verification*. Researchers, unfortunately, use different, non-compatible definitions of the term. For instance, Almeida et al. use a rather specific definition: 'program verification is the area of computer science that studies mathematical methods for checking that a program conforms to its specification' [4]. Boyer and Moore define it more broadly; in their words '[program verification] is the use of formal, mathematical techniques to debug software and software specifications' [8].

When discussing such representations, it is helpful to distinguish between properties of representations, and concrete representations designed to model the essence of a specific programming language.

In the first category, I should mention two properties of representations: First, Static Single Assignment (SSA) form, which is extensively used in modern industry compilers.[8] When Cytron et al. introduced this representation [19], it immediately made many classic optimisation techniques much easier to reason about and implement. Second, continuation passing style (CPS) representations are commonly used in compilers for functional programming languages [58], and when reasoning about the semantics of such languages.

In the second category, I mention two representation languages especially relevant to my thesis: First, Featherweight Java [32], which has made it much easier to reason about intricate aspects of Java (which, among other things, has a complex and fragile type system). Further, I mention Jimple [64], the intermediate representation I use for implementing (parts of) the analyses I present in this thesis. Jimple is custom-tailored to facilitate program analysis for Java, providing an alternative to analysing the stack-based and largely untyped JVM byte code.[9]

2.2 Combined techniques

In the last two decades, researchers have started using theory from core program analysis in tandem with techniques from other fields, to attack problems that are largely unrelated to the classical problems of optimisation or verification.

One common idea is to use theory from core program analysis as a component in empirical study of real-world programs. Typically, researchers study a large *corpus* of programs, meant to be a representative sample of programs in the real world.

Such corpus-based investigations often have a zoological nature; the aim is to describe some aspects of programs, and to create a taxonomy or some other descriptive model. One example is Høst and Østvold's work on uncovering *name patterns* in Java code [29], where they construct a mapping from common method names (such as `get`, `create`, and `toString`) to so-called semantic profiles.[10] For another example, Temporo et al. performed an empirical study of how programmers use inheritance in Java code [61].

Importantly, learning from a corpus is often only the first step towards a larger goal, namely to say something about or relevant to single programs, or other units of code, using knowledge learned from the corpus. Here, there are two main approaches, which I term *corpus-based bug detection* and *corpus-based advice generation*.

[8] SSA form is, for instance, used in gcc and in the LLVM backend, and in the compilers of HotSpot, Dalvik, and Android ART, the primary Java virtual machines.

[9] If one wants to analyse raw JVM byte code, however, ASM [12] is excellent for the job.

[10] Note that I return to compare my work to Høst and Østvold's in greater detail in Section 8.4.

Corpus-based bug detection exploits the idea that statistically improbable code is likely to be buggy. An example of corpus-based bug detection is Reiss' work on finding unusual code [52]. Reiss uses a two-step process along the lines I describe here: First, he extracts common syntactic patterns from a corpus of software. Second, he identifies code that contains syntactic patterns not common in the corpus. Another example is Høst and Østvold's work on *naming bugs* [30], where they augment their phrase book [29] with a technique for identifying methods with statistically improbable names. Høst and Østvold's technique for identifying naming bugs was implemented in a plug-in for the Eclipse IDE [35].

In corpus-based advice generation, the aim is not to identify bugs, but rather to give a programmer suggestions on how to improve or finish a unit of code. An impressive example is the JSNice 'prediction engine' [51], which can suggest names and type annotations for variables in a JavaScript function. JSNice relies on a conditional random field model [36], which the authors train using a corpus sampled from GitHub. Many techniques for corpus-based bug detection extend naturally to corpus-based advice generation; for instance, Høst and Østvold repurpose the metric they use to identify naming bugs to also suggest naming bug fixes [30].

Researcher also use corpus-based techniques to validate or refine theories about software. For instance, Gil and Maman present a catalogue of *micro patterns* (cf. design patterns [25]) in Java code, and empirically investigate their catalogue using a corpus [26].

2.3 Terminology

In this section, I make precise some program analysis terminology that will prove useful in later chapters.

2.3.1 Static vs. dynamic analysis

A *static* analysis investigates a program by looking at its source or binary code. In particular, it tries to determine its dynamic behavior, i.e. what it does at runtime, without running it.

Conversely, a *dynamic* analysis investigates a program as it runs. Dynamic analysis is especially important in just-in-time compilers such as HotSpot [46], TraceMonkey [24], and pypy [7], which derive much of their impressive performance from being able to aggressively optimise selected *hot* parts of programs, using dynamic information about typing and path frequency.

2.3.2 Intraprocedural vs. interprocedural analysis

An *intraprocedural* analysis looks only at the body of one procedure (or method) at a time. Conversely, an *interprocedural* analysis looks at the body of several

procedures (or methods) at a time. In general, interprocedural analyses are much more complex to specify and more computationally demanding.

The canonical example of interprocedural analysis is inlining, where one seeks to strategically substitute select procedure (method) calls with the bodies of their callees, so as to reduce the overhead inherent in procedure (method) calls.

Many optimisations that are typically thought of as intraprocedural analyses can be extended to run interprocedurally. For instance, one can do interprocedural dead code elimination or interprocedural constant propagation.

2.3.3 Soundness, completeness, and related terms

The convention in program verification is to say that a static analysis checks whether some property θ holds for a program. From this starting point, one can define what it means for a static analysis $\hat{\theta}$ to be *sound* and *complete*, and what it means for it to return a *true* or *false positive* or *negative*. My goal with this section is to make precise all these terms. However, I shall define them in context of static analyses that aim to find sets of bugs in programs, rather than with respect to single properties. Such analyses, I term *bug-finding static analysis*. The following things can be said about virtually any such analysis:

1. The analysis is concerned with some set $\mathcal{P}$ of programs.
2. The analysis is concerned with some set $\mathcal{B}$ of bugs.
3. It is possible to think of a hypothetical function $B : \mathcal{P} \to \{\mathcal{B}\}$ that could 'tell' which bugs exist in a given program.
4. The analysis can seen as a function $\hat{B}$ trying to approximate B.

From this, I define the terms mentioned in the start of the section as follows:

Soundness: *All reported bugs are real.* An analysis is *sound* if it never overapproximates the set of bugs in any given program. That is, if $\hat{B}(p) \subseteq B(p)$, for any program p. Conversely, if there exist a program p and a bug b such that $b \in \hat{B}(p)$ but $b \notin B(p)$, then $\hat{B}$ is *unsound*.

Completeness: *All bugs are reported.* An analysis is *complete* if it never underapproximates the set of bugs in any given program. That is, $B(p) \subseteq \hat{B}(p)$, for any program p. Conversely, if there exist a program p and a bug b such that $b \in B(p)$ but $b \notin \hat{B}(p)$, then $\hat{B}$ is *incomplete*.

True positive: Given a program p, a bug b, and a static analysis $\hat{B}$ approximating B, b is a *true positive* if $b \in \hat{B}(p)$ and $b \in B(p)$.

False positive: Given a program p, a bug b, and a static analysis $\hat{B}$ approximating B, b is a *false positive* if $b \in \hat{B}(p)$ but $b \notin B(p)$.

	Bug is real	**Bug is not real**
Analysis reports bug	True positive	False positive
Analysis does not report bug	False negative	True negative

Table 2.1: Intuitive illustration of the terms true positive, false positive, true negative, and false negative.

	$\mathbf{b} \in \mathbf{B}(p)$	$\mathbf{b} \notin \mathbf{B}(p)$
$\mathbf{b} \in \hat{\mathbf{B}}(p)$	True positive	False positive
$\mathbf{b} \notin \hat{\mathbf{B}}(p)$	False negative	True negative

Table 2.2: Illustration of the terms true positive, false positive, true negative, and false negative using B and $\hat{B}$.

True negative: Given a program p, a bug b, and a static analysis $\hat{B}$ approximating B, b is a *true negative* if $b \notin \hat{B}(p)$ and $b \notin B(p)$.

False negative: Given a program p, a bug b, and a static analysis $\hat{B}$ approximating B, b is a *false negative* if $b \notin \hat{B}(p)$ but $b \in B(p)$.

Table 2.1 gives a colloquial illustration of the terms true positive, false positive, true negative, and false negative. Table 2.2 illustrates these terms using B and $\hat{B}$.

The ideal bug-finding static analysis would be *sound and complete*, in which case one would have $\hat{B}(p) = B(p)$ for any program p, and no false negatives or false positives. This is usually impossible, however, except for very special choices of $\mathcal{P}$. (First of all because of the halting problem, as first demonstrated by Turing [62].)

CHAPTER 3

Type-centered analysis

In this chapter, I describe the type-centered analysis, the first of the two analyses I present in the thesis.

The chapter is structured as follows: First, in Section 3.1, I present the idea that checking whether a method violates a domain rule can be cast as a *typing problem*, and show examples of how to check whether a method violates a domain rule. Second, in Section 3.2, I explain how source code can be seen as an assemblage of *signs*, carrying *latent meaning*. Third, in Section 3.3, I formalise the analysis, and give a more formal example of how to find a latent domain rule violation. Fourth, in Section 3.4, I formalise *latent meaning database*, another component of the analysis. Here, I also describe an intermediate representation. Finally, in Section 3.5, I describe how to *specialise* the type-centered analysis to find violations of two security-relevant domain rules.

3.1 Domain rules as typing problems

As stated in the thesis' introduction, my goal is to design simple static analysis techniques that can find latent domain rule violations in Java code. However, I do not aim to discover every possible latent domain rule violation, as domain rules can be extremely complex.[1]

Instead of aiming to design analyses capable of finding any latent domain rule violation, I limit my scope to techniques covering a significant number of domain rules that

1. are relevant to real-world code, and
2. are feasible to attack with automatic analysis techniques.

[1]Consider the domain rule: all software used in this system must halt in finite time. This rule, like many others, is clearly impossible to assert for arbitrary Java code, no matter how clever you design a static analysis [62].

Within scope of the type-centered analysis, I focus only on domain rules that can be asserted as *typing problems*. Within scope of the flow-centered analysis, which I describe in Chapter 5, I focus on other types of domain rules.

Introductory examples

To make better sense of the idea that checking whether a method violates a domain rule can be cast as a typing problem, I now give two examples, meant to illustrate

- how a latent domain rule violation manifests as a typing error, and
- how, if a method does *not* contain a latent domain rule violation, there is no typing error.

Consider the following method:

```
public String createSecureChallenge() {
  java.util.Random rng = new java.util.Random();
  int x = rng.nextInt();
  return Integer.toHexString(x);
}
```

Let us assume we are to check whether the method obeys the domain rule:

r_{cs}: Any *secure challenge* must stem from a cryptographically secure random source.

First, before we attack this as a typing problem, observe that the rule *is* violated in the snippet:

1. The name `createSecureChallenge` implies that the method's return variable is a cryptographically secure challenge, and therefore it should stem from a cryptographically secure random source.
2. Yet, the method's actual return value stems from an instance of `java.util.Random`, a random source that does not output cryptographically secure random values.

Crucially, the evidence of the latent domain rule violation appears as a contradiction: apparently, the return value has to be and not be cryptographically secure at the same time.

Now, say we introduce what I term a *type set* consisting of two types: the type of values and variables that are cryptographically secure challenges, and the type of those that are not. Let us denote these cs and ¬cs, respectively. Further, assume that these types are distinct, in the sense that a value or variable must be of type cs or type ¬cs, but not of both. We may then rephrase the propositions above as follows:

1. The method's return value should have type cs. This is implied by the method name `createSecureChallenge`.
2. The method's return value should have type ¬cs. This is because the return value stems from an instance of `java.util.Random`, a random source that *does not* output cryptographically secure random values.

Let us call these propositions *typing constraints*. Further, let us say that these constraints are *satisfiable* if we can associate with the return value either cs or ¬cs, so that both constraints become true. This is, of course, impossible: By constraint one, the return value must have type cs. Yet, by constraint two, the return value must have type ¬cs. Here, the contradiction we reached above has become explicit as an *error* in the typing problem. The typing problem is *unsatisfiable*.

Let us now look at an example where the same domain rule is not violated. In this instance, the equivalent typing problem is satisfiable.

Consider the following method, which is equal to the preceding one save for using `java.security.SecureRandom`, a cryptographically secure RNG, instead of `java.util.Random`.

```
public String createSecureChallenge() {
  java.security.SecureRandom rng =
    new java.security.SecureRandom();
  int x = rng.nextInt();
  return Integer.toHexString(x);
}
```

With this method, I associate the following typing constraints:

1. The method's return value should have type cs. This is implied by the method name, `createSecureChallenge`.
2. The method's return value should (also) have type cs. This is because the return value stems from `SecureRandom`, a random number generator that *does* output cryptographically secure random values.

Here, there is no contradictory requirement to the return value. We can choose type cs and satisfy both constraints. The typing problem is satisfiable, and the method does not violate r_{cs}.

Fundamental idea

The main idea I want to convey with these examples is:

> Checking whether a method violates a latent domain rule violation can (for certain domain rules) be done by checking whether a 'corresponding' typing problem is satisfiable.

This idea is illustrated in Figure 3.1. Once again, I restrict the scope of the type-centered analysis to domain rules that can be checked using this technique.

My aim with the preceding examples is to illustrate the idea that checking whether a method violates a domain rule can be cast as a typing problem. I continue by discussing *signs* in source code, and specifically how signs can be seen as carrying *latent meaning*, which can be used to generate typing constraints.

Latent domain rule is not violated	⇒	Typing problem is satisfiable
Latent domain rule is violated	⇒	Typing problem is not satisfiable

Figure 3.1: Correspondence between latent domain rule violations and typing problems.

3.2 Signs and latent meaning

Source code may, like any type of text, be seen as an 'assemblage of signs' [14] forming a message.[2]

Some signs in source code carry more meaning than others. Among these are common method names, such as `toString`; calls to well-known library routines, such as `Collections.sort`; and stereotypical code segments. Researchers in the psychology of programming call such signs *beacons* [11], and argue that they are essential in the process of program comprehension.

Common method names are one type of beacon. For instance, Java programmers will see the names `toString`, `hashCode`, and `equals` as beacons. These names refer to methods defined on the common superclass `Object`, and their meanings are taught in every introductory Java course.

Calls to library methods that implement important, specific functionality, are another type of beacon. For example, consider the snippet:

```
byte[] plaintext = ...;
cipher.update(plaintext, ciphertext);
```

Here, we may see the call `cipher.update` as a beacon signifying that this code is about encryption or decryption. Another beacon is the variable name `plaintext`.

A third type of beacon is stereotypical combinations of source code constructs, such as loops and assignments sequences. One example is the standard **`for`** loop:

```
for (int i = 0; i < n; i++)
  ...;
```

This is a beacon in all statically typed curly-brace languages, signifying that the code is about iteration.

Another example is a *swap* of two array elements:

[2]In the case of source code, we may see the sender(s) of the message as the human programmer(s) or computer programs writing it, and the recipients of the message as the compilers and other programs that process source code, and, of course, the humans that maintain it.

```
int temp = arr[a];
arr[a] = arr[b];
arr[b] = temp;
```

Programmers will see this as a sign that the code is about reordering the array in some way. For instance, the code may be part of a sorting routine.

Latent meaning of a sign

I am interested only in those properties of a sign that can help determine whether a method contains a latent domain rule violation. This, I define as the *latent meaning* of a sign.

Importantly, I always speak of the latent meaning of sign *with respect to* a specific domain rule. Let me give an example to better illustrate this. Consider a method with name `createSecureChallenge`. With respect to the domain rule r_{cs}, the latent meaning of this method name is that the method's return value is a cryptographically secure random challenge. On the other hand, with respect to many other domain rules, this method name has no latent meaning. For instance, it has no latent meaning with respect to, say, the domain rule: all user-provided input must be checked against a white list before it is written to an output channel.

3.3 Formalising the type-centered analysis

In this section, I formalise most aspects of the type-centered analysis, and show how it allows analysis of many different kinds of domain rules.

The structure of the upcoming sections is as follows: First, in Section 3.3.1, I explain a crucial distinction between the type-centered analysis per se and domain rules encoded for the analysis. Second, in Section 3.3.2, I give a set of definitions, which define the core of the type-centered analysis. Third, in Section 3.3.3, I give an example meant to illustrate the definitions in Section 3.3.2. To simplify the presentation, I delay formalising the details of what I term *latent meaning databases* until Section 3.4.

3.3.1 The type-centered analysis is an abstract framework

First, a crucial distinction: The *type-centered analysis* per se is a framework. It is not tied to a specific domain rule. Rather, it must be *specialised* to create analyses for specific domain rules. Concretely, to specialise the analysis one must provide an *encoding* for a specific domain rule.[3] For example, to find latent domain rule violations of r_{cs}, one must specialise the analysis using an encoding for r_{cs}.

The following analogy illustrates this distinction. The analysis framework is like a *bare* multi-purpose food processor, while an encoding of a domain rule is

[3] I work out the details of how to *encode* domain rules in the next sections.

like a selection of accessories for the food processor, such as a specific bowl paired with a specific knife.

3.3.2 Formalisation of the type-centered analysis

In this section, I formally define the framework component of the type-centered analysis. Here, I construct typing problems as a kind of constraint satisfaction problem, using terminology such as *constraint set*, *satisfiability*, and *solution*.

When reading the text for the first time, it might be best to skip to the example in Section 3.3.3, and refer back to these definitions, instead of starting by studying them in detail.

Preliminaries

These definitions refer to some sets that I do not define explicitly. These are the set of *types*, the set of *variables*, and the set of *Java methods*, which I denote METHOD.

For our purposes, the set of *types* may be taken as some standard set of strings, such as, say, all strings over the ASCII alphabet. The set of *variables* may be taken as the set of valid Java identifiers. The set METHOD may be taken as some representation of Java methods.

Constraints and constraint sets

I define the two kinds of typing constraints: *equality constraints* and *ordering constraints*.

Definition 3.1 (Equality constraint)
An *equality constraint* is denoted

$$\mathrm{x} = t$$

where x is a variable and t is a type belonging to some *type set* (cf. Definition 3.4).

