STATIC ANALYSIS OF ANDROID APPLICATIONS

by

Shuying Liang

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2014

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Shuying Liang

has been approved by the following supervisory committee members:

Matthew Might, Chair   April 14, 2014
Date Approved

Ganesh Gopalakrishnan, Member   April 2, 2014
Date Approved

John Regehr, Member   April 2, 2014
Date Approved

Matthew Flatt, Member   April 2, 2014
Date Approved

David Van Horn, Member   April 2, 2014
Date Approved

and by Ross Whitaker, Chair of the
School of Computing

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Today's smartphones house private and confidential data ubiquitously. Mobile apps running on the devices can leak sensitive information by accident or intentionally. To understand application behaviors before running a program, we need to statically analyze it, tracking what data are accessed, where sensitive data flow, and what operations are performed with the data.

However, automated identification of malicious behaviors in Android apps is challenging: First, there is a primary challenge in analyzing object-oriented programs precisely, soundly and efficiently, especially in the presence of exceptions. Second, there is an Android-specific challenge—asynchronous execution of multiple entry points. Third, the maliciousness of any given behavior is application-dependent and subject to human judgment.

In this work, I develop a generic, highly precise static analysis of object-oriented code with multiple entry points, on which I construct an effective malware identification system with a human in the loop. Specifically, I develop a new analysis—pushdown exception-flow analysis, to generalize the analysis of normal control flows and exceptional flows in object-oriented programs. To refine points-to information, I generalize abstract garbage collection to object-oriented programs and enhance it with liveness analysis for even better precision. To tackle Android-specific challenges, I develop multientry point saturation to approximate the effect of arbitrary asynchronous events. To apply the analysis techniques to security, I develop a static taint-flow analysis to track and propagate tainted sensitive data in the pushdown exception-flow framework. To accelerate the speed of static analysis, I develop a compact and efficient encoding scheme, called Gödel hashes, and integrate it into the analysis framework.

All the techniques are realized and evaluated in a system, named *AnaDroid*. *AnaDroid* is designed with a human in the loop to specify analysis configuration, properties of interest and then to make the final judgment and identify where the maliciousness is, based on analysis results. The analysis results include control-flow graphs highlighting suspiciousness, permission and risk-ranking reports. The experiments show that *AnaDroid* can lead to precise and fast identification of common classes of Android malware.

CONTENTS

CHAPTERS

APPENDICES

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I thank my parents for their support over the years and allowing me to realize my potential.

I thank my husband, Weibin. His spiritual support, quiet patience and unwavering love were undeniably the bedrock upon which the past eight years of my life have been built. As a graduate student himself, he always made me and my concerns the priority. Whatever the circumstances are, he has always been around to help get my feet back on the ground. I owe a lot to my three-year-old daughter, Maggie. Thanks for all the laughters and tears she has brought to me.

I gratefully thank my advisor, Dr. Matthew Might. He has graciously and patiently guided me as a scientist. He always encourages and inspires me to aim for excellence. It is he that inspires my major ideas in this work. Thanks for his faith in me, and relentlessly helping me believe in myself too. Without his mentorship, I would not have been able to achieve this Ph.D. efficiently. For all the time and attention he has given me, I thank him.

I also thank members of my doctoral committee, Matthew Flatt, John Regehr, Ganesh Gopalakrishnan and David Van Horn, for their input, valuable discussions and accessibility. I am honored and grateful that I am influenced by my talented committee in theory and practicism. I especially thank John Regehr and Mary Hall for helping my first academic talk in PLAS'12. I also thank David Van Horn for his extra patience in correcting my awkward write ups.

I am lucky that Andy Keep and William Byrd joined in the lab. I thank Andy for helping me correcting and polishing my SPSM'13 paper and some write ups. William Byrd always creates a friendly and creative atomersphere for students to engage in.

I also thank my friend Xing Lin for his time and constructive comments to my oral defense.

CHAPTER 1

INTRODUCTION

Today's smartphones house private and confidential data ubiquitously. The sensitive data include things such as personal email addresses, locations, to social media accounts to bank accounts. Mobile apps running on the devices can leak the sensitive information by accident or intentionally.

Google's Android operating system is the most popular mobile platform, with a 79% share of all smartphones [1]. It allows users to install third-party applications. Due to Android's open application development community, more than one million apps are available in the Android app store [2] with 48 billion cumulative downloads by 2013 [3]. While most of these third-party apps have legitimate reasons to access private data, utilize the Internet, or make changes to local settings and file storage, the permissions provided by Android are too coarse, allowing malware to slip through the cracks. For instance, an app that needs to read information from only a specific website and access global positioning service (GPS) information must, necessarily, be granted full read/write access to the entire Internet, allowing it to maliciously leak location information. In another example, a note-taking application that writes notes to the file system can use the file system permissions to wipe out secure digital card (SD card) files when a malicious trigger gets tripped. Meanwhile, a task manager that legitimately requires every permission available can be benign. Section 1.1 describes four common classes of malware.

To understand application behaviors like these before running the program, we need to statically analyze the application, tracking what data are accessed, where sensitive data flow, and what operations are performed with the data, *i.e.*, determine whether data are tampered with.

However, automated malware detection for Android apps is challenging: First, there is a primary challenge in analyzing object-oriented programs (OOP) precisely, soundly and efficiently, especially in the presence of exceptions. This challenge will be illustrated in

Section 1.2. Second, there is an Android-specific challenge—the multiple entry points. This is illustrated in Section 1.3.

Even then, due to the maliciousness of any given behavior being application-dependent and subject to human judgment, Section 1.4 motivates involving a human analyst in the detection loop.

Section 1.5 presents the thesis statement, and Section 1.6 describes the contributions.

1.1 Common classes of Android malware

Based on related literature and the project experience, I separate Android malware into four categories: data leakage, data tampering, denial of service attacks, and behavior interference.[1]

Data leakage is one of most common, and concerning, malicious behaviors in Android apps [4, 5]. Sensitive data, including location information, an short message service (SMS) message, or a device identifier (ID), is exfiltrated to a third-party host via an hypertext transfer protocol (HTTP) request or Android web component intent, or to a predefined reachable local file via standard file operations. This kind of behavior is often embedded in a background Android service component, such as an AsyncTask or a thread, without interfering with the normal functionality of the app. The other characteristic that malicious software can exploit is that the apps are designed to avoid requesting any permissions. For instance, instead of requesting the ACCESS_FINE_LOCATION an app can instead read locations from photos stored on the file system using Exif data and instead of requesting the INTERNET the app can use the default Android web view through an ACTION_VIEW intent; in both cases avoiding the need to explicitly request these permissions.

Data tampering, similar to data leakage, is to damage the contents of private data. For instance, it corrupts the local file system by overwriting file contents with meaningless data, recursively deleting files from the SD card, or deleting SMS messages. In real-world apps, exceptions are frequently used, especially around input/output (I/O) operations. Without a comparatively sound malware detection model, it is difficult to detect or identify malware.

DoS attack, Denial of service (DoS) on mobile phones exhaust limited resources by intentionally causing the phone to use these resources in an inefficient manner. For instance, an app might drain the battery by setting brightness to maximum or keeping WiFi on at all times or exhaust file system space by logging every operation to a file.

[1] Some apps have more than one malicious behavior.

Behavior interference includes those that do not leak or tamper with sensitive data but still do not behave the way the app was intended to behave. For example, a calculator that uses a random number in a calculation rather than the expected number, or blocks SMS messages in the *onReceive* method when a trigger condition is met.

For the four categories of malware, purely automated static analyses have a hard time in identifying the malware. To start with, analysis of the object-oriented programs are known to be hard to analyze. Second, Android malware are application-dependent and subject to human judgement. The following sections illustrate these points.

1.2 Primary challenge—Mutual dependence of exception-flow, control-flow and points-to analysis

Android apps are written in Java, and it can be difficult to statically produce a precise control-flow graph of the program, particularly in the presence of exceptions. This is because of the mutual dependence of exception-flow, control-flow and points-to analysis.

Exceptions pervade the control-flow structure of modern object-oriented programs. An exception indicates an error occurred during program execution. Exceptions are resolved by dynamically locating code specified by the programmer for handling the exception (an exception handler) and executing this code. This language feature is designed to ensure software robustness and reliability. Ironically, Android malware is exploiting it to leak private sensitive information to the Internet through exception handlers [6]. Analyzing the behavior of programs in the presence of exceptions is important to detect such vulnerabilities. However, exception-flow analysis is challenging, because it depends upon control-flow analysis and points-to analysis, which are themselves mutually dependent, as illustrated in Figure 1.1.

In Figure 1.1, edge A denotes the mutual dependence between exception-flow analysis and traditional control-flow analysis (CFA). CFA traditionally analyzes which methods can be invoked at each call-site. Exception-flow analysis refers to the control-flow that is introduced when throwing exceptions [7]. Intuitively, throwing an exception behaves like a global `goto` statement, in that it introduces additional, complex, interprocedural control-flow into the program. This makes it difficult to reason about feasible run-time paths using traditional CFA. Similarly, infeasible call and return flows can cause spurious paths between throw statements and catch blocks. The complexity of exception-flow analysis makes many object-oriented analysis gives up exceptions or go unsound (see Chapter 14 Section 14.1 for detailed discussion).

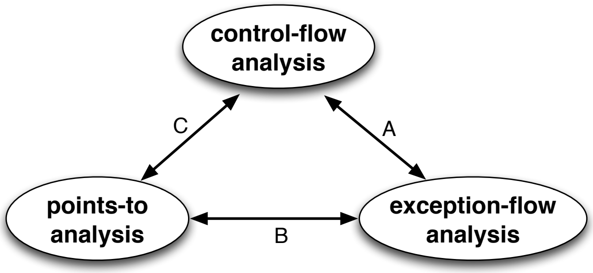

Figure 1.1: Relationship among exception-flow analysis, control-flow analysis and points-to analysis.

The insufficiency of handing exceptions impacts the precision in finding the malware. The following simple example demonstrates this:

```
try {
   maybeThrow();  // Call 1
} catch (Exception e) {  // Handler 1
  startActivity(new Intent(Intent.ACTION_VIEW,Uri.parse(url)));
}
maybeThrow();  // Call 2
```

In the code snippet, the handler code uses Android constructs (to be illustrated in Section 1.3) to leak some sensitive information that is embedded in the `url`. Under a monovariant abstraction like 0CFA [8], where the distinction between different invocations of the same procedure are lost, it will seem as though exceptions thrown from `Call 2` can be caught by `Handler 1`. This further causes the false positive report, which should not be considered as malicious. This imprecision can cause a false negative (undetected maliciousness), too. It can happen when an exception was thrown in `Call 1` but was not caught in `Handler 1` and directed to some other clean code. The imprecise security analysis result can be affected in similar way by the relationships denoted by Edge B and Edge C in Figure 1.1.

Edge B in Figure 1.1 denotes the relationship between exception-flow analysis and points-to analysis. Points-to analysis computes which abstract objects (with respect to allocation sites, calling contexts, *etc.*) a program variable or register can point to. Points-to analysis affects exception-flow analysis, because the type of the exception at a throw site determines which catch block will be executed. That is to say, exception-flow analysis requires precise points-to analysis. Similarly, exceptional flows affect points-to analysis, since the path taken by the exceptional flow can enable or disable object assignments and bindings.

The mutually recursive relationship of CFA and points-to analysis, denoted by edge C, is obvious: abstract objects (points-to analysis) determine which methods can be resolved in dynamic dispatch (CFA), while control-flow paths affect object assignments and bindings for points-to analysis. In fact, exception-flow analysis is an example of this relationship, which exacerbates the edge C relationship further!

1.3 Secondary challenge—Android specific challenge: permutations of asynchronous multientry points

The secondary challenge in analyzing Android apps is caused by the asynchronous multiple entry points into an Android application. Intuitively, to do a sound analysis, all the permutations of the entry points need to be considered. Interleavings of concurrent constructs will not be our concern, because concurrency-incurred control-flows and data-flows are very unreliable when produced or exploited in the context of security vulnerability identification [5]. Therefore, the focus is on the challenge of how to approximate the permutations of the entry points under an Android framework.

The Android framework allows developers to create rich, responsive, and powerful apps by requiring developers to organize their code into components. Each component type serves a different purpose: (1) activities for the main user-interface, (2) services for nonblocking code or remote processes, (3) content providers for managing application data, and (4) broadcast receivers to provide system-wide announcements. Applications can register various component handlers, either explicitly in code or through a resource file (res/layout/*filename*.xml). Whenever an event occurs, the callbacks for the event are invoked asynchronously, potentially interleaving their execution with those in other components. Different apps can also invoke each other by exposing functionality via an intent at both the application and component level.[2] Unlike an application with a single entry point, static analysis for an Android application must explore all permutations of these asynchronous entry points.

Analyzing all permutations can greatly increase the expense of the analysis. As a result, many analyzers use an unsound approximation that can lead to false negatives.

In order to avoid missing malicious behavior, while still performing the analysis efficiently, we need a way to approximate the possible permutations of the asynchronous multientry points without loosing soundness. This means we cannot use heuristic pruning,

[2] Apps sharing the same Linux user ID are also able to access each other's files.

but we also do not want to use a dynamic analysis, since we hope to analyze the program before we attempt to run it.

To summarize the relationship between the challenges, an unsound approximation of asynchronous entry points can miss malicious behavior, while any imprecision in the underlying control-flow, exceptional-flow and data-flow analysis can result in an analysis that falsely reports or misses malicious behavior in programs. This problem is further exacerbated by additional highly dynamic dispatched interprocedural control-flows caused by permutations of entry points. We need to address both challenges to produce precise analysis results, as well as not to miss malicious behavior in the program.

1.4 Human-in-the-loop malware identification

Even if we can tackle the main challenges (Section 1.2 and Section 1.3) of statically analyzing Android apps, purely automated static analysis of malware is insufficient. It is indeed necessary to involve a human in the loop to statically identify malware, given the precise analysis results. The reasons are two-fold:

- Malware is application-dependent and subject to human judgment. For all four categories of common malware classes (Section 1.1), we need a human to determine if this functionality is too far outside the advertised functionality of the app. A human must decide maliciousness in given contexts. The contexts, in this case, are provided by the static analyzer. They can highlight suspicious paths or brief reports, *etc.* For example, analysis results for a photo sharing app may highlight a path 1 from photos to some location A on the Internet. At the same time, there may be another path, 2, highlighted because the analysis captures that the app is sending contact information to the same place, too. Given this information, a human analyst can easily decide path 2 is probably malicious since it is out of the advertised functionality of the app. Even more, provided with more precise data, a human analyst can check where exactly the location A is to determine what the app is doing.
- The second reason we integrate a human in the detection loop is, the analyzer is *not* just for saying malicious or nonmalicious. More importantly, the task is to *identify* malware. In other words, the goal of my analyzer is to provide precise results as much as possible to aid analysts in making a final decision. In addition, to *pinpoint* the maliciousness is the primary requirement in the Automated Program Analysis for Cybersecurity (APAC) Defense Advanced Research Projects Agency (DARPA) project. In other words, the precision of the malware analysis is not judged by "yes"

or "no." The analyzer must identify the locations and behavior of the malicious code, if present.

1.5 Thesis statement

This dissertation is about advancing static analysis techniques to help human analysts detect malware, specifically malicious Android apps. The thesis statement is: *Static analysis for Android apps for malware identification with a human in the loop is feasible and useful.* The idea behind the work relies on two closely related projects: one is a foundational analytic platform with high precision and good-enough performance for object-oriented programs, which can enable a precise and efficient static security analysis built upon it. The other is application security analysis. I mainly target analyzing the most problematic malicious behavior on mobile devices—the leakage or tampering of private sensitive information [4] (the first and second categories of malware in Section 1.1), via static taint-flow analysis. The analysis results are then provided to human analysts to detect and identify malicious behaviors in Android apps. This process also makes the third and fourth categories of malware within my scope, since the two categories rely more on human analysts' judgment of the analysis results.