Definition 3.2 (Ordering constraint)
An *ordering constraint* is denoted

$$\mathrm{x} \leq \mathrm{y}$$

where x and y are variables.

The next definition, that of *constraint set*, is somewhat lax. Here, I allow myself to group two kinds of objects into the same set. However, I make sure that there is no room for ambiguity when I use this definition.

Definition 3.3 (Constraint set)
A *constraint set* is a set that contains equality constraints and ordering constraints, and nothing else. I use Γ to denote a constraint set.

I further require that all constraints in any one constraint set refer to one specific type set $(\mathbf{T}, \leq)$. In other words, one cannot put constraints that refer to different type sets into the same constraint set.

I sometimes write a constraint set on a shorthand form where I 'collapse' ordering constraints and let equation constraints take the place of variables in ordering constraints. For instance, I may write the constraint set

$$\mathtt{x} = \mathsf{high},\ \mathtt{x} \leq \mathtt{y},\ \mathtt{y} \leq \mathtt{z},\ \mathtt{z} = \mathsf{low}$$

on the shorthand form

$$\mathtt{x} = \mathsf{high} \leq \mathtt{y} \leq \mathtt{z} = \mathsf{low}$$

Encodings

Definition 3.4 (Type set)
A *type set* is a pair $(\mathbf{T}, \leq)$, where

1. $\mathbf{T}$ is a set of *types*.
2. $\leq$ is a total ordering on $\mathbf{T}$.

Definition 3.5 (Latent meaning database)
A *latent meaning database* is a function K that maps a Java method to a constraint set.

The purpose of a latent meaning database is to generate typing constraints from the latent meaning of the signs in a method. I give a full definition of latent meaning database in Section 3.4.

Definition 3.6 (Domain rule encoding)
A domain rule encoding is a pair $(T, \mathcal{K})$, where T is a type set and $\mathcal{K}$ is a latent meaning database.

The purpose of a domain rule encoding is to specialise the type-centered analysis for a specific domain rule.

Solution and satisfiability

I proceed with defining what it means to solve a typing problem.

Definition 3.7 (Solution)
Let Γ be a constraint set, where all constraints refer to the type set $(\mathbf{T}, \leq)$. A function s is a *solution* for Γ if the following conditions are true:

1. For all equation constraints $\mathtt{x} = t$ in Γ, it is the case that $s(\mathtt{x}) = t$.
2. For all ordering constraints $\mathtt{x} \leq \mathtt{y}$ in Γ, it is the case that $s(\mathtt{x}) \leq s(\mathtt{y})$.

In other words, a solution must obey all constraints in their intuitive sense.

I may also refer to a solution as a *typing*.

Definition 3.8 (Satisfiability)
Let Γ be a constraint set. I say that Γ is *satisfiable* if there exists a function that is a solution for Γ.

Framework summary

Finally, I define the *analysis function*, which makes precise what it means to specialise the analysis framework and how the resulting analysis operates.

Definition 3.9 (Analysis function)
Let $e = ((\mathbf{T}, \leq), \mathcal{K})$ be a domain rule encoding. The *analysis function* $A_e : \textsc{method} \to \textsc{boolean}$ is defined as the following algorithm:

Input:
- A Java method m.

Output:
- Boolean.

Algorithm:
1. Using $\mathcal{K}$, generate a constraint set Γ from m.
2. If Γ is satisfiable, output `false`. Otherwise, output `true`.

The algorithm defined here is rather abstract. To make it concrete, I must work out the details of its sub-steps. I address step 1 in Section 3.4, when I describe the details of a latent meaning database. However, I delay developing the details of step 2 until Chapter 4, where I present an algorithm that can find a solution for any satisfiable constraint set.

3.3.3 Example of asserting a domain rule

In this section, I give another example of how to check whether a method violates a domain rule. This time, I use the definitions in Section 3.3.2. My primary aim here is to illustrate these definitions. I also want to illustrate, intuitively, what a latent meaning database does.

The domain rule

Assume we work in a security-critical context, where there is a distinction between confidential and non-confidential data: *Confidential data* is data that an adversary must not learn; *non-confidential data* is data that an adversary is allowed to learn.

Further, assume there exist some output streams that may be observed by an adversary. For example, this may be a stream writing to a publicly readable log file, or a stream writing to an unencrypted TCP connection that is sent over the Internet. In this context, a relevant domain rule is

r_{conf}: An adversary must not be able to learn any confidential data through observing a public output stream.

Encoding: Type set
To model this domain rule, I use two types. I let low denote the type of non-confidential data and high denote the type of confidential data. I group and order these into the type set:

$$t_{\mathsf{conf}} = \{\{\mathsf{low}, \mathsf{high}\}, \mathsf{low} \leq \mathsf{high}\}$$

Encoding: Latent meaning database
To complete the encoding, I need a latent meaning database $\mathcal{K}_{\mathsf{conf}}$ to pair with t_{conf}. But, I have not yet formalised *latent meaning database*. For this example I therefore treat $\mathcal{K}_{\mathsf{conf}}$ as a black-box. I state which typing constraints it outputs, and the rationale for each typing constraint, but do not provide the full details. [4]

The method
In light of the assumptions stated above, consider the following snippet:

```
int x = getSecret();
y = x;
network.send(y);
```

Assume also here that `getSecret` returns some confidential datum and that `network.send` writes its argument to a stream that does not preserve confidentiality. That is, assume this is a latent domain rule violation of r_{conf}.

The typing constraints
I now describe which typing constraints $\mathcal{K}_{\mathsf{conf}}$ outputs for each statement. Precisely, $\mathcal{K}_{\mathsf{conf}}$ specifies how the types of the variables in a statement must be constrained to ensure that the statement is not involved in a latent domain rule violation. Alternatively, $\mathcal{K}_{\mathsf{conf}}$ says: if the constraints output for this statement do not contradict any constraints output for another statement[5], then this statement is not involved in a latent domain rule violation.

For the statement

```
int x = getSecret();
```

the latent meaning database $\mathcal{K}_{\mathsf{conf}}$ outputs the equality constraint $\mathtt{x} = \mathsf{high}$. The rationale for this is that `getSecret` returns confidential data.

Further, for the statement

[4] I formalise r_{conf} and $\mathcal{K}_{\mathsf{conf}}$ in Section 3.5.2. The example I give here agrees with this formalisation.
[5] To be precise, the constraints could also be self-contradictory.

```
network.send(y);
```

the latent meaning database $\mathcal{K}_{\mathsf{conf}}$ outputs the equality constraint $\texttt{y} = \mathsf{low}$. The rationale for this is the assumption that `network.send` writes its argument to a stream that does not preserve confidentiality,

However, for the second statement:

```
y = x;
```

the latent meaning database $\mathcal{K}_{\mathsf{conf}}$ does not output an equality constraint, but rather outputs the ordering constraint

$\texttt{x} \leq \texttt{y}$

Here, the rationale is that for this statement to not be involved in a latent domain rule violation of r_{conf}, $\texttt{x}$ must not have type high while $\texttt{y}$ has type low. In that case, we would have

```
[non-confidential] = [confidential];
```

However, all the other possible typings (the typing $\texttt{x} \mapsto \mathsf{high}, \texttt{y} \mapsto \mathsf{high}$, the typing $\texttt{x} \mapsto \mathsf{low}, \texttt{y} \mapsto \mathsf{high}$, and the typing $\texttt{x} \mapsto \mathsf{low}, \texttt{y} \mapsto \mathsf{low}$) are acceptable.

Solution

Finally, let us investigate whether the typing problem has a solution. There are three typing constraints:

1. $\texttt{x} = \mathsf{high}$
2. $\texttt{x} \leq \texttt{y}$
3. $\texttt{y} = \mathsf{low}$

Assume there existed a solution s. Any such solution would need $s(\texttt{x}) = \mathsf{high}$ and $s(\texttt{y}) = \mathsf{low}$ to satisfy constraints one and three. But, then the second constraint, $\texttt{x} \leq \texttt{y}$, would not be satisfied, because $\texttt{x}$ would not be ordered less-than-or-equal to $\texttt{y}$ with respect to the order defined in t_{conf}. In conclusion, this typing problem is unsatisfiable. We may conclude, as we would expect, that the snippet contains a latent domain rule violation of r_{conf}.

3.4 Latent meaning database

The preceding sections use the abstract definition of *latent meaning database* given in Section 3.3.2. To make this definition concrete, I must answer two questions: First, which sign types does a latent meaning database support? Second, how does a latent meaning database map from the latent meaning of a sign to a constraint set? Answering these questions is my goal with this section, which I structure as follows: In Section 3.4.1, I explain which sign types a latent meaning database supports. In Section 3.4.2, I explain Simplified Jimple, an intermediate representation

that I use when outputting typing constraints with a latent meaning database. In sections 3.4.3 to 3.4.5, I discuss the details of each supported sign type.

3.4.1 Deciding which sign types to support

Deciding which sign types a latent meaning database should support is a nontrivial puzzle. Some sign types to consider for inclusion are control flow constructs, such as `if`, `for`, `while`, and `switch`; method calls; operators and assignments; and the names of types, methods, variables, parameters, and fields. My decision is to support the following sign types:

1. Calls to library methods.
2. Names of local variables and parameters.
3. A few other signs, such as assignments, return statements, and operator applications.

The first two sign types are included because they are especially strong bearers of latent meaning. Calls to library methods often tell much about the latent meaning of standard Java types, such as `String` objects and `byte[]` arrays. For instance, when a `byte[]` array is used as the input argument to an encryption method, this suggests that the array contains confidential data. Names are one of the fundamental ways humans communicate in code, yet they are not understood by the compiler. Hence, names carry significant latent meaning, and many latent domain rule violations concern names.

My decision to restrict the analysis to these sign types is also influenced by my research hypothesis; I want to keep the analysis simple, and see how far one can go with a restricted set of sign types.

In sections 3.4.3 to 3.4.5, I describe each supported sign type in detail. Before that, however, I take a detour: In the preceding text, I use the full Java language in all examples. However, defining a latent meaning database that operates *directly* on the full Java language would be extremely laborious. Therefore, I define a latent meaning database as operating on an intermediate representation that is similar to Java code in many aspects, but much simpler.

3.4.2 Simplified Jimple

In this section, I describe the important details[6] of an intermediate representation that I call *Simplified Jimple*. A latent meaning database operates on Simplified Jimple when outputting typing constraints for a Java method.

Note that I also use Simplified Java for the flow-centered analysis, which I describe in Chapter 5. The two analyses look at different things in a Java program, and therefore there are some constructs in Simplified Jimple that are relevant

[6] I do not describe all details of Simplified Jimple in full depth. Rather, I describe it in sufficient detail to avoid ambiguity in the rest of the text.

only to one of the analyses. Note also that the grammar described here is slightly different from the one implemented in the proof-of-concept tool. Most importantly, the grammar used in the proof-of-concept tool is more compact. With that said, it can be 'rolled out' into the grammar I give in Table 3.1.

Abstract syntax

The abstract syntax of Simplified Jimple is given in Table 3.1. Here, `x`, `y`, `z`, `x0`, and `x1` denote variables. Further, `t` denotes a (primitive or reference) type, and `i` denotes the number of a statement. Further, `O` denotes a Java operator. Note here that Simplified Jimple does not discern between operators, and that one therefore can not distinguish, say, `z=x+y` from `z=x*y` after translation into Simplified Jimple.

Incidentally, observe that the sign types I listed in the start of Section 3.4 'cut across' the statement types in Simplified Jimple. For instance, names show up in almost all statement types.

Translation overview

To translate a Java method into Simplified Jimple, I do as follows: First, I convert the JVM byte code of the method into the Jimple intermediate language [64], using the Soot framework [63]. Then, I use the Soot framework to convert the Jimple representation of the method into SSA form (cf. Section 2.1). Finally, I simplify the SSA form representation into the abstract syntax given in Table 3.1.

Translation details

The translation process has a few details that influence the analysis results, and should be described in greater depth.

First, Soot's Jimple translation adds synthetic variables to represent intermediate values in a Java method. So, a Java statement that includes a complex expression, such as

```
int y = (int) round(sqrt(f.asDouble()));
```

results in a sequence of Simplified Jimple statements, with several synthesised variables.

Second, I add synthesised variables to represent constants. And, I add synthesised variables to represent ignored return variables. Thus, a sequence of statements such as

```
int y = 42;
f(y);
```

where `f` is a method that returns a value, translates to

```
y = $const0;
$ignored0 = f(y);
```

`y = x`	assignment
`y = O x`	unary operator application
`z = x O y`	binary operator application
`y = (t) x`	cast
`y = f(x0, x1, ...)`	method call
`f(x0, x1, ...)`	method call (`void` return)
`jump i`	unconditional jump
`condjump (x0, x1, ...) i`	conditional jump
`return x`	return (non-`void`)
`return`	return `void`

Table 3.1: Abstract syntax of Simplified Jimple.

Third, in the simplification step several Jimple statements disappear. Among these are the Jimple statement equivalent of the **new** operator.

Fourth, in the simplification step all method calls are rewritten on a unified form where also the receiver of a non-static method call is explicitly included in the argument list. For example, the statement

```
myList.add(xs);
```

translates to

```
add(myList, xs);
```

SSA transformation

I should also comment on why I include SSA transformation in the translation process. There are two reasons, both make the type-centered analysis more precise.

The first reason concerns Soot's JVM-to-Jimple translator [64], which I use in the first step of translation. This translator often reuses synthesised variables to represent different intermediate values. Due to this, cases may appear where we legitimately want to type one intermediate value as high and one as low, but where this is impossible because they are both represented by the same variable. With SSA transformation, any variable can hold at most one intermediate value, and such cases can not arise.

The second reason is that SSA transformation allows the type-centered analysis to understand certain cases where a variable holds a value of one type (such as cs) at one time, and then is assigned a value of another type (such as $\neg$cs) at a later time. SSA transformation can prevent false negatives in these cases. I discuss the importance of the SSA transformation in further depth in Section 9.1.

I now return to the main thread and describe in detail the sign types that a latent meaning database supports.

3.4.3 Names

A latent meaning database can generate typing constraints from two kinds of names: local variable names, and method names. Specifically, a latent meaning database includes two mappings: a mapping N_v, describing the latent meaning of local variable names, and a mapping N_m, describing the latent meaning of method names. Formally, each such mapping is a partial function from identifiers to a type set.

Constraint generation

Assume that m is a method, $t = (\mathbf{T}, \leq)$ is a type set, and that $\mathcal{K}$ is a latent meaning database with name sign mappings N_v and N_m. Constraint generation for the name signs in m proceeds as follows:

1. For each local variable $\texttt{x}$ in m, and each entry $n \mapsto w$ in N_v, if n is a substring of $\texttt{x}$, $\mathcal{K}$ outputs the equality constraint $\texttt{x} = w$.
2. For each entry $n \mapsto w$ in N_m, if n is a substring of the name of m, $\mathcal{K}$ outputs an equality constraint $\texttt{x} = w$, for each variable $\texttt{x}$ that is a return variable in m.

Example

Let K be a latent meaning database for a type set $(\{\mathsf{a}, \mathsf{b}\}, \mathsf{a} \leq \mathsf{b}\})$, with name sign mappings:

N_v: $\{\texttt{x} \mapsto \mathsf{a}\}$

N_m: $\{\texttt{foo} \mapsto \mathsf{b}\}$

Assume that a method named `foobar` with the following Simplified Jimple representation:

```
x = y;
return y;
```

is input to K. The latent meaning database K would then output the typing constraints $\texttt{x} = \mathsf{a}$ and $\texttt{y} = \mathsf{b}$.

3.4.4 Library method calls

Constraint generation for library calls works slightly different depending on whether the callee returns **`void`** or not.

For a method call to a `void`-returning library method, such as

```
f(x1, x2, x3);
```

a latent meaning database may output any typing constraints over the argument variables (in this case `x1`, `x2`, and `x3`).

For a method call to a value-returning library method, such as

```
y = f(x1, x2, x3);
```

a latent meaning database may output any typing constraints over the argument variables (in this case `x1`, `x2`, and `x3`) and the return variable (in this case `y`).

3.4.5 Other signs

In addition to names and library method calls, a latent meaning database may specify how to generate constraints for:

- assignment statements,
- operator application statements,
- synthesised variables that represent constant values (cf. Section 3.4.2),
- the `this` variable, and
- calls to methods not recognised as library methods.

Assignment and operator application statements
For assignment statements and operator application statements, a latent meaning database may specify any typing constraints for the variables involved in the statement.

Variables holding constants
For each synthesised variable representing a constant value, a latent meaning database may specify an equality constraint. In the current version of the proof-of-concept tool, the latent meaning database must map all constants to the same type.

The `this` variable
A latent meaning database may specify an equality constraint for the `this` variable.

Calls to methods not recognised as library methods
Finally, a latent meaning database may output typing constraints for the variables appearing as arguments to a method not recognised as a library method. Here, $\mathcal{K}$ must specify *one* general rule that only depends on the number of arguments to the method not recognised as a library method.

3.5 Domain rules encoded for the type-centered analysis

In this section I describe the details of how I encode the domain rules r_{cs} and r_{conf} for the type-centered analysis. That is, I describe their type sets and their latent meaning databases in full detail. Although I use, and partially define, these domain rules in preceding section in the chapter, the definitions I give in this section are authoritative.

Before I get to the details, let me state two preliminaries: First, for simplicity, I specify both these domain rules using a two-element type set. Hence, each type set can be written on the general form

$$t = (\{\alpha, \beta\}, \alpha \leq \beta)$$

for some α and β. Second, the latent meaning databases I use here are designed using the same principle I described in Section 3.3.3: For a given statement, a latent meaning database outputs constraints that are such that if they do not contradict any constraints output for another statement (or they are self-contradictory) then this statement is not involved in a latent domain rule violation.

3.5.1 Domain rule I: r_{cs}

Any *secure challenge* must stem from a cryptographically secure random source.