1.6 Contributions

The contributions of the work are as follows:

- I develop a new abstraction to analyze exception flows and control flows in a generalized fashion: an entry point to a try block is an analogy of a function call; an exit from a try block or due to a throw is an analogy of a function return; handler frames are introduced on the stack in addition to call frames, where both kinds of frames serve to resume the execution contexts. The analysis technique can precisely resolve return flows of thrown exceptions, in addition to normal function calls and returns.
- I develop an abstract garbage collection in an object-oriented setting, and enhance it with liveness analysis to refine the points-to information. This in turn affects control-flows and exception-flows, as illustrated in Figure 1.1.
- I construct the multientry point saturation to model the effect of the asynchronous entry points.
- I develop a static taint-flow analysis in an abstract interpretation framework. When it is built on classical abstract interpretation, the static taint analysis can leverage context-, object- and field-sensitivity; when it is built on pushdown exception-flow analysis, it can track taint-flow information with even better precision.

- I develop and evaluate Gödel hashing to speed up static analysis.
- The evaluation of the analysis techniques are performed on large-scale Java programs and yields precise and efficient analysis results.
- The evaluation of the static malware analyses with a human-in-the-loop demonstrates its utility.

1.7 Outline

The rest of the dissertation is organized as follows: Chapter 2 presents preliminary knowledge in abstract interpretation and introduces some related terms in Android.

Chapter 3 presents the syntax of a low level object-oriented bytecode, which generalizes Dalvik bytecode.

Chapter 4 describes a core concrete semantics—a CESK machine, which is a state-machine in which each state has four components: a (C)ontrol component, an (E)nvironment, a (S)tore and a (K)ontinuation, for the Dalvik bytecode. This is the starting point to derive abstract semantics to form static analysis.

Following abstracting abstract machine (AAM) [9] design methodology, Chapter 5 first refines the concrete semantics that are introduced in Chapter 4 to pointer-refined concrete semantics, which directly enables a classical control-flow analysis with context-, object- and field-sensitivity.

Realizing the limitation of the classical analysis to produce precise return flows for function calls and exception-flows, Chapter 6 refactors the concrete semantics into pushdown semantics to precisely match calls, returns, exception throw, and catch.

To tackle the challenge of analyzing object-oriented programs from the points-to aspect, Chapter 7 describes how to perform abstract garbage collection in object-oriented setting to collect unreachable objects. Since pure abstract garbage collection can not discover "garbage" or "dead" abstract objects in a local scope, this further enhances abstract garbage collection by combining liveness analysis.

Combining the formulations and techniques presented in Chapter 6 and Chapter 7, Chapter 8 formulates the complete reachability algorithm to construct a pushdown exception-flow analyzer.

Since an Android app has multiple entry points rather than a single `main` function, static analysis of Android apps must be able to handle it. Chapter 9 describes the entry point saturation technique to model the permutations of the multiple entry points.

At this point, the foundational analysis framework for Android applications is constructed.

For security analysis, Chapter 10 develops a static taint-flow analysis in abstract interpretation framework. A set of walkthrough examples complement the explanations in addition to the formulation.

To accelerate the analysis, Chapter 11 develops Gödel hashing techniques to transform expensive store operations into numerical operations.

Combining all the techniques described in the previous chapters, Chapter 12 briefly presents some implementation details, discusses how to enable a human in the loop and principled soundness.

Chapter 13 presents the evaluation results in three aspects: (1) In analysis precision and performance aspect, I compare pushdown exception-flow analysis and its enhanced version with classical control-flow analysis. (2) In performance aspect specifically, I evaluate the space and efficiency of Gödel hashing sets and their effectiveness in speeding up analysis. (3) In the security aspect, a series of experiments have been conducted on real Android malware to demonstrate the effectiveness of identifying Android malware by using the static analyzer of Android apps with a human in the loop.

Chapter 14 summarizes some related work and Chapter 15 draws conclusions.

CHAPTER 2

BACKGROUND

2.1 Semantics-based abstract interpretation

Abstract interpretation aims to approximate program behaviors that arise during execution. It allows a designer to specify an analysis and to make the correctness proof mechanical, rather than in an *ad hoc* way. Semantic-based abstract interpretation is a program analysis technique based on concrete semantics, which is used to specify language implementation. Abstract semantics is derived from concrete semantics, reasoning program behavior statically with the same language features that are described by concrete semantics. In the following sections, I will briefly summarize concrete semantics, abstract semantics and its relations.

2.1.1 Concrete semantics

Concrete semantics describes the core of a real language implementation. It can describe run-time execution of a program by modeling the language features. Examples of such machines include CEK and Krivine's machine. Those machines are state transition systems, defined by a transition relation $\Rightarrow$ between program states, $\varsigma \in \Sigma$. For every language feature, such as assignment, function definitions and invocations, *etc.* there is a corresponding definition of $\Rightarrow$ to describe the execution. The semantics starts from an initial state, ς_0, which is *injected* from an injection function,

$$\mathcal{I} : \mathsf{Prog} \to \Sigma, \text{ where } \mathit{prog} \in \mathsf{Prog}.$$

The *meaning* of a program is defined as a set of reachable states obtained by a partial function—the evaluation function:

$$\mathit{eval}(\mathit{prog}) = \{\varsigma \mid \mathcal{I}(\mathit{prog}) \Rightarrow \varsigma\}.$$

The set of reachable states returned by this function can possibly be infinite, which is incomputable.

2.1.1.1 CEK, CESK, CESK*, time-stamped CESK*

The work of "Abstracting Abstract Machine" [9] by Van Horn and Might, introduces a series of concrete machines. Since the semantic techniques embodied in the machines inspire the design and implementation of the analysis of the object-oriented bytecode in this work, it is worth describing those machines to ease the understanding of the rest of the dissertation. To simplify the presentation, I describe the machines for a call-by-value λ-calculus language:

$$\begin{aligned} &e \in \mathsf{Exp} ::= v^{\ell} \mid \texttt{(}e\ e\texttt{)}^{\ell} \mid \texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}^{\ell} \\ &\ell \in \mathsf{Lab}\ \text{ is an infinite set of labels.} \end{aligned}$$

A CEK machine [10] is a kind of concrete machine, where every state of the CEK machine consists of a control string (an expression), an environment that closes the control-string and a continuation. For example, the state space of a CEK machine to model the λ-calculus machine [9] is:

$$\begin{aligned} \varsigma \in \Sigma &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Kont} \\ d \in D &::= \texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)} \\ \rho \in \mathit{Env} &= \mathsf{Var} \rightharpoonup D \times \mathit{Env} \\ \kappa \in \mathit{Kont} &::= \mathbf{mt} \mid \mathbf{ar}(e, \rho, \kappa) \mid \mathbf{fn}(d, \rho, \kappa). \end{aligned}$$

It is an environment-based machine, where the environments are finite maps from variables to closures.

There are three kinds of continuations in the machine. mt denotes no operations to be done. **ar** indicates that the next computation is to evaluate the argument expression (from a function call site), once the lambada expression is evaluated; at the same time, it records the current continuation and environment (caller's context information). **fn** records the current lambda expression that is evaluated and current continuation. When the argument expression is evaluated to a value, the binding from a formal parameter of the lambda expression to the argument value is extended into the environment, and the caller's continuation is resumed. These processes are defined as transition relations of the CESK machine, as shown in Figure 2.1.

The CEK machine has possibly infinite state-space, because both environments and continuations are recursive in structure. When applying structural abstraction on the machine, the map α yields objects in abstract state-space with recursive structure [9], making the abstract state-space infinite. This further makes the analysis complicated. Therefore, to make the abstraction a finite state-space, I need to untie the recursive structures present in the CEK machine. The CESK machine of [11] unties the environment recursiveness. It has an additional store component to allocate variable bindings. The store is a finite map from

$\varsigma \Rightarrow_{CEK} \varsigma'$

$\langle v, \rho, \kappa \rangle$	$\langle d, \rho', \kappa \rangle$ where $\rho(v) = (d, \rho')$
$\langle \texttt{(}e_0\ e_1\texttt{)}, \rho, \kappa \rangle$	$\langle e_0, \rho, \mathbf{ar}(e_1, \rho, \kappa) \rangle$
$\langle d, \rho, \mathbf{ar}(e, \rho', \kappa) \rangle$	$\langle e, \rho', \mathbf{fn}(d, \rho, \kappa) \rangle$
$\langle d, \rho, \mathbf{fn}(\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}, \rho', \kappa) \rangle$	$\langle e, \rho'[v \mapsto (d, \rho)], \kappa \rangle$

Figure 2.1: The CEK machine.

addresses to values and environment is refactored to map variable to addresses. The level of indirection forces the recursive structure to go through explicitly allocated addresses.

Following the same spirit, the CESK* machine store allocates continuations to untie the recursiveness incurred from continuations in the original CEK machine. To illustrate, the abstract state-space of CESK* is shown as follows [9]:

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Addr} \\
s \in \mathit{Storable} &= D \times \mathit{Env} + \mathit{Kont} \\
\kappa \in \mathit{Kont} &::= \mathbf{mt} \mid \mathbf{ar}(e, \rho, a) \mid \mathbf{fn}(d, \rho, a).
\end{aligned}$$

The revised machine is defined in Figure 2.2 in the work of [9].

The CESK machine and the CESK* machine need to operate on addresses, which should be allocated as real computation. In concrete semantics, it is trivial to give an unused value whenever a new addresses is needed. However, in static analysis, we are only allowed to operate on finite resources, which are represented by the finite number of addresses. This means that I need to reuse the previously allocated addresses. In the literature, it is known to use machine history to represent part of an address [12, 13]. The representations that are based on machine history are called *time-stamps*. The work [14] proposes to use history calling contexts as *contours* to parameterize the address allocation strategy. This is the essence of the work k-CFA, which is concerned about what flows to where and *when*.

In this dissertation, I adopt the general time-stamp approach, while extending it to instantiate a context-sensitive, object-sensitive abstract interpretation.

$\varsigma \Rightarrow_{CESK^\star} \varsigma'$, where $\kappa = \sigma(a), b \notin dom(\sigma)$

$\langle v, \rho, \sigma, a \rangle$	$\langle d, \rho', \sigma, a \rangle$ where $(d, \rho') = \sigma(\rho(v))$
$\langle \texttt{(}e_0\ e_1\texttt{)}, \rho, \sigma, a \rangle$	$\langle e_0, \rho, \sigma[b \mapsto \mathbf{ar}(e_1, \rho, a)], b \rangle$
$\langle d, \rho, \sigma, a \rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \sigma[b \mapsto \mathbf{fn}(d, \rho, c)], b \rangle$
if $\kappa = \mathbf{fn}(\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)}, \rho', c)$	$\langle e, \rho'[v \mapsto b], \sigma[b \mapsto (d, \rho)], c \rangle$

Figure 2.2: The CESK* machine.

Here, I demonstrate the technique of constructing a time-stamped CESK* as in the literature [14, 9, 15]. The idea is to add a *Time* component into the state-space definition. For example, for call-by-value λ-calculus language:

$$\begin{aligned} t, u &\in \mathit{Time} \\ \varsigma \in \Sigma &= \mathsf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Addr} \times \mathit{Time}. \end{aligned}$$

The machine is parameterized by the functions:

$$\mathit{tick} : \Sigma \to \mathit{Time} \qquad\qquad \mathit{alloc} : \Sigma \to \mathit{Addr}.$$

The *tick* function returns the next time; the *alloc* function allocates a fresh address for a binding or continuation. We require of *tick* and *alloc* that for all t and ς, $t \sqsubset \mathit{tick}(t)$ and $\mathit{alloc}(\varsigma) \notin \sigma$ where $\varsigma = \langle \_, \_, \sigma, \_, \_ \rangle$.

The advantage of the CESK* machine and time-stamped CESK* is that it allows straightforward design of static analysis, as I will show in the next section.

2.1.2 Abstract semantics

Due to the halting problem, it is not possible to decide membership in the set of reachable states that are produced by concrete semantics. We need to safely approximate the core of the realistic run-time execution and so to reason about program behavior. Abstract semantics is the semantics to approximate the concrete semantics without too much loss of precision and make it computable. It is also a state transition system but operates in a nondeterministic way.

The transition system is defined by the abstract transition relation $\leadsto \ \subseteq \hat{\Sigma} \times \hat{\Sigma}$. The transition starts from an initial abstract state $\hat{\varsigma}_0$, injected from $\hat{\mathcal{I}} : \mathsf{Prog} \rightharpoonup \hat{\Sigma}$. Abstract semantics operate on abstract states $\hat{\Sigma}$, which represent, or *approximate* sets of concrete states. Notationally, the difference between an abstract state $\hat{\varsigma}$ and a concrete state ς is the addition of $\hat{}$ over $\hat{\varsigma}$, but essentially, abstract states are mapped from concrete states by an abstraction map $\alpha : \Sigma \to \hat{\Sigma}$.

The abstract evaluation function is defined as:

$$\widehat{eval} = \{\hat{\varsigma} \mid \hat{\mathcal{I}}(\mathit{prog}) \leadsto \hat{\varsigma}\}.$$

The most straightforward abstraction map a static analysis uses is *structural abstraction*, which lifts abstraction point-wise, element-wise, component-wise and member-wise across the structure of a state.

2.1.2.1 Abstract time-stamped CESK* machine and k-CFA

As stated in Section 2.1.1, CESK* and time-stamped CESK* allow one to design static analysis straightforwardly. Specifically, I can construct abstract machines from CESK* and time-stamped CESK* by structural abstraction.

This is because the two machines have no recursiveness in the structure of the state-space. In fact, during abstraction, the only point of approximation is the store component. This is done by bounding the address space of the store. In addition, to preserve soundness, an address may be mapped to a set of values.

Since the time-stamped CESK* machine can yield context-sensitive analysis, I present its abstract semantics in Figure 2.3.

The abstract semantics of time-stamped CESK* machine resemble in the concrete counterpart, the modifications are considering possibly multiple values for an address and allowing the machine to nondeterministically choose a particular value form the set at a given address.

Correspondingly, the analysis is parameterized by the functions:

$$\widehat{tick} : \hat{\Sigma} \to \widehat{Time}, \qquad\qquad \widehat{alloc} : \hat{\Sigma} \to \widehat{Addr}.$$

The variance of the time-stamp based address allocation strategy, including the definitions of *Time* and $\widehat{Time}$, *tick* and $\widehat{tick}$, *alloc* and $\widehat{alloc}$, are illustrated in Chapter 5, Section 5.2.1.

As stated in Section 2.1.1.1, k-CFA uses history calling-context as *contour* or *time*. In other words, $\widehat{Time}$ is defined as a list of labels, and the $\widehat{tick}$ is defined as follows:

$$\widehat{tick}((e\ e)^{\ell}, \_, \_, \_, \hat{t}) = \mathit{first}_k(\ell : \hat{t}).$$

where first_k returns the first k elements (ℓs in call sites) from a list. $\widehat{Addr}$ is instantiated with $\widehat{Time}$ too:

$$\widehat{Addr} = (\mathsf{Var}, \widehat{Time}).$$

$$\hat{\varsigma} \leadsto \widehat{CESK}^{\star}_{t}\varsigma', \text{ where } \kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(t, \kappa)$$

$\langle v, \rho, \hat{\sigma}, a, t\rangle$	$\langle d, \rho', \hat{\sigma}, a, u\rangle$ where $(d, \rho') \in \hat{\sigma}(\rho(v))$
$\langle (e_0\ e_1), \rho, \hat{\sigma}, a, t\rangle$	$\langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u\rangle$
$\langle d, \rho, \hat{\sigma}, a, t\rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(d, \rho, c)], b, u\rangle$
if $\kappa = \mathbf{fn}((\lambda\ (v)\ e), \rho', c)$	$\langle e, \rho'[v \mapsto b], \hat{\sigma} \sqcup [b \mapsto (d, \rho)], c, u\rangle$

Figure 2.3: The abstract time-stamped CESK* machine.

So, $\widehat{alloc}$ returns an abstract address whenever a new binding is needed in a state $\hat{\varsigma}$.

The definition of $\widehat{Time}$, $\widehat{tick}$ and $\widehat{alloc}$ based on calling history instantiates an k-CFA analysis.

The design methodology of deriving a time-stamped CESK* is applied to analyze object-oriented bytecode in this work (Chapter 5), achieving context-, object- and field-sensitivity (Chapter 5, Section 5.2.1). It is the first attempt of my analysis. The advantages and its limitation motivate my pushdown exception-flow analysis.

2.1.3 Soundness

This section illustrates how to formally establish soundness of the abstract time-stamped CESK* machine. Despite that the language could be different, the proof technique can be used to prove the soundness of my work in this dissertation.

Theorem 2.1 *(Soundness). If $\varsigma \Rightarrow \varsigma'$, and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$, then there exists $\hat{\varsigma}'$ such that $\hat{\varsigma} \leadsto \hat{\varsigma}'$ and $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$.*

The theorem says that the abstract semantics computes an approximate ansIr for each concrete execution that are sequences of concrete states related by $\Rightarrow$. This is illustrated in Figure 2.4. The abstract semantics should not miss any flows of concrete semantics, but it may add extra flows that can never happen. These extra flows are named “infeasible” paths or “unrealizable” paths in the literature [16].

For now, I prove the soundness of a time-stamped CESK* machine. I use an abstraction function, defined in Figure 2.5, from the state-space of the concrete time-stamped machine into the abstracted state-space.

I define the partial order ($\sqsubseteq$) on the abstract state-space as the natural point-wise, element-wise, component-wise and member-wise lifting, wherein the partial orders on the sets Exp and $Addr$ are flat.

Proof. Assume that $\varsigma \Rightarrow \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$. According to the transition relation of $\Rightarrow$, there will be ς' yielded. To prove the theorem, I need to show that an abstract state $\hat{\varsigma}'$ exists

$$\begin{tikzcd}
\varsigma_1 \arrow[r, Rightarrow] & \varsigma_2 \arrow[r, Rightarrow] & \ldots \arrow[r, Rightarrow] & \varsigma_n \\
\alpha(\varsigma_1) \arrow[d, phantom, "\sqsubseteq" rotate=-90] & \alpha(\varsigma_2) \arrow[d, phantom, "\sqsubseteq" rotate=-90] & \ldots & \alpha(\varsigma_n) \arrow[d, phantom, "\sqsubseteq" rotate=-90] \\
\hat{\varsigma}_1 \arrow[r, leadsto] & \hat{\varsigma}_2 & \ldots \arrow[r, leadsto] & \hat{\varsigma}_n
\end{tikzcd}$$

Figure 2.4: Soundness.

$$
\begin{aligned}
\alpha(e,\rho,\sigma,a,t) &= (e,\alpha(\rho),\alpha(\sigma),\alpha(a),\alpha(t)) && \text{[states]}\\
\alpha(\rho) &= \lambda v.\alpha(\rho(v)) && \text{[environments]}\\
\alpha(\sigma) &= \lambda\hat{a}.\bigsqcup_{\alpha(a)=\hat{a}}\{\alpha(\sigma(a))\} && \text{[stores]}\\
\alpha(\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)},\rho) &= (\texttt{(}\lambda\ \texttt{(}v\texttt{)}\ e\texttt{)},\alpha(\rho)) && \text{[closures]}\\
\alpha(\mathbf{mt}) &= \mathbf{mt} && \text{[continuations]}\\
\alpha(\mathbf{ar}(e,\rho,a)) &= \mathbf{ar}(e,\alpha(\rho),\alpha(a))\\
\alpha(\mathbf{fn}(d,\rho,a)) &= \mathbf{fn}(d,\alpha(\rho),\alpha(a)),
\end{aligned}
$$

Figure 2.5: The abstraction map, $\alpha : \Sigma_{CESK^{\star}_{t}} \to \hat{\Sigma}_{\widehat{CESK}^{\star}_{t}}$.

such that the $\leadsto$ *simulates* $\Rightarrow$. This can be proven by the abstraction map with case by case consideration. Finally, I can prove that $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}'$. ■

2.1.4 Static analysis as graph search

As indicated by the abstract transition relation, static analysis as a small-step abstract interpretation is essentially a graph search process, because the sequences of abstract states related by $\leadsto$ and the edges form a directed graph. The analysis terminates when no "new" knowledge is discovered in any state nodes. In other words, the analysis reaches the fixed point when every new state is subsumed ($\sqsubseteq$) by the states that are explored [16, 15].

2.2 Android

Android involves domain-specific terms, which are summarized as follows:

- *Android OS* is an operating system based on the Linux kernel, and designed primarily for touchscreen mobile devices such as smartphones and tablet computers [2].
- *Dalvik virtual machine* (DVM), is the open-source software that runs on Android OS. It houses Android apps running on the devices. The DVM is designed to be isolated from Android kernel services, such as security, memory management, process management, and the network stack.
- An *Android app*, is written in Java programming language. It is compiled into `.dex` (Dalvik executable) files from Java bytcode that are compiled in a Java compiler. Along with any data and resources files, the `.dex` files will be zipped into an *APK*: an Android package. It is an archive file with an `.apk` suffix. Every Android app has its own virtual machine instance. The DVM executes the files in `.dex` format. The core framework components are essential building blocks of an Android app. The

categories of components and the asynchronous multiple entry points that complicate static analysis of Android app, are described in Section 1.2, Chapter 1.

CHAPTER 3

THE SETTING: AN OBJECT-ORIENTED BYTECODE FOR THE DALVIK VM

This chapter defines an object-oriented bytecode language closely modeled on the Dalvik virtual machine to which Java applications for Android are compiled.

3.1 Dalvik bytecode

The machine model and calling convention of Dalvik VM approximately imitates common real architectures and C-style calling conventions. It is register-based and calling frames are fixed in size upon creation. Each frame consists of a particular number of registers (specified by the method) as well as any adjunct data needed to execute the method. The instruction sets are designed based on this: local variables are assigned to any of the 2^{16} available registers in each frame. Dalvik opcodes operate directly on the registers. There are 218 opcodes in Dalvik. Fortunately, they can be abstracted into a set of concise instructions. The generalized instructions are straightforward representations of the original instructions, more importantly, they can be interpreted and analyzed with similar semantics. In addition to the default instruction sets, I also introduce instructions specifically for exceptions to aid analysis. The specific syntax of the abstracted bytecode is described in Section 3.2.

3.2 The syntax of Dalvik bytecode

The syntax of the bytecode language is given in Figure 3.1. Statements encode individual actions for the machine; atomic expressions encode atomically computable values, and complex expressions encode expressions with possible nontermination or side effects. There are four kinds of names: Reg for registers, $\mathsf{ClassName}$ for class names $\mathsf{FieldName}$ for field names and $\mathsf{MethodName}$ for method names. There are two special register names: ret, which holds the return value of the last function called, and exn, which holds the most recently thrown exception.

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{class\text{-}def}\ \ldots \\
\mathit{class\text{-}def} & ::= & (\mathit{attribute}\ \ldots\ \mathsf{class}\ \mathit{class\text{-}name}\ \mathsf{extends}\ \mathit{class\text{-}name} \\
 & & (\mathit{field\text{-}def}\ldots)\ (\mathit{method\text{-}def}\ldots)) \\
\mathit{field\text{-}def} & ::= & (\mathsf{field}\ \mathit{attribute}\ \ldots\ \mathit{field\text{-}name}\ \mathit{type}) \\
\mathit{method\text{-}def} \in \mathsf{MethodDef} & ::= & (\mathsf{method}\ \mathit{attribute}\ \ldots\ \mathit{method\text{-}name}\ (\mathit{type}\ldots)\ \mathit{type} \\
 & & (\mathsf{throws}\ \mathit{class\text{-}name}\ldots)\ (\mathsf{limit}\ n)\ s\ \ldots) \\
s \in \mathsf{Stmt} & ::= & (\mathsf{label}\ \mathit{label}) \mid (\mathsf{nop}) \mid (\mathsf{line\ int}) \mid (\mathsf{goto}\ \mathit{label}) \\
 & \mid & (\mathsf{if}\ \text{\ae}\ (\mathsf{goto}\ \mathit{label})) \mid (\mathsf{assign}\ \mathit{name}\ [\text{\ae} \mid \mathit{ce}]) \mid (\mathsf{return}\ \text{\ae}) \\
 & \mid & (\mathsf{filled\text{-}new\text{-}array}\ \text{\ae}\ldots\text{\ae}\ \mathit{type}) \mid (\mathsf{aput}\ \text{\ae}_v\ \text{\ae}\ \text{\ae}) \\
 & \mid & (\mathsf{aget}\ \mathit{name}\ \text{\ae}\ \text{\ae}) \\
 & \mid & (\mathsf{field\text{-}put}\ \text{\ae}_o\ \mathit{field\text{-}name}\ \text{\ae}_v) \\
 & \mid & (\mathsf{field\text{-}get}\ \mathit{name}\ \text{\ae}_o\ \mathit{field\text{-}name}) \\
 & \mid & (\mathsf{push\text{-}handler}\ \mathit{class\text{-}name}\ \mathit{label}) \mid (\mathsf{pop\text{-}handler}) \\
 & \mid & (\mathsf{throw}\ \text{\ae}) \\
\text{\ae} \in \mathsf{AExp} & ::= & \mathsf{this} \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid \mathsf{void} \mid \mathit{name} \mid \mathsf{int} \\
 & \mid & (\mathit{atomic\text{-}op}\ \text{\ae}\ldots\text{\ae}) \mid (\mathsf{instance\text{-}of}\ \text{\ae}\ \mathit{class\text{-}name}) \\
 & \mid & (\mathsf{check\text{-}cast}\ \text{\ae}\ \mathit{class\text{-}name}) \\
\mathit{ce} & ::= & (\mathsf{new}\ \mathit{class\text{-}name}) \\
 & \mid & (\mathit{invoke\text{-}kind}\ (\text{\ae}\ldots\text{\ae})\ (\mathit{type}_0\ \ldots\ \mathit{type}_n)) \\
 & \mid & (\mathsf{new\text{-}array}\ \text{\ae}\ \mathit{type}) \mid (\mathsf{array\text{-}length}\ \text{\ae}) \\
\mathit{invoke\text{-}kind} & ::= & \mathsf{invoke\text{-}static} \mid \mathsf{invoke\text{-}direct} \mid \mathsf{invoke\text{-}virtual} \\
 & \mid & \mathsf{invoke\text{-}interface} \mid \mathsf{invoke\text{-}super} \\
\mathit{type} & ::= & \mathit{class\text{-}name} \mid \mathsf{int} \mid \mathsf{byte} \mid \mathsf{char} \mid \mathsf{boolean} \\
\mathit{attribute} & ::= & \mathsf{public} \mid \mathsf{private} \mid \mathsf{protected} \mid \mathsf{final} \mid \mathsf{abstract} \\
m \in \mathsf{MethodCall} & & \text{is a set of method invocation sites} \\
\ell \in \mathsf{Label}. & &
\end{array}
$$

Figure 3.1: An object-oriented bytecode adapted from the Android specification.

The syntax is largely typical for a Java-like bytecode, except the bytecode format is longer, because most of the Dalvik instructions contain source and destinations, including move statements move-∗, constant assignments const-∗, comparisons cmp∗, unary operations and binary operations. The series of the instructions are generalized in assign statement. The other generalization is abstracting away the type information of the opcodes. For example, the type specific opcodes such as ∗-byte, ∗-char, *etc.* are ignored in the core syntax. This generalization eliminates the duplicate interpretation rules and simplifies presentation of the analysis as well. The complete supporting Dalvik instructions and their abstracted versions (specified with opcode name) are presented in Appendix A.

In addition to the default instructions, I introduce additional bytecode statements related to exceptions in *method-def*, illustrated in more detail as follows:

- (throws *class-name* . . .) indicates that a method makes a `throws` declaration.
- (push-handler *class-name label*) pushes a handler frame on the stack. The frame will catch exceptions of type *class-name* and divert execution to *label*.
- (pop-handler) pops the top-most handler frame off the stack.

Every statement has a label. With respect to a given program, I assume a syntactic metafunction $\mathcal{S} : \mathsf{Label} \rightarrow \mathsf{Stmt}^*$, which maps a label to the sequence of statements that start with that label.

The subsequent chapters develop and derive semantics for the languages.

CHAPTER 4

CONCRETE SEMANTICS FOR DALVIK BYTECODE

In preparation for synthesizing abstract interpretations, I first construct a small-step machine-based semantics for Dalvik bytecode.

4.1 Concrete state-space

Figure 4.1 presents the machine's concrete configuration-space. The machine has an explicit stack, which under structural abstraction (defined in Chapter 2, Section 2.1.2) will become the stack component of a pushdown system. The stack contains not only call frames, but also miniframes for exception handlers. The *FramePointer* is the environmental component of the machine: by pairing the frame pointer with a register name, it forms the address of its value in the store.

The *Time* component is intentionally left unspecified for now. It will be instantiated when I derive an abstract semantics.

The encoding of objects abstracts over a low-level implementation: objects are a class plus a base pointer, and field addresses are "offsets" from this base pointer. Given an object (op, C), the address of field *field-name* would be $(op, \mathit{field\text{-}name})$. In the semantics, object allocation creates a single new base object pointer op'.

The machines are parameterized by the following functions:

$$\mathit{tick} : \mathsf{Label} \times \mathit{Time} \to \mathit{Time}, \quad \mathit{alloc} : \mathsf{Label} \times \mathit{Time} \to \mathit{Ptr}$$

The *tick* function returns the next time; the *alloc* function allocates a fresh pointer for an address. The definitions of the functions and its abstract counterpart are defined in Chapter 5, Section 5.2.1.

4.2 Concrete transition relation

In this section, I describe the essential cases of the $(\Rightarrow)$ relation, which deal with objects, functions and exceptions. The remaining cases are in Appendix B.1.

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Stmt}^* \times \mathit{FramePointer} \times \mathit{Store} \times \mathit{Kont} \times \mathit{Time} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \to D \\
a \in \mathit{Addr} &= \mathit{RegAddr} + \mathit{FieldAddr} \\
\mathit{ra} \in \mathit{RegAddr} &= \mathit{FramePointer} \times \mathsf{Reg} \\
\mathit{fa} \in \mathit{FieldAddr} &= \mathit{ObjectPointer} \times \mathsf{FieldName} \\
\kappa \in \mathit{Kont} &::= \mathbf{fun}(\mathit{fp}, \vec{s}, \kappa) \\
&\quad\mid \mathbf{handle}(\mathit{class\text{-}name}, \mathit{label}, \kappa) \\
&\quad\mid \mathbf{halt} \\
d \in D &= \mathit{ObjectValue} + \mathit{String} + \mathcal{Z} + \mathcal{B} \\
\mathit{ov} \in \mathit{ObjectValue} &= \mathit{ObjectPointer} \times \mathsf{ClassName} \\
t \in \mathit{Time} &\text{ is a set of time-stamps} \\
\mathit{ptr} \in \mathit{Ptr} &= \mathit{FramePointer} + \mathit{ObjectPointer} \\
\mathit{fp} \in \mathit{FramePointer} &\text{ is an infinite set of frame pointers} \\
\mathit{op} \in \mathit{ObjectPointer} &\text{ is an infinite set of object pointers.}
\end{aligned}$$

Figure 4.1: The concrete configuration-space.

The concrete semantics uses the helper functions described in Figure 4.2. The constructor lookup function $\mathcal{C}$ yields the field names and the constructor associated with a class name. A constructor $\mathcal{K}$ takes newly allocated addresses to use for fields and a vector of arguments; it returns the change to the store plus the record component of the object that results from running the constructor.

The method-lookup function $\mathcal{M}$ takes a method invocation point and an object to determine which method is actually being called at that point.

The concrete semantics are encoded as a small-step transition relation, $(\Rightarrow) \subseteq \Sigma \times \Sigma$.

$$\begin{aligned}
&\mathcal{C} : \mathsf{ClassName} \to (\mathsf{FieldName}^* \times \mathit{Ructor}) \\
&\mathcal{K} \in \mathit{Ructor} = \overbrace{\mathit{Addr}^*}^{\text{fields}} \times \overbrace{D^*}^{\text{arguments}} \to (\overbrace{\mathit{Store}}^{\text{field values}} \times \overbrace{\mathit{FramePointer}}^{\text{record}}) \\
&\mathcal{M} : D \times \mathsf{MethodCall} \rightharpoonup \mathsf{MethodDef}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{A} : \mathsf{AExp} \times \mathit{FramePointer} \times \mathit{Store} \rightharpoonup D \quad \text{evaluates atomic expressions:} \\
&\mathcal{A}(\mathit{name}, \mathit{fp}, \sigma) = \sigma(\mathit{fp}, \mathit{name}) \qquad \text{[variable look-up]} \\
&\mathcal{A}_{\mathcal{F}} : \mathsf{AExp} \times \mathit{FramePointer} \times \mathit{Store} \times \mathsf{FieldName} \rightharpoonup D \quad \text{looks up fields:} \\
&\mathcal{A}_{\mathcal{F}}(\ae, \mathit{fp}, \sigma, \mathit{field\text{-}name}) = \sigma(\mathit{op}, \mathit{field\text{-}name}) \qquad \text{[field look-up]} \\
&\qquad \text{where } (\mathit{op}, \mathit{class\text{-}name}) = \mathcal{A}(\ae, \mathit{fp}, \sigma).
\end{aligned}$$

Figure 4.2: Helper functions for the concrete semantics.