Elaboration

This domain rule concerns values that are *secure challenges*. For my purposes, I define a secure challenge as a value that is generated with purpose to be used for authentication and is generated by a cryptographically secure random number generator.[7]

Type set

I use the type set

$$t_{\mathsf{cs}} = \{\{\mathsf{cs}, \neg\mathsf{cs}\}, \mathsf{cs} \leq \neg\mathsf{cs}\}$$

where cs stands for *cryptographically secure*, and ¬cs stands for *not cryptographically secure*, as in the example in Section 3.1.

In the next paragraphs, I describe $\mathcal{K}_{\mathsf{cs}}$, the latent meaning database for r_{cs}.

Latent meaning of names

I define two minimal mappings. In particular I include `challenge` $\mapsto$ cs in the method name mapping.

[7] I do not define cryptographically secure random generator precisely.

Latent meaning of library method calls
I add to $\mathcal{K}_{\mathsf{cs}}$ information about the secure random number generator `java.security.SecureRandom`. I also add information about certain standard library methods that can preserve cryptographically secure random challenges. An example of such a method is `Long.toString`: if a `Long` is a cryptographically secure challenge, then so is its `String` representation. I comment on $\mathcal{K}_{\mathsf{cs}}$ in more detail in Section 7.2 of the validation chapter, in context of relevant examples.

Latent meaning of assignments, operators, and casts
For an assignment statement, such as

```
y = x;
```

I let $\mathcal{K}_{\mathsf{cs}}$ output the ordering constraint $\mathtt{x} \leq \mathtt{y}$. That is, I disallow any typing where `y` has type cs, but `x` has type ¬cs.

For an operator application statement, such as

```
z = x O y;
```

I err on the safe side and let $\mathcal{K}_{\mathsf{cs}}$ output an equality constraint $\mathtt{z} = \neg\mathsf{cs}$. I use a similar rule for casts.

Latent meaning of constants
As constants are hardcoded into the program text, I assume that any variable holding a constant value is not cryptographically secure. Therefore, I let $\mathcal{K}_{\mathsf{cs}}$ output an equality constraint $\mathtt{\$constx} = \neg\mathsf{cs}$ for any synthesised variable `$constx` holding a constant value.

Latent meaning of `this`
For the **`this`** variable, I let $\mathcal{K}_{\mathsf{cs}}$ output the equality constraint **`this`** $= \neg\mathsf{cs}$. Also here, I err on the safe side, as there may be cases where **`this`** is a cryptographically secure challenge.

Latent meaning of non-library method calls
Erring on the safe safe, I let $\mathcal{K}_{\mathsf{cs}}$ output an equality constraint $\mathtt{x} = \neg\mathsf{cs}$ for any variable mentioned in an unknown method call.

3.5.2 Domain rule II: r_{conf}

An adversary must not be able to learn any confidential data through observing a public output stream.

Elaboration

The definition of r_{conf} rests on the same assumptions I described in Section 3.3.3. There is a distinction between *confidential* and *non-confidential data*: Confidential data is data that an adversary must not be able to learn, while non-confidential data is data that an adversary is allowed to learn.

Type set

I use the type set

$$t_{\mathsf{conf}} = \{\{\mathsf{low}, \mathsf{high}\}, \mathsf{low} \leq \mathsf{high}\}$$

where low denotes the type of non-confidential data and high denotes the type of confidential data.

In the next paragraphs, I describe $\mathcal{K}_{\mathsf{conf}}$, the latent meaning database for r_{conf}.

Latent meaning of names
I define two minimal mappings for name signs. In particular, I map variable names such as `plaintext` to high, but variable names such as `ciphertext` and `encrypted` to low.

Latent meaning of library method calls
I add to $\mathcal{K}_{\mathsf{conf}}$ information about the library call `System.getEnv`. Further, I add information about the `PrintStream` output stream, assuming that it does not preserve confidentiality. Finally, I add information about certain standard library methods such as `StringBuilder.append`, which are like assignments in that if the 'input' is high, then so must the 'output' be high. I comment on $\mathcal{K}_{\mathsf{conf}}$ in more detail in Section 7.2 of the validation chapter, in context of relevant examples.

Latent meaning of assignments, operators, and casts
For an assignment statement sign, such as

```
y = x;
```

I let $\mathcal{K}_{\mathsf{conf}}$ output the ordering constraint $\texttt{x} \leq \texttt{y}$. That is, I disallow any typing where `x` is high but `y` is low, in which case confidential data would leak from `x` to `y`.

For an unary operator application statement, such as

```
y = (f) x;
```

I also let $\mathcal{K}_{\mathsf{conf}}$ output the ordering constraint $\texttt{x} \leq \texttt{y}$. Also here, the crux is to disallow any typing where `x` is high but `y` is low. I use a similar rule for binary operator applications and casts.

Latent meaning of constants
As constants are hardcoded into the program text, I assume that any variable holding a constant value is not confidential. Therefore, I let $\mathcal{K}_{\mathsf{conf}}$ output an equality constraint `$constx` $= \mathsf{low}$ for any variable `$constx` holding a constant value.

Latent meaning of `this`
For the **`this`** variable, I let $\mathcal{K}_{\mathsf{cs}}$ output the equality constraint **`this`** $= \mathsf{low}$. This is imprecise, as there may be cases where the **`this`** variable is confidential.

Latent meaning of non-library method calls
Erring on the safe safe, I let $\mathcal{K}_{\mathsf{conf}}$ output an equality constraint `x` $= \mathsf{low}$ for any variable mentioned in an unknown method call. This is because I cannot know how an unknown method will treat any confidential arguments.

Chapter summary and continuation

In this chapter, I introduced the type-centered analysis, the first of the two analyses I present in the thesis. First, I developed the idea that checking whether a method violates a domain rule can be cast as a *typing problem*. Following that, I defined the notion of *latent meaning*, and described a formal technique for constructing and solving typing problems using a *latent meaning database*. I also described Simplified Jimple, an intermediate representation that I use for both analyses. Finally, I described how to specialise the type-centered analysis to enforce two domain rules, r_{cs} and r_{conf}.

In the next chapter, I expand on one detail, how to find solutions that satisfy a constraint set (cf. Section 3.3.2). In Section 7.2 of Chapter 7, the validation chapter, I do experimental validation of the type-centered analysis. First, in Section 7.2.1, I use the type-centered analysis specialised for r_{cs} to identify a latent domain rule violation in the source code of an electronic voting system. Second, in Section 7.2.2, I use the type-centered analysis specialised for r_{conf} to identify latent domain rule violations in test cases from a third-party test suite. Further, in Chapter 8, I compare the type-centered analysis to closely related work. Finally, in Chapter 9, I discuss the type-centered analysis, compare it to the flow-centered analysis, and present ideas for future work.

CHAPTER 4

Solving typing problems

In this chapter, I pick up the thread from Section 3.3 and present and prove correct an algorithm that can find a solution for any satisfiable constraint set. My goal with this chapter is to demonstrate that I have developed a robust technique for solving typing problems. Readers interested only in the essence of the thesis may skip the chapter, with no risk of losing sight of the larger picture.

The structure of the chapter is as follows: First, in Section 4.1, I state a number of definitions, which match those given in Section 3.3, but are more rigorous. I also state and prove some lemmas. Then, in Section 4.2, I define the *typing algorithm*. Further, in Section 4.3, I prove that this algorithm finds a solution to any satisfiable typing problem. Finally, in Section 4.4, I make clear the connection between the definitions I work out in this chapter, and those I gave in Section 3.3. To manage complexity, I heed the advice of Lamport [37, 38] and use a *structured proof* system for some proofs. Following convention, I use the plural *we* to mean the author and the reader.

Note that, to further build confidence in the correctness of the typing algorithm, the proof-of-concept tool i) cross-checks the output of every run of the typing algorithm with the output of a fully independent solution algorithm[1], and ii) explicitly asserts the correctness of each solution.

[1] I have derived an alternative solution algorithm, which essentially is an augmentation of Tarjan's strongly connected components algorithm [59]. I do not describe or analyse this algorithm in the thesis, but interested readers may look at its implementation in the file `analysis/src/main/java/analysis/AlternativeConstraintSolver.java`, in the source code of the proof-of-concept tool.

4.1 Preliminaries

Definition 4.1 (Problem instance)
A *problem instance* is a four-tuple $(\mathbf{V}, \mathbf{C}, \mathbf{E}, \mathbf{R})$, where

- $\mathbf{V}$ is the set of integers from 1 to n, for some n.
- $\mathbf{C}$ is the set of integers from 1 to m, for some m.
- $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{C}$ is a binary relation.
- $\mathbf{R} \subseteq \mathbf{V} \times \mathbf{V}$ is a binary relation.

I refer to

- $\mathbf{V}$ as the variables,
- $\mathbf{C}$ as the value domain,
- $\mathbf{E}$ as the equality constraints, and
- $\mathbf{R}$ as the ordering constraints.

I let $\mathbf{R}'$ denote the transitive closure of $\mathbf{R}$, and call a member of $\mathbf{R}'$ a transitive ordering constraint.

An example of a problem instance is

$$(\{1,2,3\}, \{1,2,3\}, \{(1,1),(2,3)\}, \emptyset)$$

where we have the equality constraints $1\mathbf{E}1$ and $2\mathbf{E}3$, and no ordering constraints.

Definition 4.2 (Solution)
Let $p = (\mathbf{V}, \mathbf{C}, \mathbf{E}, \mathbf{R})$ be a problem instance. A function f is a *solution* for p if the following three conditions are true:

P_1: f is defined for all $v \in \mathbf{V}$.

P_2: $\forall (v, c) \in \mathbf{E}.\ f(v) = c$.

P_3: $\forall (v, v') \in \mathbf{R}.\ f(v) \leq f(v')$.

Intuitively:

1. The function f must have an assignment for every variable.
2. For all equality constraints $v\mathbf{E}c$, f must map v to c.
3. For all ordering constraints $v\mathbf{R}v'$, the value f assigns to v must be less than or equal to the value it assigns to v'.

This definition matches Definition 3.7 in Chapter 3. (The definition in Chapter 3 is less rigorous, though, as it does not include the first requirement.)

Definition 4.3 (Satisfiability)
A problem instance p is *satisfiable* if there exists a function that is a solution for p. Otherwise, if no such function exists, the problem instance is *unsatisfiable*.

This definition matches Definition 3.8 in Chapter 3.

Lemma 4.1 (Satisfiable problems obey transitive ordering constraints)
Let $p = (\mathbf{V}, \mathbf{C}, \mathbf{E}, \mathbf{R})$ be a problem instance. If f is a solution to p and (v, v') is any element in $\mathbf{R}'$ then $f(v) \leq f(v')$.

Proof. Assume that f is a solution to p, and let (v, v') be any element in $\mathbf{R}'$. Because $v\mathbf{R}'v'$ and from the definition of transitive closure, there exists a sequence of variables $v_1, v_2, \ldots, v_{n-1}, v_n$ where $v_1 = v$ and $v_n = v'$, and each adjacent pair (v_i, v_{i+1}) is an element of $\mathbf{R}$. Because f is a solution to p, $f(v_i) \leq f(v_{i+1})$ for each such pair. Therefore, $f(v) = f(v_1) \leq f(v_2) \ldots f(v_{n-1}) \leq f(v_n) = f(v')$, and, in particular, $f(v) \leq f(v')$. □

Lemma 4.2 (A problem with conflicting equality constraints is unsatisfiable)
Any problem instance $p = (\mathbf{V}, \mathbf{C}, \mathbf{E}, \mathbf{R})$ is unsatisfiable if there exists elements (v, c) and (v, c') in $\mathbf{E}$, when $c \neq c'$.

Proof. Immediate. No function can yield different output values for the same input value, which would be required to satisfy any two such constraints. □

Corollary 4.1 A satisfiable problem has no conflicting equality constraints.

Proof. Contrapositive of Lemma 4.2. □

4.2 Typing algorithm

Definition 4.4 (Typing algorithm)
Given a problem instance $p = (\mathbf{V}, \mathbf{C}, \mathbf{E}, \mathbf{R})$, I define the *typing algorithm* t_p and the utility functions q_p, m_p, and a_p as follows:

$$t_p(v) = \begin{cases} \bot & \text{if } q_p(v) \neq \bot \wedge q_p(v) > m_p(v) \\ m_p(v) & \text{if } q_p(v) = \bot \\ q_p(v) & \text{otherwise} \end{cases}$$

$$q_p(v) = \begin{cases} c & \text{if } \exists! c \in \mathbf{C}.\ v\mathbf{E}c \\ \bot & \text{otherwise} \end{cases}$$

$$m_p(v) = \begin{cases} \min a_p(v) & \text{if } a_p(v) \neq \emptyset \\ \max \mathbf{C} & \text{otherwise} \end{cases}$$

$$a_p(v) = \{q_p(v') \mid \forall v' \in \mathbf{V}.\ v\mathbf{R}'v' \wedge q_p(v') \neq \bot\}$$

The intuition behind the utility functions is as follows:

- $q_p(v)$ yields the value of the equality constraint for v, if such a constraint exists and no conflicting equality constraint exists, or otherwise $\bot$.
- $a_p(v)$ yields the set of values $q_p(v')$, for all variables v' that must be no less than v and has $q_p(v') \neq \bot$.
- $m_p(v)$ yields the value of the minimum element of $a_p(v)$, if $a_p(v)$ is nonempty, or otherwise $\max \mathbf{C}$.

4.3 Correctness proof

In the rest of the chapter, my focus is proving that the typing function t_p yields solutions for any satisfiable problem. I first introduce two lemmas.

Lemma 4.3
Let $p = (\mathbf{V}, \mathbf{C}, \mathbf{E}, \mathbf{R})$, be a problem instance, and let v be any element of $\mathbf{V}$. If p is satisfiable and $q_p(v) \neq \bot$ then $q_p(v) \leq m_p(v)$.

Proof. Let
  A: p is satisfiable
  B: $q_p(v) \neq \bot$
  C: $q_p(v) \leq m_p(v)$
where v is some element of $\mathbf{V}$. We need to demonstrate $A \wedge B \implies C$. This is equivalent to showing that $A \wedge B \wedge \neg C$ is a contradiction.

$A \wedge B \wedge \neg C \implies$ FALSE

1. Assume $A \wedge B \wedge \neg C$. Observe that
   $\neg C$: $q_p(v) > m_p(v)$
2. Let f be any solution to p. (By A, we may assume that a solution exists.)
3. FALSE
   a) $\exists! c \in \mathbf{C}.\ v\mathbf{E}c,\ q_p(v) = c$
      PROOF: Using B and the definition of q_p.
   b) $\exists (v', c') \in \mathbf{E}.\ v\mathbf{R}'v' \wedge c > c'$
      i. $m_p(v) < \max \mathbf{C}$
         PROOF: Because, by $\neg C$, $q_p(v)$ is strictly larger than $m_p(v)$, and $q_p(v)$ can not be larger than $\max \mathbf{C}$.
      ii. $a_p(v) \neq \emptyset$
         PROOF: From $m_p(v) < \max \mathbf{C}$ and the definition of m_p.
      iii. $m_p(v) = \min a_p(v) = q_p(v')$, for some v'
         PROOF: From the definition of m_p and a_p, using $m_p(v) < \max \mathbf{C}$ and $a_p(v) \neq \emptyset$, there must be some v' which makes for the smallest value of q_p in $a_p(v)$. (Note that there may well be several such v's.)
      iv. $v\mathbf{R}'v'$
         PROOF: From the definition of a_p, using the previous step.
      v. $\exists (v', c') \in \mathbf{E}.\ q_p(v') = c'$
         PROOF: From the definition of a_p and q_p, using step iii. (If this was not the case, $q_p(v')$ would not be in $a_p(v)$.)
      vi. $m_p(v) = \min a_p(v) = q_p(v') = c'$
         PROOF: From the definition of m_p, using steps iii and v.
      vii. $c = q_p(v) > m_p(v) = \min a_p(v) = q_p(v') = c'$

PROOF: Using $\neg C$, step a, and step vi.

viii. $c > c'$

PROOF: Using step vii.

c) $f(v) = c, f(v') = c'$

PROOF: Since f is a solution to p, it must obey the equality constraints $(v, c) \in \mathbf{E}$ and $(v', c') \in \mathbf{E}$.

d) $f(v) \leq f(v')$

PROOF: Since f is a solution to p, it must obey the transitive ordering constraints $v\mathbf{R}'v'$. (By Lemma 4.1.)

e) FALSE

PROOF: Given the requirement $c > c'$, no function f can satisfy $f(v) = c \leq f(v') = c'$. By Definition 4.3 this implies that p is unsatisfiable, that is $\neg A$. Hence, we reach the contradiction $A \wedge \neg A$.

□

Lemma 4.4
Let $p = (\mathbf{V}, \mathbf{C}, \mathbf{E}, \mathbf{R})$ be a problem instance. If (v, v') is any element of $\mathbf{R}$ then $m_p(v) \leq m_p(v')$.

Proof. Let (v, v') be any element in $\mathbf{R}$. First, note that if $a_p(v') = \emptyset$ then $m_p(v') = \max \mathbf{C}$, in which case the implication holds trivially, because m_p can never output anything larger than $\max \mathbf{C}$. Now, assume instead $a_p(v') \neq \emptyset$. Note that from the assumption $v\mathbf{R}v'$ and the definition of transitivity, it follows that for any v'' such that $v'\mathbf{R}'v''$, we also have $v\mathbf{R}'v''$. From the definition of a_p it therefore follows that $a_p(v) \supseteq a_p(v')$. Finally, recall that $m_p(x)$, always picks the minimum element of $a_p(x)$ if $a_p(x)$ is non-empty. Therefore, from $a_p(v) \supseteq a_p(v') \neq \emptyset$, $m_p(v) \leq m_p(v')$ follows. □

I now prove that if p is any satisfiable problem instance, then t_p is a solution for p. I do this by proving that t_p satisfies each of the three requirements to a solution, in turn.[2]

Lemma 4.5
If p is satisfiable, then t_p is defined for all $v \in \mathbf{V}$.

Proof. Let

A: p is satisfiable

B: t_p is defined for all $v \in \mathbf{V}$

$A \implies B$:

1. Assume A.

[2]Note though, that there are cases where p is not satisfiable, but where t_p still suggests 'solutions'. This is not important however, as it is straightforward to check whether a proposed solution is actually a solution, using Definition 4.2.