Each statement and expression type has a transition rule below.

The initial configuration consists of the program, the initial frame pointer, an empty heap, and an empty stack: $\varsigma_0 = \mathcal{I}(\vec{s}) = (\vec{s}, fp_0, [], \langle\rangle)$.

4.2.1 New object creation

Creating an object allocates a new object pointer; it also invokes the constructor helper to initialize the object. The (+) operation represents right-biased functional union.

$$
\begin{aligned}
(\llbracket(\mathsf{assign}\ \mathit{name}\ (\mathsf{new}\ \mathit{class\text{-}name}))^{\ell} : \vec{s}\rrbracket, fp, \sigma, \kappa, t) &\Rightarrow (\vec{s}, fp, \sigma'', \kappa, t') \\
\text{where } t' &= \mathit{tick}(\ell, t) \\
d_i &= \{\} \\
(\overrightarrow{\mathit{field\text{-}name}}, \mathcal{K}) &= \mathcal{C}(\mathit{class\text{-}name}) \\
op' &= \mathit{alloc}(\ell, t) \\
a_i &= (op', \mathit{field\text{-}name}_i) \\
(\Delta\sigma, p') &= \mathcal{K}(\vec{a}, \vec{d}) \\
d' &= (op', C) \\
\sigma' &= \sigma + \Delta\sigma + [(op, \mathit{name}) \mapsto d'].
\end{aligned}
$$

4.2.2 Instance field reference/update

Referencing a field gets the object pointer and then grabs the field value as an offset:

$$
\begin{gathered}
(\llbracket(\mathsf{field\text{-}get}\ \mathit{name}\ \text{æ}_o\ \mathit{field\text{-}name}) : \vec{s}\rrbracket, fp, \sigma, \kappa, t) \Rightarrow (\vec{s}, fp, \sigma', \kappa, t') \\
\text{where } t' = \mathit{tick}(\ell, t),\ \ \sigma' = \sigma[(fp, \mathit{name}) \mapsto \mathcal{A}_{\mathcal{F}}(\text{æ}_o, fp, \sigma, \mathit{field\text{-}name})].
\end{gathered}
$$

Updating a field grabs the object, extracts the object pointer and updates the associated field in the store:

$$
\begin{gathered}
(\llbracket(\mathsf{field\text{-}put}\ \text{æ}_o\ \mathit{field\text{-}name}\ \text{æ}_v) : \vec{s}\rrbracket, fp, \sigma, \kappa, t) \Rightarrow (\vec{s}, fp, \sigma', \kappa, t') \\
\text{where } t' = \mathit{tick}(\ell, t), (op, \mathit{class\text{-}name}) = \mathcal{A}(\text{æ}_o, fp, \sigma) \\
\sigma' = \sigma[(op, \mathit{field\text{-}name}) \mapsto \mathcal{A}(\text{æ}_v, fp, \sigma)].
\end{gathered}
$$

4.2.3 Method invocation

Method invocation is a multistep process: it looks up the object, the class of that object and then the appropriate method.[1] When transiting to the body of the resolved method, a new function continuation is instantiated, which records the caller's execution context.

[1] The logic of handling static method resolution resembles that of virtual methods, except that it does not need to climb up class hierarchy chain.

Finally, the store is updated with bindings for the formal parameters and evaluated values of passed arguments.

$$([\![(\textit{invoke-kind}\ (æ_0 \ldots æ_n)(\textit{type}_0 \ldots \textit{type}_n))^{\ell} : \vec{s}]\!], \textit{fp}, \sigma, \kappa, t) \Rightarrow (s'_0, \textit{fp}', \sigma', \kappa', t')$$

$$\begin{aligned}
\text{where } \overbrace{(\mathsf{method}\ \textit{attribute}\ \ldots\ \textit{method-name}\ (\textit{type}\ldots)\ \textit{type}\ldots\ (\mathsf{limit}\ n)\ \vec{s'}\ldots)}^{\textit{method-def}} &= \mathcal{M}(d_0, m) \\
t' &= \textit{tick}(\ell, t) \\
d_0 &= \mathcal{A}(æ_0, \textit{fp}, \sigma) \\
d_i &= \mathcal{A}(æ'_i, \textit{fp}, \sigma), i = m \ldots n-1 \\
\kappa' &= \mathbf{fun}(\textit{fp}, \vec{s}, \kappa) \\
\textit{fp}' &= \textit{alloc}(\ell, t) \\
a'_i &= (\textit{fp}', \textit{name}_j), j = m \ldots n-1 \\
\sigma' &= \sigma[a'_i \mapsto d_i].
\end{aligned}$$

In Dalvik bytecode, the formal parameters in a method are specified as the last m registers. For example, if the method has 2 arguments, and the total registers allocated for the method call frame is 5 specified in statement ($\mathsf{limit}\ n$) (n=5), then the formal parameters are $\textit{name}_3$ and $\textit{name}_4$, and so $m = 3$. Also, the first register $\textit{name}_0$ is always dedicated to the object that the method is being invoked on (for nonstatic method), and other registers are for local variables.

4.2.4 Return to call

Returning from a function checks if the top-most frame pointer is a function continuation (as apposed to an exception-handler continuation). If it is, then the machine binds the result and restores the context of the continuation; if not, then the machine skips to the next continuation. So, if $\kappa = \mathbf{fun}(\textit{fp}', \vec{s'}, \kappa')$:

$$([\![(\mathsf{return}\ æ)^{\ell} : \vec{s}]\!], \textit{fp}, \sigma, \kappa, t) \Rightarrow (\vec{s'}, \textit{fp}', \sigma', \kappa', t')$$

$$\text{where } t' = \textit{tick}(\ell, t)\sigma' = \sigma[(\textit{fp}', \mathsf{ret}) \mapsto \mathcal{A}(æ, \textit{fp}, \sigma)].$$

When returning a value restores the caller's context, an additional step is to put the return value in the dedicated return register, ret.

4.2.5 Return over handler

If attempting to return, but the topmost continuation is a handler, then the machine pops off the handler. So, if $\kappa = \mathbf{handle}(\textit{class-name},\ \textit{label}, \kappa')$:

$$([\![(\mathsf{return}\ æ)^{\ell} : \vec{s}]\!], \textit{fp}, \sigma, \kappa, t) \Rightarrow ([\![(\mathsf{return}\ æ) : \vec{s}]\!], \textit{fp}, \sigma, \kappa', t')$$

$$\text{where } t' = \textit{tick}(\ell, t).$$

4.2.6 Pushing and popping exception handlers

Pushing and popping exception handlers is straightforward:

$$(\llbracket(\textsf{push-handler}\ \textit{class-name}\ \textit{label})^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, \kappa, t)$$
$$\Rightarrow (\vec{s}, \mathit{fp}, \sigma, \mathbf{handle}(\textit{class-name}\ \textit{label}) : \kappa, t')$$
$$\text{where } t' = \mathit{tick}(\ell, t).$$
$$(\llbracket(\textsf{pop-handler}) : \vec{s}\rrbracket, \mathit{fp}, \sigma, \mathbf{handle}(\textit{class-name}\ \textit{label}, \kappa)) \Rightarrow (\vec{s}, \mathit{fp}, \sigma, \kappa)$$
$$\text{where } t' = \mathit{tick}(\ell, t).$$

4.2.7 Throw to matching handler

When the machine encounters a throw statement, it must check if the topmost continuation is both a handler and also a matching handler; if a matching handler is found, that is, *class-name* is a subclass of *class-name*$'$, where $(\mathit{op}, \textit{class-name}) = \mathcal{A}(æ, \mathit{fp}, \sigma)$ and *class-name*$'$ is from the top handler's frame (*HandlerFrame*), the execution flow jumps to code block of the handler:

$$(\llbracket(\textsf{throw}\ æ)^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, \mathbf{handle}(\textit{class-name}',\ \ \textit{label}, \kappa'), t)$$
$$\Rightarrow (\mathcal{S}(\textit{label}), \mathit{fp}, \sigma[(\mathit{fp}, \textsf{exn}) \mapsto (\mathit{op}, \textit{class-name})], \kappa', t').$$

The last thrown exception object value will be put in the dedicated exception register exn.

4.2.8 Throw past nonmatching handler

When throwing, if the topmost handler is not a match, machine continues deeper into the stack, looking far a matching handler.

If *class-name* is *not* a subclass of *class-name*$'$, where $(\mathit{op}, \textit{class-name}) = \mathcal{A}(æ, \mathit{fp}, \sigma)$ and *class-name*$'$ is from the top *HandlerFrame*, then *handle* transits to a configuration with the control state unchanged but with the top frame popped:

$$(\llbracket(\textsf{throw}\ æ)^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, \mathbf{handle}(\textit{class-name}', \textit{label}), \kappa'), t) \Rightarrow (\llbracket(\textsf{throw}\ æ)^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, \kappa', t').$$

4.2.9 Throw past return point

If throwing an exception and the topmost handler is a function return point, then it jumps over this continuation:

$$(\llbracket(\textsf{throw}\ æ) : \vec{s}\rrbracket, \mathit{fp}, \sigma, \mathbf{fun}(\mathit{fp}', \vec{s}', \kappa'), t) \Rightarrow (\llbracket(\textsf{throw}\ æ) : \vec{s}\rrbracket, \mathit{fp}, \sigma, \kappa', t').$$

The throw past nonmatching handler, throw past return and the return over handler, are the "multipop" transition relations that will require modification of the algorithm used for control-state reachability, as illustrated in detail in Chapter 8.

4.2.10 Other transition rules

The above transition relations are the core relations that demonstrate the advantage of upcoming analysis. Other rules for goto, if atomic operations, and array related instructions, *etc.*are described in Appendix B.1.

CHAPTER 5

CLASSICAL K-CFA FOR DALVIK BYTECODE

5.1 Pointer-refined machine semantics for Dalvik bytecode

In order to construct a classical static analysis framework according to the "abstracting abstract machines" (AAM) methodology [9], I need to systematically eliminate recursion from the state-space of the machine. In this case, the only source that can cause infinite state space comes from the continuations and the heap. To eliminate it, I store-allocate frames.

Figure 5.1 contains the concrete state-space for the pointer-refined small-step Dalvik machine. The state-space closely resembles previous concrete state-space. One key difference is the need to explicitly allocate continuations (from the set *Kont*) at a semantic level.

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Stmt}^* \times \mathit{FramePointer} \times \mathit{Store} \times \mathit{KontAddr} \times \mathit{Time} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup D \\
a \in \mathit{Addr} &= \mathit{RegAddr} + \mathit{FieldAddr} + \mathit{KontAddr} \\
ra \in \mathit{RegAddr} &= \mathit{FramePointer} \times \mathsf{Reg} \\
fa \in \mathit{FieldAddr} &= \mathit{ObjectPointer} \times \mathsf{FieldName} \\
\kappa \in \mathit{Kont} &::= \mathbf{fun}(\mathit{fp}, \vec{s}, a_k) \\
&\quad\mid \ \mathbf{handle}(\mathit{class\text{-}name}, \mathit{label}, a_k) \\
&\quad\mid \ \mathbf{halt} \\
d \in D &= \mathit{ObjectValue} + \mathit{String} + \mathcal{Z} + \mathcal{B} + \mathit{Kont} \\
ov \in \mathit{ObjectValue} &= \mathit{ObjectPointer} \times \mathsf{ClassName} \\
a_k \in \mathit{KontAddr} &\text{ is a set of continuation addresses} \\
t \in \mathit{Time} &\text{ is a set of time-stamps} \\
\mathit{ptr} \in \mathit{Ptr} &= \mathit{FramePointer} + \mathit{ObjectPointer} \\
\mathit{fp} \in \mathit{FramePointer} &\text{ is an infinite set of frame pointers} \\
\mathit{op} \in \mathit{ObjectPointer} &\text{ is an infinite set of object pointers.}
\end{aligned}
$$

Figure 5.1: Pointer-refined state-space for Dalvik bytecode.

To do this, I introduce another function to get continuation address. It is a parameterizable function like *tick* and *alloc*: $alloc_{\kappa} : \mathsf{MethodDef} \times \mathit{Time} \to \mathit{Addr}$. The parameterization function *tick*, *alloc* remains the same as the original signature (Chapter 4, Section 4.1).

This machine relies on in the same helper functions in Figure 4.2. The core small-step transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$ now is refactored in Figure 5.2 and Figure 5.3, where Figure 5.2 is for objects and normal function calls and returns, and Figure 5.3 is for exception handling. Other transition rules are shown in Appendix B.2. They are the abstract counterpart of the ones that are defined in Appendix B.1.

The subtle difference is that the continuation is allocated and looked up via the store component. For example, in method invocation rule in Figure 5.2, the new function continuation is created and it is updated in the store, mapped by a continuation address;

[object allocation]:

$$(\llbracket(\mathsf{assign}\ \mathit{name}\ (\mathsf{new}\ \mathit{class\text{-}name}))^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t) \Rightarrow (\vec{s}, \mathit{fp}, \sigma'', a_k, t')$$
$$\text{where } d_i = \{\}, t' = \mathit{tick}(\ell, t), (\overrightarrow{\mathit{field\text{-}name}}, \mathcal{K}) = \mathcal{C}(\mathit{class\text{-}name}),$$
$$\mathit{op}' = \mathit{alloc}(\ell, t), a_i = (\mathit{op}', \mathit{field\text{-}name}_i), (\Delta\sigma, p') = \mathcal{K}(\vec{a}, \vec{d}),$$
$$d' = (\mathit{op}', C), \sigma' = \sigma + \Delta\sigma + [(\mathit{op}, \mathit{name}) \mapsto d'].$$

[field reference]:

$$(\llbracket(\mathsf{field\text{-}get}\ \mathit{name}\ \text{æ}_o\ \mathit{field\text{-}name})^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t) \Rightarrow (\vec{s}, \mathit{fp}, \sigma', a_k, t')$$
$$\text{where } t' = \mathit{tick}(\ell, t), \sigma' = \sigma[(\mathit{fp}, \mathit{name}) \mapsto \mathcal{A}_{\mathcal{F}}(\text{æ}_o, \mathit{fp}, \sigma, \mathit{field\text{-}name})].$$

[field update]:

$$(\llbracket(\mathsf{field\text{-}put}\ \text{æ}_o\ \mathit{field\text{-}name}\ \text{æ}_v)^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k) \Rightarrow (\vec{s}, \mathit{fp}, \sigma', a_k, t')$$
$$\text{where } t' = \mathit{tick}(\ell, t), \sigma' = \sigma[(\mathit{op}, \mathit{field\text{-}name}) \mapsto \mathcal{A}(\text{æ}_v, \mathit{fp}, \sigma)], (\mathit{op}, \mathit{class\text{-}name}) = \mathcal{A}(\text{æ}_o, \mathit{fp}, \sigma).$$

[method invocation]:

$$\overbrace{(\llbracket(\mathit{invoke\text{-}kind}\ (\text{æ}_0 \ldots \text{æ}_n)(\mathit{type}_0 \ldots \mathit{type}_n))^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t)}^{c} \Rightarrow (s'_0, \mathit{fp}', \sigma', {a_k}', t')$$
$$\text{where } \overbrace{(\mathsf{method}\ \mathit{attribute}\ \ldots\ \mathit{method\text{-}name}\ (\mathit{type}\ldots)\ \mathit{type}\ldots\ (\mathsf{limit}\ n)\ s'\ \ldots)}^{\mathit{method\text{-}def}} = \mathcal{M}(d_0, m),$$
$$t' = \mathit{tick}(\ell, t), \quad d_0 = \mathcal{A}(\text{æ}_0, \mathit{fp}, \sigma), \quad d_i = \mathcal{A}(\text{æ}'_i, \mathit{fp}, \sigma)(i = m \ldots n-1),$$
$${a_k}' = \mathit{alloc}_{\kappa}(\mathit{method\text{-}def}, t), \sigma' = \sigma[{a_k}' \mapsto \mathbf{fun}(\mathit{fp}, \vec{s}, a_k)], \mathit{fp}' = \mathit{alloc}(\ell, t),$$
$$a'_i = (\mathit{fp}', \mathit{name}_j)(j = m \ldots n-1), \sigma'' = \sigma[a'_i \mapsto d_i].$$

[return to call]:

$$\text{If } \sigma(a_k) = \mathbf{fun}(\mathit{fp}', \vec{s}', {a_k}') : (\llbracket(\mathsf{return}\ \text{æ})^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t) \Rightarrow (\vec{s}', \mathit{fp}', \sigma', {a_k}', t')$$
$$\text{where } t' = \mathit{tick}(\ell, t), \sigma' = \sigma[(\mathit{fp}', \mathsf{ret}) \mapsto \mathcal{A}(\text{æ}, \mathit{fp}, \sigma)].$$

Figure 5.2: Pointer-refined concrete transition relations (objects and function call/return).