2. Let v be any member of $\mathbf{V}$.
3. B
   a) $q(v) \neq \bot \implies t_p(v) \neq \bot$
      i. $q_p(v) \leq m_p(v)$
         PROOF: By Lemma 4.3, using A and $q_p(v) \neq \bot$.
      ii. $t_p(v) = q_p(v) \neq \bot$
         PROOF: By the definition of t_p, using $q_p(v) \neq \bot$ and $q_p(v) \leq m_p(v)$.
   b) $q(v) = \bot \implies t_p(v) \neq \bot$
      i. $t_p(v) = m_p(v)$
         PROOF: By the definition of t_p, using $q_p(v) = \bot$.
      ii. $t_p(v) \neq \bot$
         PROOF: Using $t_p(v) = m_p(v)$, and m_p being a total function
   c) Either $q(v)$ is defined, or it is not defined. I have shown that in either case $t_p(v) \neq \bot$ follows. Since v is a generic particular, this implies B.

□

Lemma 4.6
If p is satisfiable, then $\forall(v, c) \in \mathbf{E}.\ t_p(v) = c$.

Proof. Let
  A: p is satisfiable
  B: $\forall(v, c) \in \mathbf{E}.\ t_p(v) = c$
$A \implies B$:
1. Assume A.
2. Let (v, c) be any member of $\mathbf{E}$.
3. B
   a) $q_p(v) = c$
      PROOF: From the definition of q_p, using $v\mathbf{E}c$, and Corollary 4.1.
   b) $q_p(v) \leq m_p(v)$
      PROOF: By Lemma 4.3, using A and $q_p(v) \neq \bot$.
   c) $t_p(v) = q_p(v)$
      PROOF: By the definition of t_p, using $q_p(v) \neq c$ and $q_p(v) \leq m_p(v)$.
   d) $t_p(v) = c$

□

Lemma 4.7
If p is satisfiable, then $\forall(v, v') \in \mathbf{R}.\ t_p(v) \leq t_p(v')$.

Proof. Let
  A: p is satisfiable

B: $\forall(v, v') \in \mathbf{R}.\ t_p(v) \leq t_p(v')$
we need to demonstrate $A \implies B$.

$A \implies B$

1. Let (v, v') be any element of $\mathbf{R}$.
2. $t_p(v) \leq m_p(v)$

   PROOF: By Lemma 4.5 and the definition of t_p, using A.
3. $t_p(v') \leq m_p(v')$

   PROOF: Similar.
4. $m_p(v) \leq m_p(v')$

   PROOF: Using Lemma 4.4 and $v\mathbf{R}v'$.
5. $t_p(v) \leq t_p(v')$
   a) $q_p(v') = \perp \implies t_p(v) \leq t_p(v')$
      i. $t_p(v') = m_p(v')$

         PROOF: By the definition of t_p, using $q_p(v) = \perp$.
      ii. $t_p(v) \leq m_p(v) \leq m_p(v') = t_p(v')$

         PROOF: Using $t_p(v) \leq m_p(v)$, $t_p(v') = m_p(v')$, and $m_p(v) \leq m_p(v')$.
      iii. $t_p(v) \leq t_p(v')$
   b) $q_p(v') \neq \perp \implies t_p(v) \leq t_p(v')$
      i. $q_p(v') \leq t_p(v')$

         PROOF: By the definition of t_p, using $q_p(v') \neq \perp$.
      ii. $m_p(v) \leq q_p(v')$

         PROOF: By the definition of m_p and a_p, using $v\mathbf{R}v'$, and $q_p(v') \neq \perp$. (Since v and v' are related by $\mathbf{R}$, $q_p(v')$ is in $a_p(v)$.)
      iii. $t_p(v) \leq m_p(v) \leq q_p(v')$

         PROOF: Using $t_p(v) \leq m_p(v)$ and $m_p(v) \leq q_p(v')$.
      iv. $t_p(v) \leq t_p(v')$

         PROOF: Using $q_p(v') \leq t_p(v')$ and $t_p(v) \leq m_p(v) \leq q_p(v')$.

□

Theorem 4.1 (Typing algorithm gives a solution for all satisfiable problems)
Let $p = (\mathbf{V}, \mathbf{C}, \mathbf{E}, \mathbf{R})$ be a problem instance. If p is satisfiable, then t_p is a solution for p.

Proof. By lemmas 4.5, 4.6, and 4.7 we know that t_p satisfies all requirements to a solution for any satisfiable problem p. □

4.4 Translating from a typing problem to a problem instance

I now clarify how to translate between the definitions in Section 3.3 and those I develop in this chapter.

In this section, let $t = (\mathbf{T}, \leq)$ be a type set (Definition 3.4) and Γ a constraint set where each equality constraint $\mathtt{x} = t$ is such that $t \in \mathbf{T}$.

Variable mapping
To map between a set of variables as defined in Chapter 3 and a set of variables as defined in this chapter, choose any function q that is a one-to-one mapping between all the variables seen in Γ and the set of integers from 1 to m, where m is the number of distinct variables seen in Γ. For instance, if

$$\Gamma = \{\mathtt{x} = \mathsf{cs}, \mathtt{y} = \mathsf{cs}, \mathtt{x} \leq \mathtt{y}\}$$

then one choice of q is such that $q(\mathtt{x}) = 1$ and $q(\mathtt{y}) = 2$.

Type mapping
To map from $\mathbf{T}$ and a value set as defined in this chapter, use the function $w : \mathbf{T} \rightarrow \mathbb{Z}$, defined as follows:

1. If $t \leq t'$ for all $t' \in \mathbf{T}$. $w(t) = 1$
2. If $t > t'$ for all $t' \in \mathbf{T}$. $w(t) = |\mathbf{T}|$
3. If $t \leq t'$ then $w(t) \leq w(t')$

In other words, w is such that if

$$t = \{\{a, b, c\}, a \leq b \leq c\}$$

then $w(a) = 1$, $w(b) = 2$, and $w(c) = 3$.

Mapping summary
The mapping from t and Γ into a problem instance $(\mathbf{V}, \mathbf{C}, \mathbf{E}, \mathbf{R})$ is defined as follows:

$$\begin{aligned}
\mathbf{V} &= \{q(\mathtt{x}) \text{ for all variables } \mathtt{x} \text{ seen in } \Gamma\} \\
\mathbf{C} &= \{w(t) \text{ for all types } t \in \mathbf{T}\} \\
\mathbf{E} &= \{(q(\mathtt{x}), w(t)) \text{ for all equation constraints } \mathtt{x} = t \in \Gamma\} \\
\mathbf{R} &= \{(q(\mathtt{x}), q(\mathtt{y})) \text{ for all ordering constraints } \mathtt{x} \leq \mathtt{y} \in \Gamma\}
\end{aligned}$$

If a typing problem is satisfiable, we may map back from the output of t_p to a solution f (as specified in Definition 4.2 in Chapter 3) as follows:

$$f(\mathtt{x}) = w^{-1}(t_p(q(\mathtt{x})))$$

where w^{-1} denotes the inverse of w.[3]

[3] Note that, by the definition of t_p, and using that the typing problem is satisfiable, any argument given w^{-1} in this mapping will be in its domain.

CHAPTER 5

Flow-centered analysis

In this chapter, I describe the flow-centered analysis, the second of the two analyses I present in the thesis.

I start with an introductory section, to motivate the analysis and to introduce its main ideas.

5.1 Introduction

Both the domain rules r_{cs} (Section 3.5.1) and r_{conf} (Section 3.5.2) essentially concern flow-related flaws. For r_{cs}, we want to assert that certain variables have a value that has flowed from a cryptographically secure random source. For r_{conf}, we want to assert that there is no flow of confidential information into insecure locations, such as a public log file. However, while the type-centered analysis can find flow-related flaws that are violations of these domain rules, there are many flow-related flaws it can not identify.[1] For example, consider the snippet:

```
void sendNext() {
  byte[] msgBytes = msgQueue.poll();
  network.write(msgBytes);
}
```

Assume here that `network.write` writes to an underlying socket that may be in a blocked state, and that `network.write` may signal this by immediately returning `-1`, without sending any data. Using the type-centered analysis, there is no way to assert that the return value of `network.write` is checked, so that should the underlying socket be blocked, the code can detect this and take some

[1] I treat the issue of which flow-related flaws each analysis can and can not find with more rigour in sections 9.1 to 9.3, where I discuss and compare the analyses.

appropriate action.[2] The purpose of the flow-centered analysis is to identify flaws like the one seen in this example.

5.1.1 Critical values

As a first step in developing the flow-centered analysis, I shall say that there are certain data values that are *critical* in the sense that 'using them in a satisfactory way' is crucial to a program's correctness. To illustrate this, consider an interactive banking application where there are some values that represent account balances, and some values that specify which colour scheme to use for a GUI. Clearly, the values in the first group are relatively more important to treat correctly than those in the second. If the colour schemes are mixed up, it is a nuisance; if the balances are corrupted, it is a business-critical flaw.

Which values are critical, and what it means for them to be used in a satisfactory way, depends on context. For the `network.write` example, we may say that its return value `numSent` is used in a satisfactory way if it ends up in an `if` statement where it is compared to `-1`, and if, then, some appropriate action is taken if the underlying network channel is blocked.

Precisely, what it means for a critical value to be used in a satisfactory way depends on its *domain type*, that is, the type that the value is meant to represent.[3] Let me illustrate this. Java programmers often use the implementation type `java.lang.String` to represent the domain type *password*. But, to assert whether a value of such a type is used in a satisfactory way, we need to use our domain knowledge of what it means to be a password.

5.1.2 Satisfactory use of critical values as a domain rule

I consider failing to use a critical value in a satisfactory way a violation of a rule that shows up in different (probably all) domains:

> r_{crit}: A critical value must be used in a way that is satisfactory with respect to its domain type.

The domain rule r_{crit} is a general rule that one must *instantiate* for concrete domain types. For example, r_{crit} can be instantiated for the domain type *cryptographically secure random challenge*.

In Section 5.3 and Section 5.4, I describe how the flow-centered analysis must be *specialised* to do analysis of specific instances of r_{crit}. This is conceptually similar to how the type-centered analysis must be specialised to do analysis of specific domain rules, using a latent meaning database.

[2] I presume here that it is not appropriate to just drop a message.

[3] Let me repeat the example from Chapter 1: To see the distinction between an implementation type and the domain type it is meant to represent, consider the distinction between a class `JetEngine` in a flight simulator and an engine on an actual jet plane.

5.1.3 Disappearing critical values

It is hard, if not impossible, to formalise a general notion of *used in a satisfactory way*. Therefore, I design the flow-centered analysis around another notion, *disappearing critical values*, which (I claim) approximates *used in a satisfactory way* in many interesting cases.

In Section 5.2, I make precise what I mean by *disappear*. For now, an analogy is that a critical value that *does not* disappear is like a message passed along carefully from its initial sender to a 'worthy recipient' through a chain of trusted intermediates. Conversely, a critical value that *does* disappear is like a message that is forgotten or tampered with at some point in the chain between its sender and recipient. In the `network.write` example, we may see `numSent` as disappearing because it does not end up with a worthy recipient, an `if` statement dealing with the possible error condition.

Structure of the remainder of the chapter

I now start formalising the flow-centered analysis. The formalisation spreads across several sections, where I describe different subcomponents of the analysis. First, in Section 5.2, I describe *the flow language* and a semantics for this language, the *disappearing criticals function*. My goal with this section is to specify what it means for a critical value to disappear in a *flow program* (a program in the flow language). Second, in Section 5.3, I describe how to translate Java methods to the flow language, and how to specialise this translation process to analyse instantiations of r_{crit}. Third, in Section 5.4 I define the flow-centered analysis proper, and describe how to use it to enforce instantiations of r_{crit}, and how to apply it to another use case.

5.2 Computing with critical values

This section is divided into two. In Section 5.2.1, I describe the flow language; in Section 5.2.2, I describe the disappearing criticals function. The purpose of the section is to describe a formal way to compute with critical values.

5.2.1 The flow language

The *flow language* is designed for modeling the aspects of a Java method relevant for finding disappearing critical values. The language's abstract syntax is given in Figure 5.1. In this grammar, i and t denote statement position indices. (Below, I only use t to refer to a jump target.) Further, $\mathtt{x}$ denotes a single variable. In general, $\overline{v}$ denotes a list of vs. For example, $\overline{\mathtt{x}}$ denotes a list of variables. In the abstract syntax, I omit separator characters such as commas or semicolons. However, when I use flow language syntax, I include some ad-hoc separators for clarity.

Flow program

A *flow program* p is a sequence of *statements*, of which there are three kinds:

1. (so-called) flow statements (`flow`),
2. unconditional jumps (`jump`), and
3. conditional jumps (`condjump`).

Flow programs may be seen as lists: If p is a flow program, p_i denotes the ith statement (zero-indexed) in p. Hence, the start of p is p_0. I let the symbol `end` denote the end of a flow program. One may think of `end` as similar to the constructor nil in an abstract data type for lists.

I now describe each kind of statement and their intuitive meanings. Note though that the only formal meaning of these statements is given by the disappearing criticals function $\mathcal{D}$, and that my informal description does not precisely describe what $\mathcal{D}$ does.[4]

The most important notions I describe here are to *output* and to *consume* a critical value.

Flow statement

A flow statement models flow of critical values into and out of variables, when a Java statement is executed. It is denoted `flow` $(\overline{w})$, where $\overline{w}$ is a list of *flow parameters*.

A flow parameter d `x` is a pairing of a *flow direction* and a variable. It describes flow behaviour with respect to *one* variable. There are four flow directions: `in`, `out`, `inout`, and `none`:

- A flow parameter `in x` indicates that a critical value is *consumed* from `x`. The intuition is that when the flow statement containing the flow parameter 'executes', any critical value in `x` (there will be one or none) is used in a satisfactory way; *that* critical value has been dealt with correctly, and is no longer a concern. An `in`-parameter can be used to model, say, the input parameter for writing to a stream.
- A flow parameter `out x` indicates that a critical value is *output* to `x`. The intuition is that when the flow statement containing the flow parameter 'executes', a critical value is put into `x`. An `out`-parameter can be used to model, say, an output parameter of an encryption method.
- A flow parameter `inout x` has the meaning that first a critical value is consumed from `x`, and then another is output to `x`. An `inout`-parameter can be used to model, say, an encryption method that takes as input an array of ciphertext and writes more ciphertext to the array.

[4]For one thing, $\mathcal{D}$ only considers one variable at a time.

$f \rightarrow \overline{s}$	flow program
$s \rightarrow$ `flow` $(\overline{w})$	flow statement
$\rightarrow$ `jump` t	unconditional jump statement
$\rightarrow$ `condjump` $(\overline{\mathtt{x}})$ t	conditional jump statement
$w \rightarrow d$ `x`	flow parameter
$d \rightarrow$ `in` | `out` | `inout` | `none`	flow direction

Table 5.1: Abstract syntax of the flow language.

- A flow parameter `none x` is ignored by the analysis. I include `none` only to simplify the implementation of the proof-of-concept tool.[5]

Jump statement
The intuitive meaning of a jump statement `jump` t is: continue execution at p_t. Any jump statement in a flow program will correspond to a jump in the Java method that the flow program models.

Conditional jump statement
The intuitive meaning of a conditional jump statement `condjump` $\overline{\mathtt{x}}$ t is: conditioned on the variables $\overline{\mathtt{x}}$, either jump to p_t, or continue execution at the next statement. We will see, however, that $\mathcal{D}$ is path-insensitive and does not depend on the variables $\overline{\mathtt{x}}$ in a conditional jump statement.

Any conditional jump statement in a flow program will correspond to a conditional jump in the Java method that the flow program models. For instance, a Java `if` statement such as

```
if (x < y)
  /* body */
```

translates to

```
condjump {x,y} φ
```

where ϕ is some statement position index corresponding to the start of `/* body */`.

5.2.2 Disappearing criticals function

Having introduced the flow language, I now make clear what I mean by *disappearing critical value*. After that, I describe the *disappearing criticals function*, $\mathcal{D}$.

[5] In principle, `none`-parameters could be factored out of any flow program without changing the meaning of the program with respect to the disappearing criticals function.

$$
\begin{aligned}
\mathcal{D}^{\perp}_{\mathtt{x}}[\![\mathtt{end}]\!] &= \emptyset \\
\mathcal{D}^{d}_{\mathtt{x}}[\![\mathtt{end}]\!] &= \{d\} \\
\mathcal{D}^{d}_{\mathtt{x}}[\![\mathtt{jump}_i\ t]\!] &= \mathcal{D}^{d}_{\mathtt{x}}[\![p_t]\!] \\
\mathcal{D}^{d}_{\mathtt{x}}[\![\mathtt{condjump}_i\ \overline{\mathtt{x}}\ t]\!] &= \mathcal{D}^{d'}_{\mathtt{x}}[\![p_{i+1}]\!] \cup \mathcal{D}^{d'}_{\mathtt{x}}[\![p_t]\!] \\
&\quad \textbf{where } d' = \perp \textbf{ if } \mathtt{x} \in \overline{\mathtt{x}} \textbf{ else } d \\
\mathcal{D}^{d}_{\mathtt{x}}[\![\mathtt{flow}_i\ \overline{w}]\!] &= X \cup \mathcal{D}^{d'}_{\mathtt{x}}[\![p_{i+1}]\!] \\
&\quad \textbf{where } X, d' = T(i, d, R(\overline{w}, \mathtt{x}), W(\overline{w}, \mathtt{x}))
\end{aligned}
$$

Table 5.2: Definition of the disappearing criticals function. The definitions of R and W are given in the text. The definition of T is given in Table 5.3.

Disappearing critical value
A disappearing critical value is a value that may be output from an `out`- or an `inout`-parameter in some `flow` statement, but that is not guaranteed to be consumed in a subsequent `flow` or `condjump` statement.

Intuition behind the disappearing criticals function
Now, let me describe the intuition of what the disappearing criticals function does. Here, let p be a flow program.

First, $\mathcal{D}^{\perp}_{\mathtt{x}}[\![p_i]\!]$ yields the set of statement position indices in p where a disappearing critical value may be output to `x`. Hence, if $\mathcal{D}^{\perp}_{\mathtt{x}}[\![p_{10}]\!] = \{12, 15\}$, it means that when 'executing' from p_{10}, a disappearing critical value may be output to `x` in p_{12} and p_{15}. That is, both p_{12} and p_{15} are flow statements with `x` as an `out`- or `inout`-parameter, and there are paths out of p_{12} and p_{15} with no `in`- or `inout`-parameter to consume these critical values.

The alternative $\mathcal{D}^{d}_{\mathtt{x}}[\![p_i]\!]$, where $d \neq \perp$, means the same as above, save for allowing an *incoming definition* d, referring to the position of a prior statement in p where a critical value was output to `x`. Hence, if $d = 9$ and $\mathcal{D}^{d}_{\mathtt{x}}[\![p_{10}]\!] = \{9, 11\}$, it means that when 'executing' from p_{10}, a disappearing critical value may be output to `x` in p_{11}, and a critical value that was output to `x` in p_9 may also disappear.

Recall that the first statement of p is p_0. Therefore, $\mathcal{D}^{\perp}_{\mathtt{x}}[\![p_0]\!]$ yields all statement positions in the flow program p where a disappearing critical value may be written to `x`.

5.2.3 Definition of $\mathcal{D}$

The definition of $\mathcal{D}$ is given in Figure 5.2. In the definition, a subscript i on a statement denotes the statement position index of that statement in the flow program. Below, I explain the details of each case in the definition.

The case $\mathcal{D}^{\perp}_{\mathtt{x}}[\![\mathtt{end}]\!]$:
If there is no incoming definition for x, and $\mathcal{D}$ has reached the end of the flow program, then $\mathcal{D}$ returns $\emptyset$.

The case $\mathcal{D}^{d}_{\mathtt{x}}[\![\mathtt{end}]\!]$:
On the other hand, if there *is* an incoming definition d for x, and $\mathcal{D}$ has reached the end of the flow program, then it means that a critical value was output to x in the statement with statement position index d, and that this value has not been consumed. In this case, $\mathcal{D}$ returns $\{d\}$.

The case $\mathcal{D}^{d}_{\mathtt{x}}[\![\mathtt{jump}_i\ t]\!]$:
When $\mathcal{D}$ reaches a unconditional jump, it continues at the jump target p_t, passing along any incoming definition position, or $\perp$, if there is no incoming definition position.

The case $\mathcal{D}^{d}_{\mathtt{x}}[\![\mathtt{condjump}_i\ \overline{\mathtt{x}}\ t]\!]$:
When $\mathcal{D}$ reaches a conditional jump, two things happen. First, if there is an incoming critical value in x, and x is mentioned in $\overline{\mathtt{x}}$, then $\mathcal{D}$ consumes the incoming definition. (As if $\mathcal{D}$ had reached a flow statement where x was mentioned in an in- or inout-parameter). Second, it branches to both jump targets and merges the results of analysis down both branches.

The case $\mathcal{D}^{d}_{\mathtt{x}}[\![\mathtt{flow}_i\ \overline{w}]\!]$:
Flow statements are the most complicated case. Here, $\mathcal{D}$ uses three auxiliary functions: The predicate functions R and W, and the mapping table T, which is defined in Table 5.3.

The function R tells whether a variable is consumed, given a parameter list. The function W tells whether a variable is output, given a parameter list. They are defined as follows:

$$R(\overline{w}, \mathtt{x}) = \exists(d, \mathtt{x}) \in \overline{w}.\ d \in \{\mathtt{in}, \mathtt{inout}\}$$
$$W(\overline{w}, \mathtt{x}) = \exists(d, \mathtt{x}) \in \overline{w}.\ d \in \{\mathtt{out}, \mathtt{inout}\}$$

Here, I use $\exists$ as an existential quantifier for program syntax.

d	**R**	**W**	**X, d′**
$\bot$	`true`	`true`	$\emptyset, i$
$\bot$	`true`	`false`	$\emptyset, \bot$
$\bot$	`false`	`true`	$\emptyset, i$
$\bot$	`false`	`false`	$\emptyset, \bot$
d	`true`	`true`	$\emptyset, i$
d	`true`	`false`	$\emptyset, \bot$
d	`false`	`true`	$\{d\}, i$
d	`false`	`false`	$\emptyset, \bot$

Table 5.3: The mapping table T. Here, d denotes an incoming definition position, and i denotes the statement position index of the active `flow` statement, when T is queried.

To illustrate R and W, consider the parameter list $\overline{w} = (\texttt{in a}, \texttt{inout b})$. Here, it is the case that $R(\overline{w}, \texttt{a})$, because `a` appears in an `in`-parameter. Further, it is the case that $R(\overline{w}, \texttt{b})$ and $W(\overline{w}, \texttt{b})$, since `b` appears in an `inout`-parameter. However, it is not the case that $W(\overline{w}, \texttt{a})$, because `a` only appears in an `in`-parameter, and not in an `out`- or `inout`-parameter.

Intuitively, the mapping table T defines what happens, with respect to critical values in `x`, when $\mathcal{D}$ processes a `flow` statement. Let me illustrate the operation of T with two examples. First, consider $\mathcal{D}^{\bot}_{\texttt{x}}[\![\texttt{flow}_2\ (\texttt{out v})]\!]$. Here, we have $W(\overline{w}, \texttt{v})$, but not $R(\overline{w}, \texttt{v})$. In this case the third row in T applies: no critical value disappear from `x` ($X = \emptyset$), but a new critical value is output to `x` ($d' = 2$). This case would correspond to the second statement in:

```
byte[] xs = new byte[256];
writeCriticalValue(xs);
```

Second, consider $\mathcal{D}^{d}_{\texttt{x}}[\![\texttt{flow}_3\ (\texttt{out v})]\!]$, with an incoming definition $d = 2$. Here, we also have $W(\overline{w}, \texttt{v})$ and not $R(\overline{w}, \texttt{v})$. But, because there is an incoming definition, the seventh row in T applies: a critical values does disappear here ($X = \{2\}$), and a new critical value is output ($d' = 3$). In other words, the incoming critical value in `x` is overwritten. This case would correspond to the third statement in:

```
byte[] xs = new byte[256];
writeCriticalValue(xs);
writeCriticalValue(xs);
```

Guaranteeing termination
As it stands, it is possible to input flow programs to $\mathcal{D}$ so that it never will terminate. For instance, the program `jump 0` will loop forever. Rather than attempting to address this with fix-point techniques [41], I step around the issue by positing the following requirement:

> For any flow language program input to $\mathcal{D}$, the target of any (conditional or unconditional) jump statement must be a *subsequent* statement, that is a statement in the flow program that has a larger statement position index than the jump statement.[6]

With this requirement, any evaluation of $\mathcal{D}$ has a definite, non-divergent value; for any evaluation of a statement p_i, we know that any recursive evaluations of $\mathcal{D}$ will be for statements with larger statement position indices than i, and that eventually `end` will be reached.

5.3 Translating Java methods into the flow language

Having described the flow language and the disappearing criticals function, and made precise what it means for a critical value to disappear, I now explain how to translate Java methods into the flow language.

5.3.1 Overview

The first steps of translation are equivalent to the translation steps used by the type-centered analysis, as described in Section 3.4.2: The JVM byte code of the method to be analysed is converted into Jimple, translated into SSA form, and finally converted into Simplified Jimple.

After these steps, the Simplified Jimple representation is translated into the flow language using a case-by-case mapping, which is defined below. This mapping must be *specialised* by what I term a *flow-analysis encoding*, with purpose to to specialise the translation for a specific instantiation of r_{crit}. This is conceptually similar to how the type-centered analysis must be specialised with a specific latent meaning database.

5.3.2 Translation rules

I now describe the rules for translating from the Simplified Jimple representation of a method (Section 3.4.2) into the flow language. Here, I also describe in which way a flow-analysis encoding specialises the translation process. Note that all the Simplified Jimple statement types not mentioned here are ignored in the translation.

[6] In particular, this requirement disallows backwards and same-statement jumps. It also disallows jumps 'off the end' of a program.

Return
A non-`void` return statement, such as

```
return x;
```

translates to

```
flow (d x)
jump φ
```

where d is a flow direction given by the flow-analysis encoding, and ϕ stands for the eventual length of the full translation. (In other words, it becomes a jump to `end`.)

A `void` return statement, such as

```
return;
```

translates to

```
jump φ
```

where ϕ is as for non-`void` return statements.

Unconditional jump
An unconditional jump statement, such as

```
jump i;
```

translates to

```
jump φ
```

where ϕ is an integer that stands for the eventual statement position matching the jump target position `i`.

Conditional jump
A conditional jump statement, such as

```
condjump (x0, x1, ...) i;
```

translates to

```
condjump (x0, x1,...) φ
```

where ϕ stands for the eventual statement position matching the jump target position `i`.

Method calls
A method call, such as, say,

```
f(z, x1, x2, x3);
```

or

```
y = f(z, x1, x2, x3);
```

translates to a `flow` statement where each flow parameter is determined by the flow-analysis encoding. Precisely, a flow-analysis encoding includes i) a mapping from known library method calls to lists of flow parameters, and ii) a rule for mapping unknown method calls to a flow statement. In the current implementation of the proof-of-concept tool, all unknown method calls are mapped to a flow statement with only `none`-parameters.

5.4 The analysis proper

I now formalise how the flow-centered analysis, as a whole, operates.

Definition 5.1 (Flow-centered analysis)
Let e be a flow-analysis encoding. The flow analysis function F_e is defined by the following algorithm:
Input:

- A Java method m.

Output:

- A map from each variable `x` in m to a set giving the statement positions in p (the flow program corresponding to m) where a disappearing critical value may be output to `x`.

Algorithm:

1. Using the translation rules described in Section 5.3, specialised with the flow-analysis encoding e, translate m into a flow program p.
2. Output $\{ v \mapsto \mathcal{D}^{\perp}_{\mathtt{x}}[\![p_0]\!],$ for each variable `x` in m $\}$

5.4.1 Using the flow-centered analysis

While the flow-centered analysis is designed to find latent domain rule violations of r_{crit}, as motivated in Section 5.1, it turns out that is another interesting way to use the analysis: several runs of the flow-centered analysis can be used to (approximately) enforce an API contract.[7] I first describe how to use the standard technique, before I comment on the alternative technique, which goes beyond r_{crit}.

Standard technique: Enforcing an instantiation of r_{crit}

The main technique for using the flow-centered analysis is to use *one* flow-analysis encoding e, designed to specialise the analysis for one instantiation of r_{crit}. This

[7] I did not realise this before I started experimenting with the analysis.

All critical values are used in a satisfactory way	$\Rightarrow$	$F_e(m)(\mathtt{x}) = \emptyset$ for all variables x.
Some critical value is not used in a satisfactory way	$\Rightarrow$	$F_e(m)(\mathtt{x}) \neq \emptyset$ for some variable x.

Figure 5.1: Correspondence between use of critical values (Section 5.1.2) and the output of the flow-centered analysis. Assume here that m is a Java method and that e is a flow-analysis encoding corresponding to some instantiation of r_{crit}.

technique can be used to find the flaw in the `network.send` example that I discussed in Section 5.1. I demonstrate this technique in Section 7.3.1, where I analyse the `network.send` example, and in Section 7.3.3 where I demonstrate analysis of a similar flaw in a third-party test suite.

The standard technique is illustrated in Figure 5.1, where I use the notion of 'used in a satisfactory way' that I introduced in the start of the chapter.

Alternative technique: Asserting an API contract

Several instances of the flow-centered analysis can be run in sequence to assert that certain types of requirements to clients of an API are obeyed. For example, to assert that any call to a method `lock` is followed by a call to a method `unlock`. I demonstrate this technique in Section 7.3.2.

Chapter summary and continuation

In this chapter, I described the flow-centered analysis, the second of the two analyses I present in the thesis. First, I motivated the analysis by showing examples of flaws that the type-centered analysis can not identify. Then, I introduced the notion of *critical* values, and, further, that of *disappearing* critical values. I then described a technique for finding disappearing critical values using a custom language, the *flow language*, paired with a custom semantics, the *disappearing criticals function*. Further, I described how to translate Java methods from Simplified Jimple (cf. Section 3.4.2) into the flow language, using a *flow-analysis encoding*. Finally, I explained how to use the flow-centered analysis to enforce instantiations of r_{crit} (the main use case of the analysis), and noted that it also can be applied to assert that clients of an object-oriented API obey certain types of requirements.

In Section 7.3 of Chapter 7, the validation chapter, I perform an experimental validation of the flow-centered analysis. First, in Section 7.3.1, I demonstrate how

to use the flow-centered analysis to check the `network.send` example given in Section 5.1. Second, in Section 7.3.2, I demonstrate how to use the flow-centered analysis to enforce certain types of requirements to clients of an object-oriented API. Third, in Section 7.3.3, I use the flow-centered analysis to identify violations of r_{crit} in test cases from a third-party test suite. Further, in Chapter 8, I compare the flow-centered analysis to closely related work. Finally, in Chapter 9, I discuss the flow-centered analysis, compare it to the type-centered analysis, and present ideas for future work.

CHAPTER 6

Implementation of the proof-of-concept tool

In this (very short) chapter, I give a high-level overview of the implementation of the proof-of-concept tool, and describe how to run the system to replicate the results in Chapter 7.

6.1 Overview of the system

The proof-of-concept tool, or, hereafter, the *system*, is implemented in Scala [43]. The core of the system comprises about 1500 lines of Scala code; in addition, there are about 500 lines of experiment-specific Scala code. The system is split into several packages. The important packages are named: `sootfacade`, `typeanalysis`, and `flowanalysis`. In addition to these packages there is a top-level (main) package, which contains code for configuring the system and running the experiments seen in Chapter 7, and a package named `util`, containing various utility methods. Further, there is a Java implementation of the alternative constraint solver, `AlternativeConstraintSolver`, and a collection of unit tests. In the following sections I give an overview of the packages `sootfacade`, `typeanalysis`, and `flowanalysis`. I do not discuss other parts of the system in depth.

The `sootfacade` package

The `sootfacade` package consists of two files. The file `types.scala` contains the implementation of the Simplified Jimple intermediate representation (Section 3.4.2). As noted in Section 3.4.2, the grammar used in the implementation is somewhat more compact than the one listed in the thesis. There are also other minor differences, such as that the `condjump` statement is called `IfStmt` in

the implementation. The file `JimpleSimplifier.scala` contains one class, `JimpleSimplifier`, which performs translation from Jimple into Simplified Jimple, as described in Section 3.4.2.

The `typeanalysis` package

The package `typeanalysis` implements the core parts of the type-centered analysis. Most importantly, the class `TypingAlgorithm` implements the typing algorithm, as described in Section 4.2. The file `types.scala` contains classes such as `EqConstraint` and `LeqConstraint`, representing typing constraints. The domain rules r_{cs} and r_{conf} are specified in `domain_rules.scala`. The file `latent_meaning_repr.scala` contains classes for representing parts of a latent meaning database: `ConstLatentMeaningDb`, `APILatentMeaningDb`, and `NamingLatentMeaningDb`.

The proof-of-concept implementation of the type-centered analysis is restricted in two ways, with respect to the description given in Chapter 3. First, parts of the implementation only support domain rules with two-element type sets, such as r_{cs} and r_{conf} (Section 3.5). Second, latent meaning databases must be written using an (internal) domain specific language that slightly limits the flexibility in outputting constraints, with respect to the description in Section 3.4. Both these restrictions are relatively simple to lift.

The `flowanalysis` package

The package `flowanalysis` implements the core parts of the flow-centered analysis. Most importantly, the class `FlowAnalyser` implements the disappearing criticals function. (Specifically, $\mathcal{D}$ is implemented by the private method `find-DisappearingForVar`.) The class `FlowModelTranslator` performs translation from Simplified Jimple to flow programs, as described in Section 5.3. The file `flow_ana_encodings.scala` contains a class `FlowAnaEncoding` for representing flow-analysis encodings. Note that some of the flow-analysis encodings used for the experiments in Chapter 7 are defined in the file `demo_experiments.-scala` in the top-level (main) package.

6.2 Re-running the experimental validation

With a recent Scala version and the proof-of-concept tool installed, one can re-run the experiments in Chapter 7 by issuing

```
$ sbt run
```

from the folder `analysis`. The proof-of-concept tools will then output experimental results to `analysis/experiment/out/`.

CHAPTER 7

Experimental validation

In this chapter, I aim to build support towards my research hypothesis through a practical demonstration of the analyses. To further support the hypothesis, I follow the experimental validation with qualitative arguments in Chapter 8 and Chapter 9.

The structure of the chapter is as follows: First, in Section 7.1, I comment on my experimental validation strategy. Second, in Section 7.2, I report on the experimental validation of the type-centered analysis. Third, in Section 7.3, I report on the experimental validation of the flow-centered analysis.

7.1 Experimental validation strategy

To lend support to the research hypothesis I posited in Section 1.1, I must demonstrate that the analysis techniques I present are useful for finding violations of a variety of interesting domain rules.

One persuasive strategy would be to demonstrate that the analyses can identify latent domain rule violations in a representative software corpus (cf. Section 2.2). While I, ideally, would have performed such corpus-based validation, time constraints forced me to look to an alternative validation strategy. This is a potential weakness of my work, which I return to discuss in Section 9.6.

To guide my design of a sound experimental validation within my constraints, I looked to Shaw's well-known text 'Writing Good Software Engineering Research Papers' [56], where she discusses the merits of different validation strategies. From analyses of the relative acceptance ratios for the strategies used in submissions to ICSE 2002[1], Shaw reports that '[the] most successful kinds of validation were based on analysis and real-world experience', but that '[well-chosen] examples

[1] International Conference on Software Engineering.