[return over handler]:
If $\sigma(a_k) = \textbf{handle}(\textit{class-name}, \textit{label}, {a_k}')$:

$$(\llbracket(\mathsf{return}\ \textit{æ})^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t) \Rightarrow (\llbracket(\mathsf{return}\ \textit{æ}) : \vec{s}\rrbracket, \mathit{fp}, \sigma, {a_k}', t').$$

[push handler]:

$$(\llbracket(\mathsf{push\text{-}handler}\ \textit{class-name}\ \textit{label})^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t) \Rightarrow (\vec{s}, \mathit{fp}, \sigma', {a_k}', t')$$
$$\text{where } t' = \mathit{tick}(\ell, t), \hat{a}'_k = \mathit{alloc}_{\kappa}(c), \sigma' = \sigma[\hat{a}'_k \mapsto \textbf{handle}(\textit{class-name},\ \textit{label}, a_k)].$$

[pop handler]:
If $\sigma(a_k) = \textbf{handle}(\textit{class-name}, \textit{label}, {a_k}')$

$$(\llbracket(\mathsf{pop\text{-}handler})^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t) \Rightarrow (\vec{s}, \mathit{fp}, \sigma, {a_k}', t')$$
$$\text{where } t' = \mathit{tick}(\ell, t).$$

[throw to matching handler]:
If $\sigma(a_k) = \textbf{handle}(\textit{class-name}', \textit{label}, {a_k}')$ and *class-name* is sub-class of *class-name*$'$:

$$(\llbracket(\mathsf{throw}\ \textit{æ})^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t) \Rightarrow (\mathcal{S}(\textit{label}), \mathit{fp}, \sigma[(\mathit{fp}, \mathsf{exn}) \mapsto (\mathit{op}, \textit{class-name})], {a_k}', t')$$
$$\text{where } t' = \mathit{tick}(\ell, t).$$

[throw to matching handler]:
If $\sigma(a_k) = \textbf{handle}(\textit{class-name}',\ \textit{label}, {a_k}')$ and *class-name* is *not* sub-class of *class-name*$'$:

$$(\llbracket(\mathsf{throw}\ \textit{æ})^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t) \Rightarrow (\llbracket(\mathsf{throw}\ \textit{æ})^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, {a_k}', t')$$
$$\text{where } t' = \mathit{tick}(t).$$

[throw past return point]:
If $\sigma(a_k) = \textbf{fun}(\mathit{fp}', \vec{s}', {a_k}')$

$$(\llbracket(\mathsf{throw}\ \textit{æ}) : \vec{s}\rrbracket, \mathit{fp}, \sigma, a_k, t) \Rightarrow (\llbracket(\mathsf{throw}\ \textit{æ}) : \vec{s}\rrbracket, \mathit{fp}, \sigma, {a_k}', t')$$
$$\text{where } t' = \mathit{tick}(t).$$

Figure 5.3: Pointer-refined concrete transition relations (exceptions).

when a function returns, in Figure 5.3, it first looks up the continuation by the a_k and then decides whether return to calls (or return over handlers). Other mechanics are much like the original concrete semantics described in Section 4.2, I omit the explanations for the others here.

5.2 Classical analysis

Having untied the recursion in continuations by allocating them in the store, I can directly derive the abstract interpretation for Dalvik machine by structural abstraction on the pointer-refined concrete semantics. Essentially, I derive a parameterized analysis framework to conduct traditional, finite-state static analysis.

Figure 5.4 contains the abstract state-space for Dalvik machine. It closely mirrors the concrete semantics. I assume the natural partial order for the components of the abstract

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Stmt}^* \times \widehat{\mathit{FramePointer}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{KontAddr}} \times \widehat{\mathit{Time}} \\
\hat{\sigma} \in \widehat{\mathit{Store}} &= \widehat{\mathit{Addr}} \to \widehat{D} \\
\hat{a} \in \widehat{\mathit{Addr}} &= \widehat{\mathit{RegAddr}} + \widehat{\mathit{FieldAddr}} + \widehat{\mathit{KontAddr}} \\
\widehat{ra} \in \mathit{RegAddr} &= \widehat{\mathit{FramePointer}} \times \mathsf{Reg} \\
\widehat{fa} \in \mathit{FieldAddr} &= \widehat{\mathit{ObjectPointer}} \times \mathsf{FieldName} \\
\hat{\kappa} \in \widehat{\mathit{Kont}} &::= \mathbf{fun}(\hat{fp}, \vec{s}, \hat{a}_k) \\
&\quad\mid\ \mathbf{handle}(\mathit{class\text{-}name}, \mathit{label}, \hat{a}_k) \\
&\quad\mid\ \mathbf{halt} \\
\hat{d} \in \hat{D} &= \mathcal{P}\left(\widehat{\mathit{ObjectValue}} + \widehat{\mathit{String}} + \hat{\mathcal{Z}} + \hat{\mathcal{B}} + \widehat{\mathit{Kont}}\right) \\
\hat{ov} \in \widehat{\mathit{ObjectValue}} &= \widehat{\mathit{ObjectPointer}} \times \mathsf{ClassName} \\
\hat{a}_k \in \widehat{\mathit{KontAddr}} &\text{ is a set of continuation addresses} \\
\hat{t} \in \widehat{\mathit{Time}} &\text{ is a set of time-stamps} \\
\widehat{ptr} \in \widehat{\mathit{Ptr}} &= \widehat{\mathit{FramePointer}} + \widehat{\mathit{ObjectPointer}} \\
\hat{fp} \in \widehat{\mathit{FramePointer}} &\text{ is an infinite set of frame pointers} \\
\widehat{op} \in \widehat{\mathit{ObjectPointer}} &\text{ is an infinite set of object pointers.}
\end{aligned}
$$

Figure 5.4: The pointer-refined abstract configuration-space.

state-space.

The parameterization functions are lifted to the abstract form correspondingly:

$$
\begin{aligned}
\widehat{\mathit{tick}} : \mathsf{Label} \times \widehat{\mathit{Time}} \to \widehat{\mathit{Time}}, \quad \widehat{\mathit{alloc}} &: \mathsf{Label} \times \widehat{\mathit{Time}} \to \widehat{\mathit{Ptr}} \\
\widehat{\mathit{alloc}_\kappa} &: \mathsf{MethodDef} \times \widehat{\mathit{Time}} \to \widehat{\mathit{Addr}}
\end{aligned}
$$

The abstract semantics rely on evaluators for atomic expressions as well I fields reference/update:

$$
\begin{aligned}
\hat{\mathcal{A}} : \mathsf{AExp} \times \widehat{\mathit{FramePointer}} \times \widehat{\mathit{Store}} \rightharpoonup \widehat{D} &\quad \text{evaluates atomic expressions:} \\
\hat{\mathcal{A}}(\mathit{name}, \hat{fp}, \hat{\sigma}) = \hat{\sigma}(\hat{fp}, \mathit{name}) &\qquad \text{[variable look-up]} \\
\hat{\mathcal{A}_{\mathcal{F}}} : \mathsf{AExp} \times \widehat{\mathit{FramePointer}} \times \widehat{\mathit{Store}} \times \mathsf{FieldName} \rightharpoonup \widehat{D} &\quad \text{looks up fields:} \\
\hat{\mathcal{A}_{\mathcal{F}}}(æ, \hat{fp}, \hat{\sigma}, \mathit{field\text{-}name}) = \hat{\sigma}(\widehat{op}, \mathit{field\text{-}name}) &\qquad \text{[field look-up]} \\
\text{where } (\widehat{op}, \mathit{class\text{-}name}) \in \hat{\mathcal{A}}(æ, \hat{fp}, \hat{\sigma}). &
\end{aligned}
$$

Helper functions used by abstract semantics are also lifted to their corresponding abstract form, as described in Figure 5.5.

The constructor-lookup function $\hat{\mathcal{C}}$ yields the field names and the abstract constructor associated with a class name. An abstract constructor $\hat{\mathcal{K}}$ takes abstract addresses to use

$$\hat{\mathcal{C}} : \mathsf{ClassName} \to (\mathsf{FieldName}^* \times \widehat{Ructor})$$

$$\hat{\mathcal{K}} \in \widehat{Ructor} = \overbrace{\widehat{Addr}^*}^{\text{fields}} \times \overbrace{\hat{D}^*}^{\text{arguments}} \to (\overbrace{\widehat{Store}}^{\text{field values}} \times \overbrace{\widehat{FramePointer}}^{\text{record}})$$

$$\mathcal{M} : \widehat{D} \times \mathsf{MethodCall} \rightharpoonup \mathsf{MethodDef}$$

Figure 5.5: Helper functions for the abstract semantics.

for fields and a vector of arguments; it returns the "change" to the store plus the record component of the object that results from running the constructor. The abstract method-lookup function $\hat{\mathcal{M}}$ takes a method invocation point and an object to determine which methods could be called at that point.

The abstract semantics are encoded as a small-step transition relation $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$. Given an initial machine state, the transitive closure of this relation constitutes a classical finite-state static analysis. Details of the relation $(\leadsto)$ for each expression and statement are summarized in Figure 5.6 and Figure 5.7.

Thanks to the pointer-refined concrete semantics, this abstraction is guaranteed to be terminated [9].[1] The difference of the abstract interpretation from the concrete semantics is that continuation is *weak* updated (using the join operator $\sqcup$ into the store, and when we look up a continuation by an abstract continuation address, we are at risk of getting multiple abstract entities (continuations).

5.2.1 Classical instantiation: *k*-CFA-like analysis

I have factored out time-stamp allocation and even the structure of time-stamps themselves:

$$\begin{aligned} Time &= \mathsf{Lab}^* & \widehat{Time} &= \mathsf{Lab}^k \\ Addr &= Ptr \times \mathsf{Offset} & \widehat{Addr} &= \widehat{Ptr} \times \mathsf{Offset} \\ Ptr &= \mathsf{Label} \times Time + Time & \widehat{Ptr} &= \mathsf{Label} \times \widehat{Time} + \widehat{Time} \\ \mathsf{Offset} &= \mathsf{Reg} + \mathsf{FieldName} + \mathsf{Method}. \end{aligned}$$

To justify the analysis, a concrete time-stamp is the sequence of labels traversed since the program began execution. Addresses pair either a variable/field name or a method with a pointer (includes object pointer and frame pointer). Method names are allowed, so that

[1] Since abstraction of base types like int is not the concern of this work, base types are constructed as flat lattices with the type name as the top.

[object allocation]:

$$(\llbracket(\textsf{assign}\ \mathit{name}\ (\textsf{new}\ \mathit{class\text{-}name}))^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}'', \hat{a}_k, \hat{t}')$$
$$\text{where } \hat{d}_i = \{\}, \hat{t}' = \widehat{tick}(\ell, \hat{t}), (\overrightarrow{\mathit{field\text{-}name}}, \hat{\mathcal{K}}) = \hat{\mathcal{C}}(\mathit{class\text{-}name})$$
$$\widehat{op}' = \widehat{alloc}(\ell, \hat{t}), \hat{a}_i = (\widehat{op}', \mathit{field\text{-}name}_i), (\Delta\hat{\sigma}, \hat{p}') = \hat{\mathcal{K}}(\vec{\hat{a}}, \vec{\hat{d}})$$
$$\hat{d}' = (\widehat{op}', C), \hat{\sigma}' = \hat{\sigma} \sqcup \Delta\hat{\sigma} \sqcup [(\widehat{op}, \mathit{name}) \mapsto \hat{d}'].$$

[field reference]:

$$(\llbracket(\textsf{field-get}\ \mathit{name}\ æ_o\ \mathit{field\text{-}name})^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{a}_k, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, \mathit{name}) \mapsto \mathcal{A}_{\mathcal{F}}(æ_o, \hat{fp}, \hat{\sigma}, \mathit{field\text{-}name})].$$

[field update]:

$$(\llbracket(\textsf{field-put}\ æ_o\ \mathit{field\text{-}name}\ æ_v)^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{a}_k) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{a}_k, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \hat{\sigma}' = \hat{\sigma} \sqcup [(\widehat{op}, \mathit{field\text{-}name}) \mapsto \hat{\mathcal{A}}(æ_v, \hat{fp}, \hat{\sigma})], (\widehat{op}, \mathit{class\text{-}name}) \in \hat{\mathcal{A}}(æ_o, \hat{fp}, \hat{\sigma}).$$

[method invocation]:

$$(\llbracket(\mathit{invoke\text{-}kind}\ (æ_0 \ldots æ_n)(\mathit{type}_0 \ldots \mathit{type}_n))^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto (s_0', \hat{fp}', \hat{\sigma}'', \hat{a}_k', \hat{t}')$$
$$\text{where } \overbrace{(\textsf{method}\ \mathit{attribute}\ \ldots\ \mathit{method\text{-}name}\ (\mathit{type} \ldots)\ \mathit{type} \ldots\ (\textsf{limit}\ n)\ s'\ \ldots)}^{\mathit{method\text{-}def}} \in \mathcal{M}(\hat{d}_0, m)$$
$$\hat{t}' = \widehat{tick}(\ell, \hat{t}), \quad \hat{d}_0 = \hat{\mathcal{A}}(æ_0, \hat{fp}, \hat{\sigma}), \quad \hat{d}_i = \hat{\mathcal{A}}(æ_i', \hat{fp}, \hat{\sigma})(i = m \ldots n-1),$$
$$\hat{a}_k' = \widehat{alloc}_{\kappa}(\mathit{method\text{-}def}, t), \quad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_k' \mapsto \mathbf{fun}(\hat{fp}, \vec{s}, \hat{a}_k)], \quad \hat{fp}' = \widehat{alloc}(\hat{c}),$$
$$\hat{a}_i' = (\hat{fp}', \mathit{name}_j)(j = m \ldots n-1), \quad \hat{\sigma}'' = \hat{\sigma}' \sqcup [\hat{a}_i' \mapsto \hat{d}_i].$$

[return to call]:

$$\text{If } \hat{\sigma}(\hat{a}_k) \ni \mathbf{fun}(\hat{fp}', \vec{s}', \hat{a}_k') : (\llbracket(\textsf{return}\ æ)^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto (\vec{s}', \hat{fp}', \hat{\sigma}', \hat{a}_k', \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \quad \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}', \textsf{ret}) \mapsto \hat{\mathcal{A}}(æ, \hat{fp}, \hat{\sigma})].$$

Figure 5.6: Abstract transition relations (objects and function call/return).

continuations can have a binding point for each method at each time. (Were method names not allowed, then all procedures would return to the same continuations in "0"CFA.)

The time-stamp function prepends the most recent label, as shown below:

$$\begin{aligned} tick(\ell, t) &= \ell : t & \widehat{tick}(\ell, \hat{t}) &= \mathit{first}_k(\ell : \hat{t}) \\ alloc(\ell, t) &= (\ell, t) & \widehat{alloc}(\ell, \hat{t}) &= (\ell, \hat{t}) \\ alloc_{\kappa}(\mathit{method\text{-}def}, t) &= (\mathit{method\text{-}def}, t) & \widehat{alloc}_{\kappa}(\mathit{method\text{-}def}, \hat{t}) &= (\mathit{method\text{-}def}, \hat{t}). \end{aligned}$$

The variable/field-allocation function pairs the current time with or without current label, while the continuation-allocation function pairs the method being invoked with the current time: Since a label can indicate call-sites as well as an allocation site, I call the formulation

[return over handler]:
If $\hat{\sigma}(\hat{a}_k) \ni \mathbf{handle}(\mathit{class\text{-}name}, \mathit{label}, \hat{a}'_k)$:

$$([\![(\mathsf{return}\ \text{\ae})^{\ell} : \vec{s}]\!], \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto ([\![(\mathsf{return}\ \text{\ae}) : \vec{s}]\!], \hat{fp}, \hat{\sigma}, \hat{a}'_k, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}).$$

[push handler]:

$$([\![(\mathsf{push\text{-}handler}\ \mathit{class\text{-}name}\ \mathit{label})^{\ell} : \vec{s}]\!], \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{a}'_k, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \hat{a}'_k = \widehat{alloc}_{\kappa}(\hat{c}), \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}'_k \mapsto \mathbf{handle}(\mathit{class\text{-}name},\ \mathit{label}, \hat{a}_k)].$$

[pop handler]:
If $\hat{\sigma}(\hat{a}_k) \ni \mathbf{handle}(\mathit{class\text{-}name}, \mathit{label}, \hat{a}'_k)$:

$$([\![(\mathsf{pop\text{-}handler})^{\ell} : \vec{s}]\!], \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}, \hat{a}'_k, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}).$$

[throw to matching handler]:
If $\hat{\sigma}(\hat{a}_k) \ni \mathbf{handle}(\mathit{class\text{-}name}', \mathit{label}, \hat{a}'_k)$ and *class-name* is sub-class of *class-name*$'$:

$$([\![(\mathsf{throw}\ \text{\ae})^{\ell} : \vec{s}]\!], \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto (\mathcal{S}(\mathit{label}), \hat{fp}, \hat{\sigma} \sqcup [(\hat{fp}, \mathsf{exn}) \mapsto (\widehat{op}, \mathit{class\text{-}name})], \hat{a}'_k, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}).$$

[throw to non-matching handler]:
If $\hat{\sigma}(\hat{a}_k) \ni \mathbf{handle}(\mathit{class\text{-}name}',\ \mathit{label}, \hat{a}'_k)$ and *class-name* is *not* sub-class of *class-name*$'$:

$$([\![(\mathsf{throw}\ \text{\ae})^{\ell} : \vec{s}]\!], \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto ([\![(\mathsf{throw}\ \text{\ae})^{\ell} : \vec{s}]\!], \hat{fp}, \hat{\sigma}, \hat{a}'_k, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\hat{t}).$$

[throw past return point]:
If $\hat{\sigma}(\hat{a}_k) \ni \mathbf{fun}(\hat{fp}', \vec{s}', \hat{a}'_k)$:

$$([\![(\mathsf{throw}\ \text{\ae}) : \vec{s}]\!], \hat{fp}, \hat{\sigma}, \hat{a}_k, \hat{t}) \leadsto ([\![(\mathsf{throw}\ \text{\ae}) : \vec{s}]\!], \hat{fp}, \hat{\sigma}, \hat{a}'_k, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\hat{t}).$$

Figure 5.7: Abstract transition relations (exceptions).

k-CFA-like analysis. So, the above formulation describes k-call site sensitivity analysis, k-object-sensitivity, or the mix of the two, depending on how I $\widehat{tick}$ the $\hat{t} \in \widehat{Time}$. In addition, since the field is addressed by pairing the field name and object pointer, which uses $\widehat{Time}$ to distinguish different objects, the analysis is also field-sensitive.

5.3 Limitations of *k*-CFA

While the construction of a classical control-flow analysis is in place, there are noticeable limitations illustrated in the subsequence sections.

5.3.1 Spurious return flows to exception handlers

The first limitation is that, even for the best co-analysis, where boosting context-sensitivity improves the analysis of exceptions, it does not improve as much as it does for points-to analysis. It is too easy for exceptions to cross context boundaries and merge. For the previous simple example presented in Chapter 1, Section 1.2, I could increase to 1-call-site sensitivity. However, context-sensitivity costs more and is easily confused when calls are wrapped, as in:

```
try {
   callsMaybeThrow();  // Call 1
} catch (Exception e) {  // Handler 1
   System.err.println("Got an exception");
}
callsMaybeThrow();  // Call 2
// ...
void callsMaybeThrow() {
  maybeThrow();
}
```

In this case, an exception thrown in maybeThrow can not be distinguished where it is from. What is worse, analysis can not resolve which handlers to go to!

5.3.2 Limitations of *k*-sensitivity

The limitation of k-sensitivity shares some similar characteristics with the one described in Section 5.3.1, especially in the k-context-sensitivity aspect, but the point is that the k-sensitivity, be it k-context-sensitivity or k-object-sensitivity, can always fail when exceeding the level of k (call site, allocation sites, receiver objects, *etc.*) in modern software constructs. The following toy example is listed for presentation purpose:

```
L1: B b1 = new B(10);
L2: B b2 = new B(100);
L3: foo(b1);
L4: foo(b2);
L5: foo(B b) { bar(b); }
L6: void bar(B bv) { bv.toFork(); }
```

In 1-object sensitivity based on allocation site, I can distinguish the two new objects allocated in L1 and L2; in 1-call site history sensitivity, I can distinguish the object values bound to formal parameter of foo. However, the context-sensitivity fails to distinguish the values bound to bv, because it requires an increase to two levels of sensitivity to be able to distinguish the calls for the nested function bar. When the conflation of the objects in bv in bar happens (which can be easily caused by limited k-object-sensitivity, too, if not by the k-context-sensitivity), the method calls on bv spawn additional spurious function call flows (and the return flows).

CHAPTER 6

PUSHDOWN EXCEPTION-FLOW ANALYSIS FOR OBJECT-ORIENTED PROGRAMS

In Chapter 5, I have derived a parameterized finite-state static analysis to analyze Dalvik bytecode. I did so by reformulating the original machine semantics in Chapter 4 with a pointer refinement and then conducting a structural abstraction. While the abstraction guarantees termination, it is over-approximating in analyzing normal control-flow, exception-flow and data-flow.

To remedy the imprecision issue with good-enough performance, I introduce several analysis techniques in the subsequent chapters.

6.1 A pushdown semantics of exceptions

Rather than adopting the technique of abstracting abstract machines (AAM) to yield a finite state-based analysis, which is equivalent to most of the conservative static analyses, I choose to lightly reformulate the concrete semantics and conduct a similar structural abstraction pushdown analysis. This approach abstracts less than AAM does: I leave the stack unbounded in height. Ultimately, I will extend control-state reachability in pushdown systems to handle the new behaviors introduced by exceptions.

6.1.1 Abstract configuration-space

Abstract semantics are defined on an abstract state-space. Figure 6.1 contains the abstract state-space for the pushdown version of the small-step Dalvik bytecode machine. I assume the natural element-wise, point-wise and member-wise lifting of a partial order across this state-space.

What is important is that, I can extract the high-level structure of the pushdown system from the state-space. A configuration in a pushdown system is a control state (from a finite set) paired with a stack (with a finite number of frames). Observe the following formulas:

$$
\begin{aligned}
\hat{c} \in \widehat{Conf} &= \mathsf{Stmt}^* \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{Kont} \times \widehat{Time} && \text{[configurations]}\\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \rightharpoonup \widehat{D} && \text{[stores]}\\
\hat{a} \in \widehat{Addr} &= \widehat{RegAddr} + \widehat{FieldAddr} && \text{[addresses]}\\
\widehat{ra} \in \widehat{RegAddr} &= \widehat{FramePointer} \times \mathsf{Reg} &&\\
\widehat{fa} \in \widehat{FieldAddr} &= \widehat{ObjectPointer} \times \mathsf{FieldName} &&\\
\hat{\kappa} \in \widehat{Kont} &= \widehat{Frame}^* && \text{[continuations]}\\
\hat{\phi} \in \widehat{Frame} &= \widehat{CallFrame} + \widehat{HandlerFrame} && \text{[stack frames]}\\
\hat{\chi} \in \widehat{CallFrame} &::= \mathbf{fun}(\hat{fp}, \vec{s}) &&\\
\hat{\eta} \in \widehat{HandlerFrame} &::= \mathbf{handle}(\mathit{class\text{-}name}, \mathit{label}) &&\\
\hat{d} \in \widehat{D} &= \mathcal{P}\left(\widehat{ObjectValue} + \widehat{String} + \widehat{\mathcal{Z}} + \widehat{B}\right) && \text{[abstract values]}\\
\hat{ov} \in \widehat{ObjectValue} &= \widehat{ObjectPointer} \times \mathsf{ClassName} &&\\
\widehat{ptr} \in \widehat{Ptr} &= \widehat{FramePointer} + \widehat{ObjectPointer} &&\\
\hat{fp} \in \widehat{FramePointer} &\text{ is a finite set of frame pointers} && \text{[frame pointers]}\\
\widehat{op} \in \widehat{ObjectPointer} &\text{ is a finite set of object pointers} && \text{[object pointers]}\\
\hat{t} \in \widehat{Time} &\text{ is a finite set of of time.} &&
\end{aligned}
$$

Figure 6.1: The abstract configuration-space.

$$
\begin{aligned}
\widehat{Conf} &= \mathsf{Stmt}^* \times \widehat{FramePoiner} \times \widehat{Store} \times \widehat{Kont} \times \widehat{Time}\\
&\cong \mathsf{Stmt}^* \times \widehat{FramePoiner} \times \widehat{Store} \times \widehat{Time} \times \widehat{Kont}\\
&= \left(\mathsf{Stmt}^* \times \widehat{FramePoiner} \times \widehat{Store} \times \widehat{Time}\right) \times \widehat{Kont}\\
&= \underbrace{\left(\mathsf{Stmt}^* \times \widehat{FramePoiner} \times \widehat{Store} \times \widehat{Time}\right)}_{\text{control states}} \times \underbrace{\widehat{Frame}^*}_{\text{stack}}
\end{aligned}
$$

To synthesize the abstract state-space, I force frame pointers and object pointers (and thus addresses) to be a finite set, but crucially, I leave the stack untouched. When I compact the set of addresses into a finite set, the machine may run out of addresses to allocate, and when it does, the pigeon-hole principle will force multiple abstract values to reside at the same address. As a result, I have no choice but to force the range of the $\widehat{Store}$ to become a poIr set in the abstract configuration-space.

6.1.2 Abstract transition relation

The abstract transition relation replies on the helper functions in abstract forms. Here, I reuse the ones defined in Figure 5.5.

The inject function is different, as defined: $\mathcal{I} : \mathsf{Stmt}^* \rightarrow \widehat{Conf}$ injects an sequence of instructions into a configuration:

$$\hat{c}_0 = \hat{\mathcal{I}}(\vec{s}) = (\vec{s}, \hat{fp}_0, [], \langle\rangle).$$

The rules for the abstract transition relation $(\leadsto) \subseteq \widehat{Conf} \times \widehat{Conf}$ largely mimic the structure of the concrete relation $(\Rightarrow)$. The biggest difference is that the structural abstraction forces the abstract transition to become nondeterministic.

Figure 6.2 and Figure 6.3 detail the transition relations in pushdown semantics. It is straightforward to construct the abstract rules for object allocation, field reference/update, method invocation and return, as shown in Figure 6.2.

Notice that for method invocation (nonstatic methods) in Figure 6.2, there can be a *set* of possible objects that are invoked, rather than only one as in its concrete counterpart. This also means that there could be multiple method definitions resolved for each object.

It is not immediately clear how to abstract the rules involving exceptions and what the effects are, in the presence of the unbounded stack. My solution is shown in Figure 6.3. The abstract rules resemble their concrete counterparts, but essentially, the abstraction of these multipop transition relations simplify the control-state reachability algorithm substantially, as I show in Chapter 8. Taken together, Figure 6.2 and Figure 6.3 are the principle rules to construct the subroutine next that is called in the extended reachability Alg. 4 in Chapter 8.

Since other pushdown semantics are not essential to demonstrate the advantage of the analysis, they are summarized in Appendix B.2.

[object allocation]:

$$\overbrace{(\llbracket(\mathsf{assign}\ name\ (\mathsf{new}\ class\text{-}name))^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t})}^{\hat{c}} \rightsquigarrow (\vec{s}, \hat{fp}, \hat{\sigma}'', \hat{\kappa}, \hat{t}')$$

$$\text{where } \hat{d}_i = \{\}, \hat{t}' = \widehat{tick}(\ell, \hat{t}), (\overrightarrow{\mathit{field\text{-}name}}, \hat{\mathcal{K}}) = \hat{\mathcal{C}}(class\text{-}name),$$

$$\widehat{op}' = \widehat{alloc}(\ell, \hat{t}), \hat{a}_i = (\widehat{op}', \mathit{field\text{-}name}_i), (\Delta\hat{\sigma}, \hat{p}') = \hat{\mathcal{K}}(\vec{\hat{a}}, \vec{\hat{d}}),$$

$$\hat{d}' = (\widehat{op}', C), \hat{\sigma}' = \hat{\sigma} \sqcup \Delta\hat{\sigma} \sqcup [(\widehat{op}, name) \mapsto \hat{d}'].$$

[field reference]:

$$(\llbracket(\mathsf{field\text{-}get}\ name\ \ae_o\ \mathit{field\text{-}name})^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \rightsquigarrow (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}')$$

$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, name) \mapsto \mathcal{A}_{\mathcal{F}}(\ae_o, \hat{fp}, \hat{\sigma}, \mathit{field\text{-}name})].$$

[field update]:

$$(\llbracket(\mathsf{field\text{-}put}\ \ae_o\ \mathit{field\text{-}name}\ \ae_v)^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}) \rightsquigarrow (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}')$$

$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \hat{\sigma}' = \hat{\sigma} \sqcup [(\widehat{op}, \mathit{field\text{-}name}) \mapsto \hat{\mathcal{A}}(\ae_v, \hat{fp}, \hat{\sigma})], (\widehat{op}, class\text{-}name) = \hat{\mathcal{A}}(\ae_o, \hat{fp}, \hat{\sigma}).$$

[method invocation]:

$$\overbrace{(\llbracket(invoke\text{-}kind\ (\ae_0 \ldots \ae_n)(type_0 \ldots type_n))^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t})}^{\hat{c}} \rightsquigarrow (s_0', \hat{fp}', \hat{\sigma}', \hat{\kappa}', \hat{t}'),$$

$$\text{where } \overbrace{(\mathsf{method}\ attribute\ \ldots\ method\text{-}name\ (type \ldots)\ type \ldots\ (\mathsf{limit}\ n)\ s'\ \ \ldots)}^{method\text{-}def} \in \mathcal{M}(\hat{d}_0, m),$$

$$\hat{t}' = \widehat{tick}(\ell, \hat{t}), \quad \hat{d}_0 = \hat{\mathcal{A}}(\ae_0, \hat{fp}, \hat{\sigma}), \quad \hat{d}_i = \hat{\mathcal{A}}(\ae_i', \hat{fp}, \hat{\sigma})(i = m \ldots n-1),$$

$$\hat{\kappa}' = \mathbf{fun}(v, succ(\ell), \hat{p}) : \hat{\kappa}, \quad \hat{fp}' = \widehat{alloc}(\ell, \hat{t}),$$

$$\hat{a}_i' = (\hat{fp}', name_j)(j = m \ldots n-1), \quad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i' \mapsto \hat{d}_i].$$

[return to call]:

$$\text{If } \hat{\kappa} = (\mathbf{fun}(\hat{fp}', \vec{s}') : \hat{\kappa}'), (\llbracket(\mathsf{return}\ \ae)^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \rightsquigarrow (\vec{s}', \hat{fp}', \hat{\sigma}', \hat{\kappa}', \hat{t}')$$

$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}', \mathsf{ret}) \mapsto \hat{\mathcal{A}}(\ae, \hat{fp}, \hat{\sigma})].$$

Figure 6.2: Pushdown abstract transition relations (objects and function call/return).