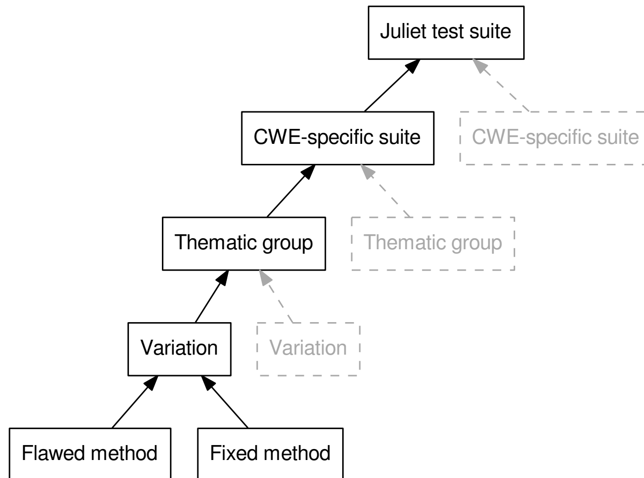

Figure 7.1: Structure of the Juliet test suite. Gray nodes and edges represent parts of the hierarchy not drawn.

were also successful'. Heeding this advice, I chose a primarily example-based validation strategy.

In addition to using examples, I include another, primarily quantitative validation strategy; I run the analyses on a subset of cases from the *Juliet* test suite [6], which was designed by a group of NSA researchers as a means to evaluate static analyses. I describe the Juliet test suite below.

7.1.1 The Juliet test suite

Juliet is a test suite developed at NSA's Center for Assured Software with purpose to '[assess] the effectiveness of static analyzers and other software-assurance tools' [6].

The test suite has a hierarchical structure, illustrated in Figure 7.1. Topmost, it divides into 112 smaller suites, each specific to a *Common Weakness Enumeration* (CWE) entry.[2] Further, each CWE-specific suite divides into a number of *thematic groups* (my term), each pertaining to one way a weakness may manifest. For instance, within the suite corresponding to CWE-78: OS Command Injection, one thematic group concerns OS command injection specifically through `java.io.File` objects. Further again, within each thematic group, there are numerous *variations* (my term). Each variation maps to one or two `class` files, containing several methods—at least one method where the weakness is present and at least one method where the weakness has been fixed. I term these *flawed* and

[2]The CWE database is maintained by the MITRE Corporation (an American non-profit) and accessible at `https://cwe.mitre.org/`.

fixed methods. Including both flawed and fixed methods is valuable; it allows tool developers to check that they correctly identify flawed methods (true positives), but also to ensure that they do not mark non-flawed methods as flawed (false positives).

The Juliet test cases are predominantly machine-generated from a templates and have a fairly regular structure. As such, Juliet is not a corpus of 'fully independent' test cases; the variations are fairly similar, both between and within thematic groups.

Listings
To simplify the presentation, I have reformatted and simplified the Juliet source code listings seen in this chapter. In particular, I have removed some tangential comments, and removed superfluous syntax, such as brackets around single-line `if`-statement bodies. The listings are semantically equivalent to those input to the analyses.

7.2 Experimental validation for the type-centered analysis

7.2.1 Slice-of-life example: E-vote 2011

In this section, I demonstrate how the type-centered analysis can identify a latent domain rule violation of r_{cs}, taken from the source code of an electronic voting system.

Background

Before the Norwegian municipal elections of 2011, the Norwegian government published the source code repository of an electronic voting system that would be evaluated during the elections [44].[3] By granting the public access to the source code, the government aimed to build trust in the security and correctness of the system as a whole, and to allow interested parties to review the system's source code for bugs or other issues.

Example for analysis

The following method[4] is found in the source code repository published before the elections. The method is defined on class `Request`, in the package `com.scytl.evote.auditing.tpm.bizz`.

[3]Note though, that the source code actually used in the elections, was a later version, and that the source code of the later version was not made public until after the elections.

[4]The method is not listed verbatim, but is semantically equivalent to the one in the software repository. I have reformatted the code, and I use the full name of `java.security.SecureRandom`.

```
private String createChallenge() {
  java.security.SecureRandom sr =
    new java.security.SecureRandom();
  return new Long(Math.round(sr.nextLong())).toString();
}
```

Without scrutiny, the method may appear to be a fairly innocent challenge creator. However, due to subtleties related to Java's numeric coercion rules, the method is no more random than a coin toss [44]. Indeed, it virtually always returns the string representation of either `Integer.MIN_VALUE` or `Integer.MAX_VALUE`. In other words, this methods contains a latent domain rule violation of $r_{\textsf{cs}}$.

I now demonstrate that the type-centered analysis can find this latent domain rule violation.

Analysis setup

To configure and run the type-centered analysis on this method, I add annotations for the method's library calls to $\mathcal{K}_{\textsf{cs}}$. In particular, I add annotations for `SecureRandom`, for the constructor of `Long`, and for the `toString` method defined on `Long`.

Results

Table 7.1 shows the Simplified Jimple representation of the method, all constraints output by $\mathcal{K}_{\textsf{cs}}$, the output of the typing algorithm, and the judgement of the type-centered analysis as a whole. As seen, the typing algorithm is unable to find a typing, and the type-centered analysis (correctly) concludes that the method contains a latent domain rule violation of $r_{\textsf{cs}}$.

Discussion

To understand how the type-centered analysis processes this method, one must first look at the Simplified Jimple representation of the method's byte code (listed in the top of Table 7.1). First, the fairly complex statement

```
return new Long(Math.round(sr.nextLong())).toString();
```

translates to seven Simplified Jimple statements. Second, there are synthesised variables (cf. Section 3.4.2) such as `$l0` and `$f0`, which represent intermediate values.

I now work through the constraints generated for the method, and illustrate why the type-centered analysis concludes that the method contains a latent domain rule

SIMPLIFIED JIMPLE

```
this = $this
<java.security.SecureRandom: void <init>()>($r0)
sr = $r0
$l0 = <java.security.SecureRandom: long nextLong()>(sr)
$f0 = (float) $l0
$i1 = <java.lang.Math: int round(float)>($f0)
$l0_1 = (long) $i1
<java.lang.Long: void <init>(long)>($r1, $l0_1)
$r2 = <java.lang.Long: java.lang.String toString()>($r1)
return $r2
```

METHOD CALL, ASSIGNMENT, OPERATOR, AND CAST SIGNS

Statement	Constraint
`this = $this`	`$this` $\leq$ `this`
`<java.security.SecureRandom: void <init>()>($r0)`	`$r0` $= \neg$cs
`sr = $r0`	`$r0` $\leq$ `sr`
`$l0 = <java.security.SecureRandom: long nextLong()>(sr)`	`$l0` $=$ cs
`$f0 = (float) $l0`	`$f0` $= \neg$cs
`$i1 = <java.lang.Math: int round(float)>($f0)`	`$i1` $= \neg$cs, `$f0` $= \neg$cs
`$l0_1 = (long) $i1`	`$l0_1` $= \neg$cs
`<java.lang.Long: void <init>(long)>($r1, $l0_1)`	`$l0_1` $\leq$ `$r1`
`$r2 = <java.lang.Long: java.lang.String toString()>($r1)`	`$r1` $\leq$ `$r2`

NAME SIGNS
`$r2` $=$ cs

OTHER SIGNS
`this` $= \neg$cs

TYPING
(typing error)

JUDGEMENT
Violated

Table 7.1: Type-centered analysis overview for the E-vote secure challenge creator.

violation. To simplify the presentation, I ignore constraints that are not relevant for identifying the violation.[5]

First, $\mathcal{K}_{\text{cs}}$ recognises

```
$l0 = <java.security.SecureRandom: long nextLong()>(sr)
```

as a library method call and outputs the equality constraint `$l0` $=$ cs.

Second, for the cast

```
$f0 = (float) $l0
```

$\mathcal{K}_{\text{cs}}$ outputs the equality constraint `$f0` $= \neg$cs, requiring that `$f0` can not be a cryptographically secure challenge.

[5] These include the constraints on `$this`, `this`, and `sr`.

Third, $\mathcal{K}_{\mathsf{cs}}$ does not recognise the method call

```
$i1 = <java.lang.Math: int round(float)>($f0)
```

For this reason, it outputs cautious constraints for all the variables involved, requiring both `$i1` and `$f0` to have type $\neg\mathsf{cs}$.[6]

Fourth, for the cast

```
$l0_1 = (long) $i1
```

$\mathcal{K}_{\mathsf{cs}}$ outputs the equality constraint $\texttt{\$l0_1} = \neg\mathsf{cs}$, requiring that `$l0_1` can not be a cryptographically secure challenge.[7]

Fifth, $\mathcal{K}_{\mathsf{cs}}$ recognises

```
<java.lang.Long: void <init>(long)>($r1, $l0_1)
```

as a library method call. In this case it outputs the (precise) ordering constraint $\texttt{\$l0_1} \leq \texttt{\$r1}$. In other words, it only disallows a typing where the new `Long` object `$r1` is typed as cs, but its value is taken from an **`long`** primitive typed as $\neg\mathsf{cs}$.

Sixth, $\mathcal{K}_{\mathsf{cs}}$ also recognises

```
$r2 = <java.lang.Long: java.lang.String toString()>($r1)
```

as a library method call. Here, it outputs a similar constraint as for the statement discussed in step five. Together, these constraints say that `$r2`, the `String` representation of the `Long` object `$l0_1`, may be typed as cs only if `$l0_1` is typed as cs.

Finally, $\mathcal{K}_{\mathsf{cs}}$ recognises the substring `challenge` in the method name, and therefore outputs the equality constraint $\texttt{\$r2} = \mathsf{cs}$, for the return variable `$r2`.

To see that there is no typing for this method, it sufficies to consider the constraints generated for the fourth, fifth, and sixth statements I discussed above, and the constraint generated for the method name. Together these form the constraint set

$$\texttt{\$l0_1} = \neg\mathsf{cs} \leq \texttt{\$r1} \leq \texttt{\$r2} = \mathsf{cs}$$

[6]This is overly stringent, of course. It would have been sufficient to require only `$i1` to be $\neg\mathsf{cs}$, and not output any constraint for `$f0`, as the *primitive* value `$f0` can not be modified in the call to `Math.round`. The analysis is not precise enough to discern between primitive and reference types, though.

[7]This is also overly stringent. If the analysis knew more about Java types, it could have output a more optimistic ordering constraint here, as for an assignment statement.

The crux is that it is not possible to assign type ¬cs to `$l0_1` and type cs to `$r2`, when the type assigned to `$l0_1` is required to be ordered less than the type assigned to `$r2`, by the ordering defined in t_{conf} (cf. Section 3.5.1). Hence, the type-centered analysis concludes that the method contains a latent domain rule violation of r_{cs}.

Discussion

The reader may object that the type-centered analysis does not really 'get at' the core issue here, since the first element in the set of constraints that prohibits a typing is a **`int`**-to-**`long`** cast statement that $\mathcal{K}_{\mathsf{cs}}$ constrains too stringently. For the sake of argument, assume that an improved version of the analysis was able to output the more precise constraint

$$\texttt{\$l0_1} \leq \texttt{\$i1}$$

for the **`int`**-to-**`long`** cast statement. Then, the set of constraints prohibiting a typing would extend back to `$i1`, the return variable of the `Math.round` statement:

$$\texttt{\$i1} = \neg\mathsf{cs} \leq \texttt{\$l0_1} \leq \texttt{\$r1} \leq \texttt{\$r2} = \mathsf{cs}$$

and we would have the same problem to find a typing.

Analysis of a corrected version

In a later source code release, published after the 2011 elections, the faulty method had been corrected. The corrected method reads as follows:

```
private String createChallenge() {
  java.security.SecureRandom sr =
    new java.security.SecureRandom();
  return new Long(sr.nextLong()).toString();
}
```

Table 7.2 shows the analysis overview for the corrected method. In this case, the typing algorithm finds a solution satisfying all constraints, and the analysis concludes (correctly) that the method does not contain a latent domain rule violation of r_{cs}.

Regarding the constraints and the typing, note that there is a 'chain' of constraints running from the synthesised variable representing the output value of `SecureRandom.nextInt` to the return variable `$r2`:

$$\texttt{\$l0} = \mathsf{cs} \leq \texttt{\$r1} \leq \texttt{\$r2} = \mathsf{cs}$$

and that all the variables in this 'chain' are typed as cs.

SIMPLIFIED JIMPLE

```
this = $this
<java.security.SecureRandom: void <init>()>($r0)
sr = $r0
$l0 = <java.security.SecureRandom: long nextLong()>(sr)
<java.lang.Long: void <init>(long)>($r1, $l0)
$r2 = <java.lang.Long: java.lang.String toString()>($r1)
return $r2
```

METHOD CALL, ASSIGNMENT, OPERATOR, AND CAST SIGNS

Statement	Sign
`this = $this`	`$this` $\leq$ `this`
`<java.security.SecureRandom: void <init>()>($r0)`	`$r0` $= \neg$cs
`sr = $r0`	`$r0` $\leq$ `sr`
`$l0 = <java.security.SecureRandom: long nextLong()>(sr)`	`$l0` $=$ cs
`<java.lang.Long: void <init>(long)>($r1, $l0)`	`$l0` $\leq$ `$r1`
`$r2 = <java.lang.Long: java.lang.String toString()>($r1)`	`$r1` $\leq$ `$r2`

NAME SIGNS
`$r2` $=$ cs

OTHER SIGNS
`this` $= \neg$cs

TYPING
`$l0` $\mapsto$ cs `$r0` $\mapsto \neg$cs `$r1` $\mapsto$ cs `$r2` $\mapsto$ cs `$this` $\mapsto \neg$cs `sr` $\mapsto \neg$cs `this` $\mapsto \neg$cs

JUDGEMENT
Not violated

Table 7.2: Type-centered analysis overview for the corrected version of the E-vote secure challenge creator.

7.2.2 Juliet experiment for r_{conf}

Introduction

In this experiment, I analyse the Juliet suite specific to CWE-526: Info Exposure Environment Variables [8], which is a special case of r_{conf}.

The theme of CWE-526 is that an environment variable, assumed to be confidential, is exposed to an output channel, assumed not to preserve confidentiality. As an example, one of the simplest flawed methods in the Juliet sub-suite corresponding to this vulnerability reads as follows:

```
public void bad(HttpServletRequest request,
                HttpServletResponse response)
                throws Throwable {
  /* FLAW: environment variable exposed */
  response.getWriter().println(
    "Not in path: " + System.getenv("PATH"));
}
```

The Juliet suite corresponding to CWE-526 comprises two thematic groups. The first thematic group is `Servlet`-specific, and uses `HttpServletResponse` objects for output. In the second group, output is written to `IO`, a Juliet-specific class which is used to model standard input and output. Within each thematic group there are 17 variations: four variations contain one flawed and one fixed method; the remaining 13 contain one flawed and two fixed methods.

Analysis setup

First, I add annotations for the relevant library methods to the latent meaning database for r_{conf}. Concretely, I add selected annotations for `System.getEnv`, `HttpServletResponse`, and the class `IO`. I then input all 34 variations, for a total of 94 methods, to the type-centered analysis.

Results and discussion

The result matrix for this experiment reads as follows:

	True	**False**
Positives	34	0
Negatives	60	0

As evident, the type-centered analysis correctly identifies all cases in the test set. This is not surprising; while there is variance in flow patterns among the test cases, the type-centered analysis ignores these[9], and outputs (largely) similar constraint sets for all cases.

[8] This is the name used in the Juliet test suite. In the CWE database it is named Information Exposure Through Environmental Variables.

[9] I return to this point in Section 9.1.

7.3 Experimental validation for the flow-centered analysis

7.3.1 Checking the `network.send` example

I now demonstrate how to use the flow-centered analysis to check the `network.send` example given in Section 5.1.

Method
I use the following source code, where `Network` is a mock implementation.

```
import java.util.*;

class test {
  private Queue<byte[]> msgQueue = new ArrayDeque<>();
  private Network network = new Network();

  void sendNext() {
    byte[] msgBytes = msgQueue.poll();
    network.write(msgBytes);
  }

  /* mock implementation */
  static class Network {
    int write(byte[] data) { return -1; }
  }
}
```

Flow-analysis encoding
What we want to assert here is that the return value of `network.send` is checked. We may see this as an instantiation of r_{crit} where the critical data is the return value of `network.send`. To specialise the flow-centered analysis for finding this violation, I create a flow-analysis encoding where a method call such as

```
z = network.send(xs);
```

translates to

```
flow(out z, none network, none xs)
```

Recall from Section 3.4.2 that in the translation process from JVM byte code to Simplified Jimple, synthesised variables are added for all ignored return values. Hence, a call to `network.send` where the return value is ignored, such as

```
network.send(xs);
```

translates to

```
$ignoredK = network.send(xs);
```

where `K` is some integer. It is into this synthesised variable that the flow-centered analysis will track the disappearing critical value.

Simplified Jimple representation
The method translates to the following Simplified Jimple representation:

```
this = $this
$r0 = #msgQueue
$r1 = <java.util.Queue: java.lang.Object poll()>($r0)
msgBytes = (byte[]) $r1
$r2 = #network
$ignored0 = <test$Network: int write(byte[])>($r2, msgBytes)
return
```

As seen, although the return value of `network.send` is ignored in the Java-level code, it is explicitly modeled in the synthesised variable `$ignored0`.

Flow program
Using the flow-analysis encoding described above, the Simplified Jimple representation further translates[10] into the following flow program.

```
2: flow (none $r1, none $r0)
5: flow (out $ignored0, none $r2, none msgBytes)
6: jump 7
```

Disappearing critical values
The analysis output is as follows:

```
5: $ignored0
```

The flow-centered analysis reports that there is one disappearing critical value, which is output to `$ignored0` in the flow program statement with statement position index 5. This disappearing critical value corresponds to the ignored return value of `network.send`.

[10]Note, incidentally, that most of the Simplified Jimple statements are ignored in the translation.

7.3.2 Asserting correct usage of an object-oriented API

In addition to asserting instances of r_{crit}, the flow-centered analysis can approximately assert certain domain rules that are about how to correctly use an object-oriented API. In this section I demonstrate this use case.

Consider a resource class such as the following:

```
class Resource {
  void lock() { /* ... */ }
  void process() { /* ... */ }
  void unlock() { /* ... */ }
}
```

Assume this class requires client code to obey the following requirements:

- Any call to `lock` must be paired with a call to `unlock`.
- Any call to `lock` must be followed by a call to `process`. (After a client has locked a resource, they must process it.)
- Any call to `process` must be wrapped inside a matching pair of calls to `lock` and `unlock`.

The flow-centered analysis can fully enforce requirements 1 and 2, and partially enforce requirement 3 (I will explain where it falls short).

The main idea when enforcing such requirements (on any class) is to use *several* flow-analysis encodings where a critical value is output to the call target of each API call; that is, the **this** variable, with respect to the API class.

Concretely, to enforce the client-code requirements for `Resource` we need to define two flow-analysis encodings: First, a flow-analysis encoding where we specify that `lock` outputs a critical value to the **this** variable, and that `process` is the only method that consumes a critical value from the **this** variable. Second, a flow-analysis encoding where we specify that `process` outputs a critical value to the **this** variable, and that `unlock` is the only method that consumes a critical value from the **this** variable. Call these flow-analysis encodings e_1 and e_2, respectively. Then, to approximately assert that the requirements are followed, we can run the flow-centered analysis two times, once specialised with e_1 and once specialised with e_2.

Analysis of a method that violates the requirements

Let me demonstrate how to use the flow-centered analysis, specialised with the flow-analysis encodings e_1 and e_2 described above, to identify a method that does not obey the requirements to clients of `Resource`.

Consider the following method:

```
static void useResource(Resource res, boolean flag) {
  res.lock();
  if (flag) {
    res.process();
    /* more code, without res.unlock() */
    return;
  }
  /* more code */
  res.process();
  res.unlock();
}
```

Note here that if `flag` is set, the method may return without calling `unlock` on `res`, and thus violate the first requirement to clients of `Resource`.

Analysis details

First, I input the method to the flow-centered analysis specialised with e_1, which enforces that any call to `lock` must be followed by a call to `process`. In this run, the flow-centered analysis finds no disappearing critical values. This is as expected; whether `flag` is set or not, the method always calls `process` after the call to `lock`.

Second, I input the method to the flow-centered analysis specialised with e_2, which enforces that any call to `process` must be followed by a call to `unlock`. In this run, the flow-centered analysis *does* find a disappearing critical values, as we would expect.

Let me illustrate what happens in the second run. First, using e_2, the method translates (cf. Section 5.3) into the following flow program:

```
2: flow (none res)
3: condjump (flag, $const0) 6
4: flow (out res)
5: jump 9
6: flow (out res)
7: flow (in res)
8: jump 9
```

In this flow program, the `flow` statement with statement position index 4 corresponds to `res.process()` inside the **`if`** block in the method, and the `jump` statement with statement position index 5 corresponds to the **`return`** statement inside the same block. (Recall that a **`return`** statement translates into a `jump` to `end`.) Crucially, on the path $2 \mapsto 3 \mapsto 4 \mapsto 5 \mapsto$ `end` the critical value output to `res` in statement 4 disappears. It is this disappearing critical value that indicates that there is a call to `process` not necessarily followed by a call to `unlock` in the

method. The output of the flow-centered analysis is as follows:

```
4: res
```

That is, it reports that there is a disappearing critical value output to `res` in statement four.

Caveats

Still, as I noted in the introduction of the section, the flow-centered analysis can not fully enforce requirement three. In particular, it can not understand that a method missing a `lock`, such as

```
static void useResource(Resource res) {
  res.process();
  res.unlock();
}
```

is incorrect. However, there is an relatively simple way to fix this: adapting the flow-centered analysis so it can run backwards as well as forwards. I return to this point in Section 9.3.

7.3.3 Juliet experiment for r_{crit}

Introduction

For this experiment, I analyse the Juliet suite specific to CWE-252: Unchecked Return Value .

As the name suggests, CWE-252 concerns flaws where a return value is critical, and must be checked. One of the simplest flawed methods in the Juliet sub-suite corresponding to CWE-252 reads as follows:

```
public void bad() throws Throwable {
  FileInputStream streamFileInput = null;

  try {
    int bytesToRead = 1024;
    byte[] byteArray = new byte[bytesToRead];

    streamFileInput = new FileInputStream("c:\\file.txt");
    streamFileInput.read(byteArray);

    /* FLAW: Do not check the return value of read() */
    IO.writeLine(new String(byteArray, "UTF-8"));
  } ...
}
```

The flaw is that the return value of `streamFileInput.read` is not explicitly checked. In the fixed cases a test like the following has been added:

```
if (numberOfBytesRead < bytesToRead) {
  IO.writeLine("Could not read " + bytesToRead + " bytes.");
}
```

The Juliet suite corresponding to CWE-252 comprises one thematic group, specific to `FileInputStream` objects. In this thematic group four variations contain one flawed method and one fixed method, while the remaining 13 contain one flawed and two fixed methods.

Analysis setup

First, I add annotations for the relevant library methods to a flow-analysis encoding specialised for asserting that the return value of `FileInputStream` is checked. In this case, it is only necessary to annotate library calls to `FileInputStream`, and the flow-analysis encoding is very compact. I then input all 17 variations, for a total of 47 methods, to the flow-centered analysis.

Results and discussion

The result matrix for this experiment reads as follows:

	True	**False**
Positives	17	1
Negatives	29	0

As evident, the flow-centered analysis correctly identifies all but one case. Recall from Section 5.2.3 that the flow-centered analysis does not support analysis of methods with backwards jumps. In the current implementation of the proof-of-concept tool, methods with such jumps are automatically marked as buggy.[11]

Chapter summary and continuation

In this chapter, I performed an experimental evaluation of the type-centered analysis and the flow-centered analysis. I evaluated both analyses on test cases from the Juliet test suite, and on a selection of other examples. In particular, I showed that the type-centered analysis can identify a latent domain rule violation in the source code of an electronic voting system.

In the next chapter, I compare the analyses to closely related work. Further, in Chapter 9, I discuss the analyses and relate them to each other.

[11]It would have been more precise to include an *analysis error* result type in the proof-of-concept tool.

CHAPTER 8

Related work

In this chapter, I pick up the thread from Chapter 2 and discuss research relating directly to the analyses I present. Note that my aim here is not to argue that my techniques are more precise than related works. Rather, I aim to discuss i) ideas in my work that are comparable to ideas seen in related work, ii) ideas I include that are not in related work, and iii) ways in which the techniques I present are simpler than techniques described in related work.

The chapter is structured as follows: First, in Section 8.1, I discuss Engler et al.'s work on *bugs as deviant behaviour*, which shares similarities with the analyses I present. Second, in Section 8.2, I discuss work on *pluggable type systems*. Third, in Section 8.3, I discuss the *FindBugs* tool, which, like both the analysis I present, uses simple intraprocedural techniques. Finally, in Section 8.4, I discuss Høst and Østvold's work on *naming bugs*.

8.1 Bugs as deviant behaviour

Engler et al. [21] present a two-step technique for finding bugs, based on first extracting so-called 'beliefs' from source code, and then checking whether any of these beliefs contradict each other. As examples of the first step, 'a dereference of a pointer `p`, implies a belief that `p` is non-null' [21], whereas a conditional statement checking whether a pointer `p` is null would imply a belief that `p` could be null. If these example beliefs arose together, Engler et al.'s technique would identify them as contradicting each other.

This technique is largely comparable to the type-centered analysis. First, what Engler et al. term beliefs is comparable to what I call latent meaning, and the process of extracting beliefs from source code is similar to the process I describe for extracting latent meaning from various signs in source code. Second, the idea

of gathering beliefs and trying to find contradictions among these is conceptually similar to how the type-centered analysis first generates typing constraints from latent meaning, and then tries to find a typing which satisfies these.

While there are conceptual similarities, Engler et al. use static analysis techniques than are much more advanced than the ones I use, and their analysis would outperform the ones I present if put to a head-to-head test. On the other hand, the internals of their techniques are therefore more complex than the ones I present. They use a custom version of the GNU `g++` compiler, which is augmented with path-dependent static analysis.[1]

One difference from the type-centered analysis, is that, as far as I understand, Engler et al. use no comparable notion to the ordering I apply to type sets (cf. Definition 3.4). Using such an ordering is essential for checking domain rules such as r_{conf}, where it is acceptable to treat one type of data as another (e.g. to treat non-confidential data as confidential), but unacceptable to treat the second kind of data as the first (e.g. to treat confidential data as non-confidential).

Importantly, Engler et al. use machine learning techniques to automatically infer rules[2] to be checked. This significantly reduces the amount of manual labour needed for large-scale analysis. Using machine learning to uncover domain rules is an interesting avenue for future work, which I discuss in Section 9.5.

8.2 Pluggable type systems

The purpose of *pluggable type systems* is to '[allow] multiple type systems to be used simultaneously and/or sequentially for various semantic analyses' [9]. A pluggable type system operates independently of a programming language's built-in type system, so that variables (or values) that have the same type in a programming language's own type system can have different types in a pluggable type system, and vice versa.

To use a pluggable type system one annotates code using syntax that does not influence the main semantics of a program. For example, to add 'pluggable types' to Java variables, one may use comments, such as in

```
/*#NonNull*/ List<Integer> = ...;
```

or one may use Java annotations, such as in

```
@NonNull String str;
```

The type-centered analysis is essentially a pluggable type system. For example, when we use the type-centered analysis to assert r_{conf} (Section 3.5.2) for some method, what we are doing is to 'plug' the types high and low into the method,

[1] Although this custom compiler is rather complex, the *checkers*, which are input to this compiler, are expressed in a very elegant domain specific language, which Engler et al. `metal` [15].

[2] What they term rules is comparable to what I call domain rules.

and see if the method has a valid typing in the type system defined by the type-centered analysis (Section 3.3).

The most comprehensive implementation of pluggable type systems is the Checker Framework [47]. Checker uses relatively sophisticated static analysis techniques, and integrates tightly with the Java compilation tool chain. One distinction between the techniques I present and Checker is that Checker, as far as I understand, does not exploit information found in names.[3]

8.3 FindBugs

FindBugs [5] is a widely used free and open source bug checker for Java. It is comparable to the techniques I present in that it uses simple, unsound and incomplete analysis techniques, with aim to catch 'low-hanging fruit'.

FindBugs uses so-called *bug detectors* to find instances of various bug patterns. To add support for a new bug pattern, one must write code that examines JVM byte code for evidence of a bug. This can be a laborious process, and requires users to have a good grasp of JVM byte code. I hypothesise that some of FindBugs's bug detectors would be simpler to implement with the analysis framework I present, which operates on a a higher level of abstraction. For instance, I hypothesise that the flow-centered analysis can assert a similar rule to FindBugs' bug detector OS: Method may fail to close stream[4], by using the ideas for enforcing API contracts that I described in Section 7.3.2.

FindBugs includes some convenient abstractions for simplifying the writing of detectors. Still, it is inherently coupled to JVM byte code, a low-level, stack-based language with about 150 different instructions [39]. The analyses I present work on a higher level of abstraction; for one thing, they are defined on top of Jimple [64] (and further, on top of Simplified Jimple), which abstracts away many of the intricacies of JVM byte code. In this sense, my analyses are simpler than FindBugs.

Importantly, FindBugs does not actively use information about names when it looks for bugs. (It has, however, a few bug detectors that are name-specific, such as Nm: Class names shouldn't shadow simple name of implemented interface.) I hypothesise that the analyses I present could find several bugs that FindBugs misses, precisely because they also use information about names.

Further, FindBugs does not use the idea of seeing all bugs (or latent domain rule violations) as contradictions, as the type-centered analysis does, and as Engler et al.'s technique does. Rather, each FindBugs bug detector determines whether there is a bug or not using ad-hoc decision criteria. In my view, the idea of seeing

[3]Incidentally, I believe augmenting Checker with techniques for automatically inserting annotations based on names, would be an interesting idea to explore.

[4]The FindBugs bug detectors I mention are described at `http://findbugs.sourceforge.net/bugDescriptions.html`.

violations of latent domain rule violations as contradictions is an elegant aspect of the type-centered analysis.[5]

8.4 Naming bugs

My work relates to Høst and Østvold's research into *naming bugs* [28, 29, 30, 31]. While Høst and Østvold's techniques are, in themselves, more sophisticated than the ones I develop, I touch on some interesting ideas that they do not explore. First, the type-centered analysis looks at both method and variable names, whereas Høst and Østvold's analysis only concerns method names. Second, Høst and Østvold always start analysis by assuming either that a method's implementation, as a whole, is correct, or, that its name is correct. The type-centered analysis makes no such assumptions; it considers names as just one among many sign types that carry latent meaning. Hence, it is also capable of finding bugs that manifest as contradictions solely among names used in one method, something that Høst and Østvold's analysis can not do. Finally, unlike Høst and Østvold, I explore the idea that a name can mean different things with different contexts (domains). As one example, the name `key` has different latent meaning inside a method that does encryption, than in a method accessing a key-value store.

[5] I do realise that Engler et al. discovered this idea about 15 years before me.

CHAPTER 9

Discussion and conclusion

I have three goals in this final chapter of the thesis. First, I want to discuss the merits of the two analyses I have presented, and relate the analyses to each other. Second, I want to build up a persuasive argument for the research hypothesis I stated in Section 1.1. Third, I want to present ideas for future conceptual work. To discuss suggestions for how to 'engineer' the analyses for better precision is not a primary goal.

The chapter is structured as follows: In sections 9.1 and 9.2, I discuss the type-centered analysis and the flow-centered analysis, respectively. In these sections, I argue that both analyses have properties that support my research hypothesis. In Section 9.3, I compare the two analyses. Sections 9.2 and 9.3 also include ideas for future work following from the discussion. In Section 9.4, I discuss some ideas to improve the precision of the type-centered analysis. In Section 9.5, I discuss ideas for future work not specific to either analysis. In Section 9.6, I comment on threats to validity. Finally, in Section 9.7, I conclude the thesis.

9.1 Discussion of the type-centered analysis

In this section, I argue that the type-centered analysis satisfies the requirements to an analysis that I stated in Section 1.1. I begin by arguing that the analysis generalises; it can assert many relevant domain rules. After that, I argue that the analysis is simple, by pointing to various aspects of its design. Following this argument, I discuss the importance of the SSA transformation for making the analysis precise, and I suggest ideas for future work. I introduce a notion that I term *flow requirements*, to help me characterise and compare the analyses.

Generalisability

Although the domain rules r_{cs} (Section 3.5.1) and r_{conf} (Section 3.5.2) are rather specific, each can be seen as a member of a larger group of domain rules, where the common denominator within each group is a *flow requirement*. The group to which r_{cs} belongs is characterised by a *must-flow requirement*, and the group to which r_{conf} belongs is characterised by a *must-not-flow requirement*.

For a must-flow requirement, what we want to assert is that certain variables are guaranteed to hold a value stemming from some *approved source(s)*. For example, in the case of r_{cs} we want to assert that any variable that should be a cryptographically secure challenge (where *should be* is implied by latent meaning) is guaranteed to contain a value that stems from a cryptographically secure random source.

For a must-not-flow requirement, what we want to assert is that certain values do not flow into specific locations, such as specific variables or parameter positions. For instance, in the case of r_{conf} we want to assert that confidential values do not leak into output channels that do not preserve confidentiality. Colloquially, a must-not-flow requirement is about enforcing a border around specific values.

Importantly, these flow requirements are general; many specific domain rules can be seen as a must-flow or must-not-flow requirement. For example, consider the following domain rule, concerning cross-site-scripting [23]:

> r_{xss}: All client-provided data in an HTTP request must be *sanitised* before it is output in an HTTP response.

This domain rule, which the type-centered analysis can assert, is essentially a must-not-flow requirement, where the part that must not flow is non-sanitised user input.[1] Recall also from Section 7.2.2 that r_{conf} corresponds to CWE-200: Information Exposure, one of the largest entries in the CWE database, with many specialised weakness types.

Simplicity

Considering that the these requirements concern the *flow* of values, one might, prima facia, assume that enforcing them would require an analysis that i) computes with values and ii) simulates program execution, for example by using abstract interpretation [18]. In other words, one might assume that an analysis with an explicit concept of state and time is required.[2] However, with the type-centered analysis I demonstrate how to approximately enforce these flow requirements with a significantly simpler approach. Firstly, the type-centered analysis fully abstracts away values, and operates only with variables. Secondly, the type-centered analysis is not concerned with order of execution and does no simulation. It can approximately enforce these flow requirements using a simple technique that

[1] Note also in this the case of r_{xss}, that which must not flow is in a sense the 'bad' data, while in the case of r_{conf}, that which must not flow is the 'good' data, and that the type-centered analysis can be used to check both cases.

[2] This is what I assumed when I started work on this project.

does not explicitly model state or time. By virtue of these properties, the analysis supports the research hypothesis I stated in Section 1.1.

Comment

My argument above neglects one detail, which I should comment on. It is true that the core of the type-centered analysis—the fundamental definitions, the latent meaning database, and the typing algorithm—is not concerned with order of execution and uses no simulation. However, the SSA transformation step used in the translation from JVM byte code into Simplified Jimple (Section 3.4.2) brings an element of time into the analysis, and improves precision in certain cases. For instance, with the SSA transformation, the type-centered analysis can distinguish between the snippet

```
x = f();
x = null;
y = x;
```

where the value of `f()` does not flow to `y`, and the snippet

```
x = null;
x = f();
y = x;
```

where the value of `f()` does flow to `y`.[3] In a sense, the SSA transformation makes it so that the core of the analysis *implicitly* considers time and execution order, to some degree.

Future work

The value of the SSA transformation suggests two avenues for future work:

1. Are there other standard transformation techniques that can be applied during the translation process to improve the precision of the type-centered analysis?
2. Are there ad-hoc techniques that can be used *during the translation process* to make the type-centered analysis more precise? In particular, are there interesting ways to 'split' variables as an SSA transformation does, to 'inject' more information into a Simplified Jimple program?

9.2 Discussion of the flow-centered analysis

I continue by arguing that the flow-centered analysis also satisfies the requirements to an analysis that I stated in the research hypothesis.