[return over handler]:
If $\hat{\kappa} = \mathbf{handle}(\mathit{class\text{-}name}, \mathit{label}) : \hat{\kappa}'$

$$(\llbracket(\mathsf{return}\ \textit{æ})^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\llbracket(\mathsf{return}\ \textit{æ}) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}', \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}).$$

[push handler]:

$$(\llbracket(\mathsf{push\text{-}handler}\ \mathit{class\text{-}name}\ \mathit{label})^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}', \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \hat{\kappa}' = \mathbf{handle}(\mathit{class\text{-}name},\ \mathit{label}) : \hat{\kappa}.$$

[pop handler]:
If $\hat{\kappa} = \mathbf{handle}(\mathit{class\text{-}name}, \mathit{label}) : \hat{\kappa}'$

$$(\llbracket(\mathsf{pop\text{-}handler})^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}', \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}).$$

[throw to matching handler]:
If $\hat{\kappa} = \mathbf{handle}(\mathit{class\text{-}name}', \mathit{label}) : \hat{\kappa}'$ and *class-name* is sub-class of $\mathit{class\text{-}name}'$:

$$(\llbracket(\mathsf{throw}\ \textit{æ})^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\mathcal{S}(\mathit{label}), \hat{fp}, \hat{\sigma} \sqcup [(\hat{fp}, \mathsf{exn}) \mapsto (\widehat{op}, \mathit{class\text{-}name})], \hat{\kappa}', \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}).$$

[throw to non-matching handler]:
If $\hat{\kappa} = \mathbf{handle}(\mathit{class\text{-}name}',\ \mathit{label}) : \hat{\kappa}'$ and *class-name* is *not* sub-class of $\mathit{class\text{-}name}'$:

$$(\llbracket(\mathsf{throw}\ \textit{æ})^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\llbracket(\mathsf{throw}\ \textit{æ})^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}', \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\hat{t}).$$

[throw past return point]:
If $\hat{\kappa} = \mathbf{fun}(\hat{fp}', \vec{s}') : \hat{\kappa}'$

$$(\llbracket(\mathsf{throw}\ \textit{æ}) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\llbracket(\mathsf{throw}\ \textit{æ}) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}', \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\hat{t}).$$

Figure 6.3: Pushdown abstract transition relations (exceptions).

CHAPTER 7

ENHANCED ABSTRACT GARBAGE COLLECTION FOR OBJECT-ORIENTED PROGRAMS

The previous chapter formulates a pushdown system (which termination is ensured in Chapter 8) to handle complicated control-flows (both normal and exceptional). This section describes how I prune the analysis for exceptions from the angle of points-to analysis with enhanced garbage collection generalized for object-oriented programs.

7.1 Abstract garbage collection in an object-oriented setting

The idea of abstract garbage collection was first proposed in [17]. As an analog to the concrete garbage collection, abstract garbage collection reallocates unreachable abstract resources. Order-of-magnitude improvements in precision have been reported, even as it drops run-times by cutting away false positives. It is natural to think that this technique can benefit exception-flow analysis for object-oriented languages. In fact, in an object-oriented setting, abstract garbage collection can free the analysis from the context-sensitivity and object-sensitivity limitation, since the "garbage" discarded is ignorant of any form of sensitivity. For example, in the following simple code snippet:

```
A a1 = idA(new A());
A a2 = idA(new A()):
B b1 = idB(a1.makeB());
B b2 = idB(a2.makeB());
```

`idA` and `idB` are identity functions. Traditionally, with one level of object-sensitivity and one level of context-sensitivity, I am able to distinguish the arguments passed in all of the four lines. However, it is easy to exceed the k-sensitivity (call site, allocation sites, receiver objects, *etc.*) in modern software constructs (Chapter 5, Section 5.3). Abstract garbage collection can play a role in the way that it discards conservative values and enables fresh bindings for reused variables (formal parameters). This does not need knowledge about any sensitivity. Thus, it can avoid "merging" of abstract object values (and so indirectly eliminate potentially spurious function calls). For exceptions specifically, abstract garbage collection can help avoid conflating exception objects at various throw sites.

To gain the promised analysis precision and performance, I must conduct a careful and subtle redesign of the abstract garbage collection machinery for object-oriented languages. Specifically, I need to make it work with the abstract semantics defined in Chapter 6. In addition, the reachability algorithm should also be able to work with abstract garbage collection. Fortunately, the challenge of how to adapt abstract garbage collection into pushdown systems has been resolved in the work of [18]. I will review this work in Section 8.1.

Here, I describe how I adapt abstract garbage collection to analyze object-oriented languages. Abstract garbage collection discards unreachable elements from the store, it modifies the transition relation to conduct a "stop-and-copy" garbage collection before each transition. To do so, I define a garbage collection function $\hat{G} : \widehat{Conf} \to \widehat{Conf}$ on configurations:

$$\hat{G}(\overbrace{\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}) = (\vec{s}, \hat{fp}, \hat{\sigma}|\mathit{Reachable}(\hat{c}), \hat{\kappa}),$$

where the pipe operation $f|S$ yields the function f, but with inputs not in the set S mapped to bottom—the empty set. The reachability function $\mathit{Reachable} : \widehat{Conf} \to \mathcal{P}(\widehat{Addr})$ first computes the root set and then the transitive closure of an address-to-address adjacency relation:

$$\mathit{Reachable}(\overbrace{\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}}^{\hat{c}}) = \left\{ \hat{a} : \hat{a}_0 \in \mathit{Root}(\hat{c}) \text{ and } \hat{a}_0 \overset{*}{\underset{\hat{\sigma}}{\rightarrowtriangle}} \hat{a} \right\},$$

where the function $\mathit{Root} : \widehat{Conf} \to \mathcal{P}(\widehat{Addr})$ finds the root addresses:

$$\mathit{Root}(\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}) = \{(\hat{fp}, v) : (\hat{fp}, v) \in \mathit{dom}(\hat{\sigma})\} \cup \mathit{StackRoot}(\hat{\kappa}),$$

The $\mathit{StackRoot} : \widehat{Kont} \to \mathcal{P}(\widehat{Addr})$ function finds roots on the stack. However, only $\widehat{CallFrame}$ has the component to construct addresses, so I define a helper function $\hat{\mathcal{F}} : \widehat{Kont} \to \widehat{CallFrame}^*$ to extract only $\widehat{CallFrame}$ out from the stack and skip over all the handle frames. Now $\mathit{StackRoot}$ is defined as

$$\mathit{StackRoot}(\hat{\kappa}) = \{(\hat{fp}_i, v) : (\hat{fp}_i, v) \in \mathit{dom}(\hat{\sigma}) \text{ and } \hat{fp}_i \in \hat{\mathcal{F}}(\hat{\kappa})\},$$

and the relation:

$$(\rightarrowtriangle) \subseteq \widehat{Addr} \times \widehat{Store} \times \widehat{Addr},$$

connects adjacent addresses:

$$\hat{a} \underset{\hat{\sigma}}{\rightarrowtriangle} \hat{a}' \text{ iff there exists } (\widehat{op}, \mathit{class\text{-}name}) \in \hat{\sigma}(\hat{a}),$$

such that $\hat{a}' \in \{(\widehat{op}, \mathit{field\text{-}name}) : (\widehat{op}, \mathit{field\text{-}name}) \in \mathit{dom}(\hat{\sigma})\}$.

The formulated abstract garbage collection semantics constructs the subroutine `egc` that is called in Alg. 4, which is the interface to enable abstract garbage collection in the reachability algorithm, similar to description in the work of [18].

7.2 Abstract garbage collection enhanced with liveness analysis

Abstract garbage collection can avoid conflating abstract objects for reused variables or formal parameters, but it can not discover "garbage" or "dead" abstract objects in the local scope. The following example illustrates this:

```
bool foo(A a) {
  B b = B.read(a);
  C p = C.doSomething(b);
  return bar(C.not(p));
}
```

Obviously, in the function body `foo`, `b` is actually "dead" after the second line. However, näive abstract garbage collection has no knowledge of this. In fact, this is a problem for näive concrete garbage collection [19]. In the realm of static analysis, the garbage value pointed to by `p` can pollute the exploration of the entire state space.

In addition, in the register-based byte code that my implementation analyzes, there are obvious cases where the same register is reassigned multiple times at different sites within a method. The direct adaptation of abstract garbage collection to an object-oriented setting in Section 7.1 cannot collect these registers between uses. For object-oriented programs, I want to collect registers that are unreachable, but not without an intervening assignment. This problem can be easily solved by using liveness analysis. Of course, I could also solve it by transforming the byte code into static single assignment (SSA) form. However, as mentioned above, liveness analysis has additional benefits, so I chose to enhance the abstract garbage collection with live variable analysis (LVA).

LVA computes the set of variables that are *alive* at each statement within a method. The garbage collector can then more precisely collect each frame.

Since LVA is well-defined in the literature [20], I skip the formalization here, but the *Root* is now modified to collect only *live* variables of the current statement $\mathit{Lives}(s_0)$:

$$\mathit{Root}(\vec{s}, \hat{\mathit{fp}}, \hat{\sigma}, \hat{\kappa}) = \{(\hat{\mathit{fp}}, v') : (\hat{\mathit{fp}}, v') \in \mathit{dom}(\hat{\sigma}) \text{ and } v' \in \mathit{Lives}(s_0)\} \cup \mathit{StackRoot}(\hat{\kappa}).$$

The liveness property is embedded in the overall `egc` subroutine in Alg. 4.

CHAPTER 8

PUSHDOWN EXCEPTION-FLOW REACHABILITY

Given the formalisms in the previous chapters, it is not immediately clear how to convert these rules into a static analyzer, or more importantly, how to handle the unbounded stack without it always visiting new machine configurations. Thus, we need a way to compute a finite summary of the reachable machine configurations.

In abstract interpretation frameworks, the Dyck state graph synthesis algorithm [21], which is a purely functional version of the saturation algorithm [16], provides a method for computing reachable pushdown control states. I build my algorithms on the work of [18]. As it turns out, it is not hard to extend the summarization idea to deal with an unbounded stack with exceptions. In the following sections, I present the complete algorithm in a top-down fashion, which aims to easily turn into actual working code.

8.1 Analysis setup

The analysis for a program starts from the Analyze function, as shown in Alg. 1. It accepts a program expression (an entry point to a program), and gives out a Dyck state graph (DSG). Formally speaking, a DSG of a pushdown system is the subset of a pushdown system reachable over legal paths. (A path is *legal* if it never tries to pop a when a frame other than a is on top of the stack.) The $\widehat{Time}$ component is designed for accommodating traditional analysis, depending on actual implementation. For example, the last k call sites or object-allocation labels, or the mix of them. The analysis produces DSG from the subroutine Eval, which is the fix-point synthesis algorithm.

In Alg. 1, *IECG* is a composed data structure used in the ϵ summarization algorithm. It is derived from the idea of an ϵ closure graph (ECG) in the work of [21], but supports efficient caching of ϵ closures along with transitive push frames on the stack. Specifically, $IECG = (\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{TF}, \overrightarrow{PSF}, \overleftarrow{PFP}, \overleftarrow{NEP})$. The six components can be considered maps:

- ϵ predecessors $\overleftarrow{G}$: $\hat{\Sigma} \to \{\hat{\Sigma}\}$, maps a target node to the source node(s) of an ϵ edge(s).

Algorithm 1: ANALYZE

```
Input: $s$: a list of program statements (with an initial entry point $s_0$).
Output: Dyck State Graph $DSG$ : a triple of a set of control states
          a set of edges, and a initial state.
$\hat{\sigma}_0 \longleftarrow$ empty store
$\hat{fp}_0 \longleftarrow$ initial empty stack frame pointer
$\hat{t}_0 \longleftarrow$ empty list of contexts
$\hat{q}_0 \longleftarrow (s_0, \hat{fp}_0, \hat{\sigma}_0, \hat{t}_0)$
The initial working set $W_0 \longleftarrow \{\hat{q}_0\}$
$IECG_0 \longleftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
$DSG_0 \longleftarrow (\{\hat{q}_0\}, \emptyset, \hat{q}_0)$
$(DSG, IECG, \hat{\sigma}, W) \longleftarrow$ EVAL$(DSG_0, IECG_0, \hat{\sigma}_0, W_0)$
return $DSG$
```

- ϵ successors $\overrightarrow{G}$: $\hat{\Sigma} \rightarrow \{\hat{\Sigma}\}$, maps a source node to the target node(s) of an ϵ edge(s).
- top frames $\overrightarrow{TF}$: $\hat{\Sigma} \rightarrow \{\widehat{Frame}\}$, records the shallow pushed stack frame(s) for a state node.
- possible stack frames $\overrightarrow{PSF}$: $\hat{\Sigma} \rightarrow \{\widehat{Frame}\}$, compute all possible pushed stack frame of a state. It is used for abstract garbage collection.
- predecessors for push action $\overleftarrow{PFP}$: $(\hat{\Sigma}, \widehat{Frame}) \rightarrow \{\hat{\Sigma}\}$, records source state node(s) for a pushed frame and the net-changed state. For example, in the legal path: $\hat{q}_0 \xrightarrow{g^+} \hat{q}_1 \longrightarrow ... \xrightarrow{g^-} \hat{q}_2$, the entry $(\hat{q}_1, g^+) \longrightarrow \{\hat{q}_0\}$ is in $\overleftarrow{PFP}$.
- non-ϵ predecessors ($\overleftarrow{NEP}$: $\hat{\Sigma} \rightarrow \{\hat{\Sigma}\}$), maps a state node to non-ϵ predecessors.

These data structures (and *IECG*) have the same definition in the following algorithms.

8.2 Fix-point algorithm of the pushdown exception framework

Alg. 2 describes the fix-point computation for the reachability algorithm. It iteratively constructs the reachable portion of the pushdown transition relation (Ln. 5-12) by inserting ϵ-summary edges whenever it finds an empty-stack (Ln. 13-20) (*e.g.*, push a, push b, pop b, pop a) paths between control states. Ln. 22-25 decides when to terminate the analysis: no new frontier edges and the new store component $\hat{\sigma}''$ is subsumed by the old store $\hat{\sigma}'$. The second condition uses the technique presented in [8]. Otherwise, it recurs to EVAL.

Now I explain Ln. 5-12 in more detail by examining the subroutines that are called. As is shown in Ln. 7, the *raw* new states and edges are obtained from calling STEP (shown in Alg. 3). The algorithm enables the widening strategy in the pushdown reachability algorithm by instrumenting the $\hat{\sigma}$ component (it is *widened* during iteration in EVAL (Ln. 7

```
Algorithm 2: EVAL
   Input: $DSG, IECG$(definition referred to Section 8.1)$, \hat{\sigma},$ working set $W$
   Output: $DSG', IECG', \hat{\sigma}'', W'$
1  $\Delta S, \Delta E, \hat{\sigma}'', W' \leftarrow \emptyset$
2  $(E, S, \hat{q}_0) \leftarrow DSG$
3  $(\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{TF}, \overrightarrow{PSF}, \_, \_) \leftarrow IECG$
4  $IECG' \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
5  for $s \in W$ do
6      for $\hat{\kappa} \in \overrightarrow{TF}(s)$ do
7          for $(g, s_1, \hat{\sigma}') \in$ STEP$(s, \hat{\kappa}, \overrightarrow{PSF}(s), \hat{\sigma})$ do
8              if $\{s\} \not\sqsubseteq (\overrightarrow{G}(s) \cup S \cup \overleftarrow{G}(s))$ then
9                  insert $s_1$ in $\Delta S$
10                 insert $E(s, g, s_1)$ in $\Delta E$
11                 insert $s_1$ in $W'$
12                 $\hat{\sigma}'' = \hat{\sigma}' \sqcup \hat{\sigma}$
13     for $E \in \Delta E$ do
14         switch $E$ do
15             case $(s', \varepsilon, s'')$
16                 $IECG' \leftarrow$ PROPAGATE$(E,\ IECG)$
17             case $(s', g^+, s'')$
18                 $IECG' \leftarrow$ PROCESSPUSH$(E, IECG)$
19             case $(s', g^-, s'')$
20                 $IECG' \leftarrow$ PROCESSPOP$(E, IECG)$
21     $DSG' \leftarrow (E \cup \Delta E, S \cup \Delta S)$
22     if $\hat{\sigma}'' \sqsubseteq \hat{\sigma} \wedge \Delta E == \emptyset$ then
23         return $(DSG', IECG', \hat{\sigma}'', W')$
24     else
25         EVAL$(DSG', IECG', \hat{\sigma}'', W')$
```

and Ln. 12)).