[3]Precisely, constraint generation for the first snippet works from the Simplified Jimple representation `x1 = f(); x2 = null; y = x2;` while constraint generation for the second snippet works from the Simplified Jimple representation `x1 = null; x2 = f(); y = x2;`.

Generalisability

The noteworthy feature of the flow-centered analysis is that it is inherently compositional. The analysis can be specialised to assert many different kinds of latent domain rule violations by using simple compositions of one building block, the disappearing criticals function, $\mathcal{D}$:

1. To assert that *all* critical values of one domain type are *used in a satisfactory way* within a method (cf. r_{crit}, as described in Chapter 5), we can do one independent evaluation of $\mathcal{D}$ for each variable in the method. I demonstrate this in Section 7.3.1 and Section 7.3.3 of the validation chapter.
2. Several instances of the flow-centered analysis can be run in sequence to assert that certain types of API contract requirements are obeyed. I demonstrate this in Section 7.3.2 of the validation chapter.

Simplicity of the disappearing criticals function

Importantly, the disappearing criticals function $\mathcal{D}$ is also relatively simple. First, $\mathcal{D}$ considers only one variable at a time. Second, in one run of $\mathcal{D}$ the whole 'environment' is a single integer (or the value $\bot$). Finally, $\mathcal{D}$ has no notion of types; the only thing that matters is where critical values of *the one type that is analysed* are output or consumed.[4]

In sum, with the flow-centered analysis I show that a simple, single-variable analysis primitive (the disappearing criticals function $\mathcal{D}$) can be used as a building block to create specialised analyses that can assert many different types of latent domain rule violations. By virtue of being simple and generalisable, the flow-centered analysis further supports the validity of the hypothesis stated in Section 1.1.

9.3 Relating the flow-centered analysis and the type-centered analysis

I now compare the flow-centered analysis with the type-centered analysis, using the notion of flow requirements that I introduced in Section 9.1. Also, I suggest some ideas for future work on the flow-centered analysis. In this section, I ignore minor details in the differences between the analyses.

In Section 9.1, I defined the must-flow requirement as demanding that certain *variables* are guaranteed to hold a value stemming from some approved source(s). However, it does not demand that all *values* output from an approved source must end up in one of these variables. The flow-centered analysis enforces a flow requirement similar to that demand.

[4]Of course, when we compose several runs of $\mathcal{D}$ to do a larger analysis, several domain types are often used. My point is that each run of $\mathcal{D}$ deals with only one type at a time.

Recall from Chapter 5 that I described the flow-centered analysis as identifying *disappearing critical values*. Using the notion of flow requirements, identifying disappearing critical values can be seen as a requirement that any *value* output from one among a set of *sources*—such as the return value of some method—must flow to one among a set of *approved sinks*—such as a return value or an output stream method. An important property of the flow-centered analysis is that it can enforce this flow requirement, while the type-centered analysis can not.

Importantly, though, the flow-centered analysis can not enforce the must-flow requirement, which, using the notion of sinks and sources, would be that any *sink* is guaranteed to consume a value output from some *source*. (The flow-centered analysis can leave some sinks empty.)

Future work

An intriguing question for future work is whether the flow-centered analysis can be adapted to enforce the must-flow requirement, the must-not-flow requirement, and possibly other kinds of flow requirements.

First, there might be a simple solution for enforcing the must-flow requirement: running the analysis backwards.[5] This can be done either by modifying the definition of $\mathcal{D}$, or by 'reversing' programs and then inputting them to $\mathcal{D}$. Incidentally, running $\mathcal{D}$ backwards in addition to forwards should also make the flow-centered analysis better able to enforce API contracts.

How to adapt the flow-centered analysis to enforce the must-not-flow requirement is a puzzling question. I speculate that it can be done using the following technique: First, let all flow parameters that consume data (`in` parameters) correspond to 'bad' locations, where values should not flow. Second, modify $\mathcal{D}$ so it answers: are there values that *may* be consumed, instead of: are there values that *may not* be consumed, as it currently does. Exploring this technique for running the flow-centered analysis in 'negative mode' is another idea for future work.

9.4 Ideas for improving the precision of the type-centered analysis

In this section, I discuss some issues that limit the precision of the type-centered analysis, and discuss ideas for how to address these in future work. Here, I step away from the requirement that the analysis techniques must be simple, and rather ask: if the main objective is to increase precision, how can this be done?

9.4.1 Interprocedural analysis

That the type-centered analysis is intraprocedural makes it simple and computationally efficient, but also reduces its precision. Consider, in context of r_{conf} (Section 3.5.2) a method such as

[5] Here, I can apply theory from standard reference texts, such as Nielson et al's monograph [41].

```
void g(String xs) { secureChannel.write(xs); }
void f() {
  String secret = getSecret();
  g(secret);
}
```

Assume here that `getSecret` returns a high datum, that `secureChannel` takes as input a high datum, and that `g` is not known in $\mathcal{K}_{\mathsf{conf}}$, the latent meaning database for r_{conf}. If we input `f` to the analysis, it will err and output a false positive. When it does not recognise

```
g(secret);
```

as a library call, it outputs the overly pessimistic constraint `secret` = low, which will be in conflict with the constraint `secret` = high generated from the known method call to `getSecret`. In this case, it would be better if the analysis could 'peek inside' `g`, so it could understand that the more precise constraint `secret` = high is possible, and flag the method as bug-free.

For cases like the example above—where there is only one call target for an unknown method call—one could simply *inline* all calls to unknown methods.[6] After inlining, the example method would read as follows:

```
String secret = getSecret();
secureChannel.write(secret);
```

and another run of the analysis (as it is today) would flag the method as bug-free.

However, this inlining technique is probably too naïve to be useful in real-world cases. (For one thing, I am targeting Java, a language where liberal use of polymorphism is standard practice.) A more intriguing alternative is to extend the analysis so that whenever it saw an unrecognised method call it would i) run itself recursively for all possible call targets and ii) merge the information learnt in the recursive runs to generate typing constraints for the variables mentioned in the method call.

Let me illustrate the above idea. Say we run the type-centered analysis on a method and encounter an unrecognised method call

```
x.m(a, b);
```

Assume this is a polymorphic site call that may resolve to two different methods f_1 and f_2. We could then run the type-centered analysis on f_1 and f_2 in turn, and merge the resulting constraints on `a`, `b` from each sub-analysis. For instance, say the domain rule we are analysis is r_{conf} and that analysis of f_1 results in a typing where `a` $\mapsto$ high and `b` $\mapsto$ high, but analysis of f_2 results in a typing where `a` $\mapsto$ low and `b` $\mapsto$ high. We could then output the constraints `a` = low, `b` = high

[6]While remembering to rewrite variable names to avoid collisions.

for the unrecognised method call. I believe a technique like this could greatly improve the precision of the analysis.

9.4.2 Context-dependent latent meaning

Another issue with the type-centered analysis is that it cannot discern library method calls that have different latent meaning depending on context. For example, consider the Java standard library method

```
java.security.Cipher byte[] update(byte[] input)
```

which 'continues a multiple-part encryption or decryption operation (depending on how this cipher was initialized), [and processes] another data part'.[7] If we want to add this method to the latent meaning database for r_{conf}, we are forced to describe the method as doing only encryption, or as doing only decryption.

One idea for addressing this issue is to extend the features of a latent meaning database so that one may specify more than one constraint generation rule for a library method call. For this to be computationally feasible, I believe one must add some intelligence to the constraint generation and typing stages of the analysis; the analysis must understand that 10 `update` calls on the same `Cipher` are either all decryption operations or all encryption operations, instead of considering 2^{10} possibilities.

9.4.3 Supporting more sign types

It is fairly obvious that the precision of the type-centered analysis can be improved by adding support for more sign types, and considering more details of the sign types already supported. The simplest improvement is to add support for distinguishing between types of operators and distinguishing between narrowing and widening casts. If this is done, the type-centered analysis specialised for, say, r_{cs}, can understand that in the snippet

```
int x = secureRandom.nextInt();
long y = (long) -x;
```

the variable `y` can be typed as r_{cs}.

9.5 Ideas for future conceptual work

In this section, I propose ideas for future conceptual work not specific to either analysis. A common aspect of these ideas is that they concern automatically learning domain rules from code examples.

[7] This description is taken from the Java standard library documentation at http://docs.oracle.com/javase/7/docs/.

Corpus-based bug detection

In my view, the most interesting avenue for future work not specific to either analysis is to develop a technique for automatically learning domain rules and domain rules specifications. This means to i) identify domain rules and ii) automatically generate latent meaning databases for the type-centered analysis, or flow-analysis encodings for the flow-centered analysis. With such a technique, one can build a corpus-based bug detection system (Section 2.2) around the analyses. Learning domain rules automatically has several benefits; most importantly, it removes the need for costly manual labour.

It is not clear, however, which domain rules can be learned automatically, and which are so 'hidden' that they are impossible to uncover. For instance, I assume that subtle instantiations of r_{crit} (Section 5.1.2) can be challenging to find. For certain domain rules it may be possible to use a semi-supervised learning technique. For example, one can first let a domain expert specify i) a set of types and method calls with important latent meaning, and ii) one of the flow requirements discussed in Section 9.1 and Section 9.3, and then use this information to guide a learning algorithm.

Other possibilities with learning domain rules automatically

Learning domain rules automatically from a corpus opens other possibilities outside of improved bug detection. Firstly, it has value as a purely zoological project. For instance, one could build a 'domain rule catalogue', comparable to Høst and Østvold's method name phrase book [29]. Secondly, one could create a domain rule-based search engine, which took as input, say, a snippet of code or a type name, and output relevant domain rules paired with use case examples from the corpus.[8] With theory for indexing code based on domain rules, one could also build a domain rule-based code browser. Such a tool would, for example, allow a reviewer to search for all code relevant to the domain rule r_{conf} (Section 3.5.2) in a code repository.

9.6 Threats to validity

A primary goal of this chapter is to structure an argument in support of the hypothesis I posited in Section 1.1. The argument is not invulnerable to criticism, however, and in this section, I address what I regard the most powerful objections.

Volume of persuasion in the experimental validation

In my view, the critical objection to my argument is that the experimental validation in Chapter 7 does not lend sufficient support to the claim that the analyses I present can find interesting latent domain rule violations in real-world code.

I can not decisively rebuke this objection without conducting large-scale empirical validation of the analyses; hence, this is a critical objective for future work.

[8] This would be a corpus-based advise generation system, as discussed in Section 2.2.

Concretely, I plan to do large-scale validation using the Qualitas corpus [60], which is a curated collection of 112 well-known Java projects, such as JBoss and the Eclipse SDK.

Precision of the type-centered analysis on large and complex methods
A large-scale empirical validation will almost certainly show that the analyses must be modified slightly for acceptable precision on complex, real-world code. As long as the validation does not uncover fundamental flaws inherent in the analyses' design, I do not consider this a substantial problem.

One thing I suspect is that the type-centered analysis may produce too many false positives when run on large methods with hundreds or thousands of Simplified Jimple variables. To address this issue, one option is to add heuristics or fuzziness to the type-centered analysis framework. A specific idea is to add a distinction between 'decisive' and 'probable' constraints to the constraint generation and solving process, and augment the typing algorithm with ideas from fuzzy constraint satisfaction [54]. Of course, this raises the issue whether introducing heuristics or fuzziness will add so much complexity that the type-centered analysis can no longer be considered simple—in which case its support of the research hypothesis fails for lack of simplicity. To still support the hypothesis, it must be possible to add heuristics or fuzziness without dramatically increasing complexity.

9.7 Conclusion

In Section 1.1, I posited the hypothesis:

> Simple intraprocedural static analyses techniques are useful for finding interesting and relevant latent domain rule violations.

To support this hypothesis, one must show how to design an analysis that satisfies two criteria: it must be simple and it must find interesting latent domain rule violations of different kinds.

I contribute two static analysis that largely satisfy both criteria.

On the first criteria, I argued for the simplicity of the analyses by comparing them to related work (Chapter 8), and by illustrating simple elements of each analysis' design (Section 9.1 and Section 9.2). On the second criteria, I performed an experimental validation (Chapter 7), and argued that both analyses can approximately enforce various *flow requirements* that generalise across domain rules (Section 9.1 and Section 9.3).

I have also written a proof-of-concept implementation of the analyses, which I contribute as free and open source software.

References

[1] Collins English Dictionary – Complete & Unabridged 10th Edition. Aug 2015.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[3] F. E. Allen. A Technological Review of the FORTRAN I Compiler. In *Proceedings of the June 7-10, 1982, National Computer Conference*, AFIPS '82, pages 805–809, New York, NY, USA, 1982. ACM.

[4] J. B. Almeida et al. *Rigorous software development: an introduction to program verification*. Undergraduate topics in computer science. Springer, London, 2011.

[5] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[6] T. Boland and P. E. Black. Juliet 1.1 C/C++ and Java Test Suite. *Computer*, 45(10):88–90, Oct. 2012.

[7] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[8] R. S. Boyer and J. S. Moore. Program Verification. *J. Autom. Reasoning*, 1(1):17–23, 1985.

[9] G. Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[10] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013.

[11] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543 – 554, 1983.

[12] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[13] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 234–250. Springer Berlin Heidelberg, 1995.

[14] D. Chandler. Semiotics for beginners, 2005.

[15] B. Chelf, D. Engler, and S. Hallem. How to Write System-specific, Static Checkers in Metal. *SIGSOFT Softw. Eng. Notes*, 28(1):51–60, Nov. 2002.

[16] E. M. Clarke. The Birth of Model Checking. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008.

[17] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[18] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[20] E. W. Dijkstra. Formal Techniques and Sizeable Programs. In *Proceedings of the Proceedings of the 1st European Cooperation in Informatics on ECI Conference 1976*, pages 225–235, London, UK, UK, 1976. Springer-Verlag.

[21] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs As Deviant Behavior: A General Approach to Inferring Errors in Systems Code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, Oct. 2001.

[22] R. W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[23] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress Publishing, 2007.

[24] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based Just-in-time

Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM.

[25] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 406–431, London, UK, UK, 1993. Springer-Verlag.

[26] J. Gil and I. Maman. Micro patterns in Java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), October 16-20, 2005, San Diego, CA, USA*, pages 97–116. ACM, 2005.

[27] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[28] E. W. Høst. Understanding programmer language. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA Companion*, pages 943–944. ACM, 2007.

[29] E. W. Høst and B. M. Østvold. The Java programmer's phrase book. In D. Gasevic, R. Lammel, and E. V. Wyk, editors, *SLE*, volume 5452 of *Lecture Notes in Computer Science*, pages 322–341. Springer, 2008.

[30] E. W. Høst and B. M. Østvold. Debugging Method Names. In *Proc. of the 23rd European Conf. on Object-Oriented Programming*, pages 294–317. Springer-Verlag, 2009.

[31] E. W. Høst and B. M. Østvold. Canonical method names for Java: using implementation semantics to identify synonymous verbs. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 226–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[32] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[33] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, Feb. 2012.

[34] N. D. Jones and S. S. Muchnick. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 66–74, New York, NY, USA, 1982. ACM.

[35] Karlsen, Edvard K. and Høst, Einar W. and Østvold, Bjarte M. Finding and Fixing Java Naming Bugs with the Lancelot Eclipse Plugin. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 35–38, New York, NY, USA, 2012. ACM.

[36] J. D. Lafferty, A. McCallum, and F. C. N. Pereira. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 282–289, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[37] L. Lamport. How to write a proof. *The American Mathematical Monthly*, 102(7):600–608, 1995.

[38] L. Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, 2012.

[39] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[40] M. Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

[41] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[42] U. Norell. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08, pages 230–266, Berlin, Heidelberg, 2009. Springer-Verlag.

[43] M. Odersky et al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[44] B. M. Østvold and E. K. Karlsen. Public Review of E-Voting Source Code: Lessons learnt from E-Vote 2011. In *Norsk informatikkonferanse*, 2012.

[45] D. Padua. The Fortran I Compiler. *Computing in Science and Engg.*, 2(1):70–75, Jan. 2000.

[46] M. Paleczny, C. Vick, and C. Click. The Java HotSpot (TM) Server Compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[47] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM.

[48] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.*, 12(5):393–434, July 2002.

[49] T. Prabhu, S. Ramalingam, M. Might, and M. Hall. EigenCFA: Accelerating Flow Analysis with GPUs. *SIGPLAN Not.*, 46(1):511–522, Jan. 2011.

[50] C. Radoi, S. J. Fink, R. Rabbah, and M. Sridharan. Translating Imperative Code to MapReduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 909–927, New York, NY, USA, 2014. ACM.

[51] V. Raychev, M. Vechev, and A. Krause. Predicting Program Properties from "Big Code". *SIGPLAN Not.*, 50(1):111–124, Jan. 2015.

[52] S. P. Reiss. Finding unusual code. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, pages 34–43. IEEE Computer Society, 2007.

[53] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[54] Z. Ruttkay. Fuzzy Constraint Satisfaction. In *In Proc. 3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268, 1994.

[55] A. Sabelfeld and A. C. Myers. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, Sept. 2006.

[56] M. Shaw. Writing Good Software Engineering Research Papers: Minitutorial. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 726–736, Washington, DC, USA, 2003. IEEE Computer Society.

[57] O. G. Shivers. *Control-flow Analysis of Higher-order Languages or Taming Lambda*. PhD thesis, Pittsburgh, PA, USA, 1991. UMI Order No. GAX91-26964.

[58] G. J. Sussman and J. Steele, GuyL. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, 1998.

[59] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[60] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pages 336–345, Dec. 2010.

[61] E. Tempero, H. Yang, and J. Noble. What Programmers Do with Inheritance in Java. In G. Castagna, editor, *ECOOP 2013 – Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 577–601. Springer Berlin Heidelberg, 2013.

[62] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[63] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot – a Java bytecode optimization framework. In *Proceedings of the 1999 conference*

of the Centre for Advanced Studies on Collaborative research, CASCON '99, pages 13–. IBM Press, 1999.

[64] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations, 1998.

[65] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Journal of Automated Software Engineering*, 10(2):203–232, Apr. 2003.