The other important part of the algorithm is STEPIPDS, Alg. 4 shows the details. STEPIPDS does three things: (1) It incorporates the enhanced abstract garbage collection into the pushdown framework by calling `eagc` (Ln. 3). The actual algorithm can be derived from the semantics presented in Chapter 7; (2) it calls the pushdown abstract transition relation function of `next` based on the cleaned state after garbage collection. The semantics presented in Chapter 6 reflect the structure of `next`; (3) it summarizes the stack actions from the newly explored nodes, to construct possible edges for *DSG*. This is done in the Alg. 5, which compares the continuation before the transition and the continuation after, then decides which of the three stack actions, epsilon, push and pop, to take.

Algorithm 3: Step

```
Input: control state $\hat{q}$, continuation $\hat{\kappa}$, a list of frames $\vec{\hat{\phi}}$
Output: a set of records (stack action $g$, $\hat{q}'$, $\hat{\sigma}$)
result $\longleftarrow \emptyset$
for ($g$, $\hat{q}'$) $\in$ StepIPDS($\hat{q}$, $\hat{\kappa}$, $\vec{\hat{\phi}}$) do
    insert ($g$, $\hat{q}'$, $\hat{\sigma}$) in result
return result
```

Algorithm 4: StepIPDS

```
Input: a source state $\hat{q}$, continuation $\hat{\kappa}$, list of frames $\vec{\hat{\phi}}$, Options: global analysis
       options
Output: a set of tuples ($\hat{\phi}'$, $\hat{q}'$)
result $\longleftarrow \emptyset$
$\hat{q}' \longleftarrow \hat{q}$
if Options.doGC then $\hat{q}' \longleftarrow$ eagc($\hat{q}$, $\hat{\phi}$)
confs $\longleftarrow$ next($\hat{q}'$, $\hat{\kappa}$)
for ($\hat{q}''$, $\hat{\kappa}'$) $\in$ confs do
    g $\longleftarrow$ DecideStackAction($\hat{\kappa}$, $\hat{\kappa}'$)
    insert (g, $\hat{q}''$) in result
return result
```

Algorithm 5: DecideStackAction

```
Input: continuation before transition $\hat{\kappa}$, new continuation $\hat{\kappa}'$
Output: stack action $g$
if $\hat{\kappa} = \hat{\kappa}'$ then return $\epsilon$
($g_1 :: \hat{\kappa}_1$) $\longleftarrow \hat{\kappa}$
($g_2 :: \hat{\kappa}'_2$) $\longleftarrow \hat{\kappa}'$
if $\hat{\kappa}_1 == \hat{\kappa}'$ then
    return $g_1^-$
else if $\hat{\kappa} == \hat{\kappa}'_2$ then
    return $g_2^+$
```

Also note that I add only state nodes (*e.g.*, $\hat{q}$) into the working set if they do not appear in the following sets: state nodes of the current *DSG*, predecessors of $\hat{q}$ and successors of $\hat{q}$, for the purpose of avoiding nonnecessary recomputation.

Returning to Eval in Alg. 2, Ln. 13-20 summarizes and propagates the new knowledge of the stack, given ΔE, by calling the algorithms based on stack action, which are defined in Section 8.3, along with the mechanism to deal with exceptions.

8.3 Synthesizing a Dyck state graph with exceptional flow

For pushdown analysis *without exception handling*, only two kinds of transitions can cause a change to the set of ϵ-*predecessors* ($\overleftarrow{G}$): an intraprocedural empty-stack transition and a frame-popping procedure return. With the addition of **handle** frames to the stack, there are several new cases to consider for popping frames (and hence adding ϵ-edges).

In the following text, I first highlight how to handle the exceptional flows during DSG synthesis, particularly as it relates to maintaining ϵ-summary edges. Then I present the generalized algorithms for these cases. The figures in this section use a graphical scheme for describing the cases for ϵ-edge insertion. Existing edges are solid lines, while the ϵ-summary edges to be added are dotted lines.

8.3.1 Intraprocedural push/pop of handle frames

The simplest case is entering a `try` block and leaving a `try` block entirely intraprocedurally—without throwing an exception. Figure 8.1 shows such a case: if there is a handler push followed by a handler pop, the synthesized (dotted) edge must be added.

8.3.2 Locally caught exceptions

Figure 8.2 presents a case where a local handler catches an exception, popping it off the stack and continuing.

8.3.3 Exception propagation along the stack

Figure 8.3 illustrates a case where an exception is not handled locally, and must pop off a call frame to reach the next handler on the stack. In this case, a popping self-edge from control state q' to q' lets the control state q' see frames beneath the top. Using popping

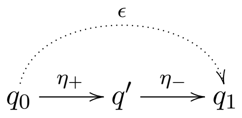

Figure 8.1: Intraprocedural handler push/pop.

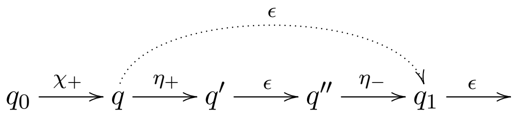

Figure 8.2: Locally caught exceptions.

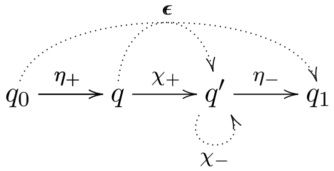

Figure 8.3: Exception propagation.

self-edges, a single state can pop off as many frames as necessary to reach the handle—one at a time.

8.3.4 Control transfers mixed in try/catch

Figure 8.4 illustrates the situation where a procedure tries to return while a **handle** frame is on the top of the stack. It uses popping self-edges as well to find the top-most **call** frame.

8.3.5 Uncaught exceptions

The case in Figure 8.5 shows popping all frames back to the bottom of the stack—indicating an uncaught exception.

8.3.6 `finally` blocks

To analyze full featured exceptions, I have to deal with the `finally` blocks. It is known to be nontrivial to handle `finally` in static analysis [22]. However, this is not a problem in my analysis. The reason is that the analyzer directly works on object-oriented byte code, where the `finally` is compiled away by compiler in this level. Specifically, the blocks of code for `finally` are copied into try and catch blocks before any possible exit points, which include normal `return` statements or `throw` statements. This eases the static analysis

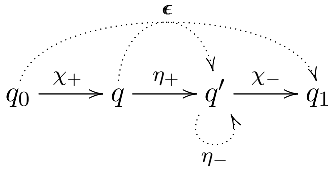

Figure 8.4: Control transfers mixed in try/catch.

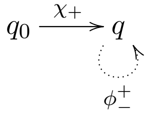

Figure 8.5: Uncaught exceptions.

substantially. In addition, `finally` blocks are translated as one kind of `catch` handler, which is the `catchall` handler, with the exception type `java/lang/Exception`. During the pushdown analysis, `catchall` is placed below any other normal `catch` handlers on the stack, it is matched last and executed for any possible `throw` exceptions.

8.4 The generalized algorithms: Propagate, ProcessPop, ProcessPush

Section 8.3 graphically illustrates the new cases for handling exceptions. The following text presents the working algorithms to achieve the synthesis process. Alg. 6 handles the cases when an ϵ edge is added. These cases are: intraprocedural empty-stack transition, a frame-popping procedure return, or a frame-popping intraprocedural or interprocedural exception catch.

The algorithm works as follows: it accepts an ϵ edge E and the current record of *IECG* (introduced in Section 8.1) and produces a new IECG $IECG'$. It propagates the ϵ successors for each control state in $\overleftarrow{G}(s_1) \cup s_1$ (Ln. 8) and prepares the accumulated top frames for propagation for each successor state node in $\overrightarrow{G}$ (Ln. 12). Similarly, it propagates the ϵ

Algorithm 6: Propagate

```
   Input: An edge$E$, an $IECG$ (refer to Section 8.1)
   Output: $IECG'$
1  $(\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{TF}, \overrightarrow{PSF}, \overleftarrow{PFP}, \overleftarrow{NEP}) \leftarrow IECG$
2  $topFramesToAdd \leftarrow \emptyset$
3  $\overleftarrow{G}', \overrightarrow{G}', \overrightarrow{TF}', \overrightarrow{PSF}', \overleftarrow{PFP}', \overleftarrow{NEP} \leftarrow \emptyset$
4  $(s_1, \varepsilon, s_2) \leftarrow E$
5  $preds \leftarrow \overleftarrow{G}(s_1) \cup \{s_1\}$
6  $nexts \leftarrow \overrightarrow{G}(s_2) \cup \{s_2\}$
7  for $s \in preds$ do
8      $\overrightarrow{G}' \leftarrow \overrightarrow{G} \sqcup [s \mapsto \overrightarrow{G}(s) \cup nexts]$
9      insert $\overrightarrow{TF}(s)$ in $topFramesToAdd$
10 for $s \in nexts$ do
11     $\overleftarrow{G}' \leftarrow \overleftarrow{G} \sqcup [s \mapsto \overleftarrow{G}(s) \cup preds]$
12     $\overrightarrow{TF}' \leftarrow \overrightarrow{TF} \sqcup [s \mapsto \overrightarrow{TF}(s) \cup topFramesToAdd]$
13     for $f \in \overrightarrow{TF}'(s_1)$ do
14         $\overleftarrow{PFP}' \leftarrow \overleftarrow{PFP} \sqcup [(s, f) \mapsto \overleftarrow{PFP}(s, f)]$
15     $\overrightarrow{PSF}' \leftarrow$ updatePSF$(s, \overrightarrow{TF}', \overrightarrow{PSF}, \overleftarrow{NEP}, \overleftarrow{G}')$
16 $IECG' \leftarrow (\overleftarrow{G}', \overrightarrow{G}', \overrightarrow{TF}', \overrightarrow{PSF}', \overleftarrow{PFP}', \overleftarrow{NEP})$
17 return $IECG'$
```

predecessors for each control state in $\overrightarrow{G}(s_2) \cup s_2$. The predecessor nodes of pushed frames for the current target note state s will also be propagated with the new propagated top frames (Ln. 13-14). Finally, it propagates the possible stack frames $\overrightarrow{PSF}$ (for abstract garbage collection) in Ln. 15, for each control state in the original non-ϵ predecessors and new ϵ predecessors $\overleftarrow{G}$, as shown in Alg. 7 Ln. 2-3.

Alg. 8 handles the case of popping frames, including function call return popping and exception handling popping. The algorithm is reduced to Alg. 6 to introduce ϵ edges, for each tuple in $\overleftarrow{PFP}$.

Alg. 9 is presented for completeness. It handles pushing stack frames in function calls and exception handlers in `try` blocks. Since the pushing action introduces a new top frame, it maintains extensions (propagation) for the data structure top frames $\overrightarrow{TF}$, predecessors for push frames $\overleftarrow{PFP}$, non-ϵ predecessors $\overleftarrow{NEP}$ and possible stack frames $\overrightarrow{PSF}$.

```
Algorithm 7: UpdatePSF
  Input: $s, \overrightarrow{TF}', \overrightarrow{PSF}, \overleftarrow{NEP}, \overleftarrow{G}'$
  Output: $\overrightarrow{PSF}''$
1 $\overrightarrow{PSF}' \leftarrow \overrightarrow{PSF} \sqcup [s \mapsto \overrightarrow{TF}'(s)]$
2 for $spred \in \overleftarrow{NEP}(s) \cup \overleftarrow{G}'(s)$ do
3     $\overrightarrow{PSF}'' \leftarrow \overrightarrow{PSF}' \sqcup [s \mapsto \overrightarrow{PSF}'(spred)]$
4 return $\overrightarrow{PSF}''$
```

```
Algorithm 8: Processpop
  Input: $E, IECG$
  Output: $IECG'$
  $IECG' \leftarrow \emptyset$ $(s_1, g^-, s_2) \leftarrow E$
  for $s \in \overleftarrow{PFP}(s_1, g^-)$ do
      $IECG' \leftarrow IECG \sqcup$ Propagate$((s, \varepsilon, s_2), IECG)$
  return $IECG'$
```

Algorithm 9: Processpush

```
Input: $E, \mathit{IECG}$
Output: $\mathit{IECG}'$
$\mathit{IECG}'\ (\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{\mathit{TF}}, \overrightarrow{\mathit{PSF}}, \overleftarrow{\mathit{NEP}}) \leftarrow \mathit{IECG}$
$\overleftarrow{G}', \overrightarrow{G}', \overrightarrow{\mathit{TF}}', \overrightarrow{\mathit{PSF}}', \overleftarrow{\mathit{PFP}}', \overleftarrow{\mathit{NEP}}' \leftarrow \emptyset$
$(s_1, g^+, s_2) \leftarrow E$
for $s \in \overrightarrow{G}(s_2) \cup \{s2\}$ do
    $\overrightarrow{\mathit{TF}}' \leftarrow \overrightarrow{\mathit{TF}} \sqcup [s \mapsto \{f\}]$
    $\overleftarrow{\mathit{PFP}}' \leftarrow \overleftarrow{\mathit{PFP}} \sqcup [(s, f) \mapsto \{s_1\}]$
    $\overleftarrow{\mathit{NEP}}' \leftarrow \overleftarrow{\mathit{NEP}} \sqcup [s \mapsto \{s_1\}]$
    $\overrightarrow{\mathit{PSF}}' \leftarrow$ UpdatePSF$(s, \overrightarrow{\mathit{TF}}', \overrightarrow{\mathit{PSF}}, \overleftarrow{\mathit{NEP}}', \overleftarrow{G})$
$\mathit{IECG}' \leftarrow (\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{\mathit{TF}}', \overrightarrow{\mathit{PSF}}', \overleftarrow{\mathit{PFP}}', \overleftarrow{\mathit{NEP}}')$
return $\mathit{IECG}'$
```

CHAPTER 9

MULTIENTRY POINTS SATURATION

The pushdown control-flow analysis described in the previous section provides the foundation for my object-oriented analysis. Now, I shift my focus to addressing the Android specific challenge: asynchronous multiple entry points using entry point saturation (EPS) and integrating it into pushdown control-flow analysis.

9.1 Entry points discovery

An entry point is defined as any point through which the system can enter the user application [23]. This means that any method that can be invoked by the framework is an entry point. Since there is no single "main" method, the static analysis must first identify the entry points in the program. Entry point discovery is not a challenge, however, since they are defined by the Android framework. I briefly summarize possible entry points here.

There are three categories of entry points, which I generalize as *unit*s. First, all the callback events of components defined by the Android framework are entry points. These entry points are designed to be overridden by application code and are invoked and managed by the framework for the purpose of component life cycle management, coordinating among different components, and responding to user events which are themselves defined to be asynchronous. Second, asynchronous operations that can be executed in the background by the framework are considered to be entry points. These include the *AsyncTask* class for short background operations, the *Thread* class for longer operations, and the *Handler* class for responding to messages. Finally, all event handlers in Android user interface (UI) widgets, such as button, check box, *etc.*are entry points. Each UI widget has standard event listeners defined, where the event handler interface methods are meant to be implemented by application code. Entry points from the first two categories are found by parsing Dalvik bytecode and organized into a set attached to the corresponding *unit.* Entry points from the final category can also be defined in resource layout files *res/layout/filename.xml.* These entry points can be obtained by parsing files before analysis.

9.2 The approximation of multiple entry points execution—entry point saturation (EPS)

After the entry point set for each unit is determined the real challenge begins.

Intuitively, if we want to do a sound analysis, all the permutations of the entry points need to be considered. The permutations of the entry points differ from the interleavings of concurrent constructs, such as threads. They are not my concern, because concurrency-incurred control-flows and data-flows are very unreliable in being produced or exploited in the context of security vulnerability identification [5]. Therefore, the focus is how to "soundly" approximate the permutations of the entry points under the Android framework. However, it is possible to model thread interleavings in the spirit of EPS. I will discuss this more after illustrating EPS.

Entry point saturation (EPS) directly relies on the underlying pushdown analytic engine presented in Chapter 8. Figure 9.1 illustrates the process. For each entry point E_i in a *unit* (represented as a square), I compute the fixed point via pushdown analysis (refer to Section 8.2). After one round of computation the analysis returns a set of configurations. I then use the *configuration widening* technique from Might [15] on the set of configurations to generate a widened $\hat{\sigma}_i$. This abstract component will be "inherited" by the next entry point in the fixed point computation. The process repeats until the last entry point finishes its computation in a *unit*. In this way, the unit has reached a fixed point. The next step computes the fixed point between *unit*s. This is computed in a similar fashion to the intra-unit fixed point computation, so the widened result $\hat{\sigma}'_n$ from the previous unit (the left square) participates in the reachability analysis of the next unit (the right square).

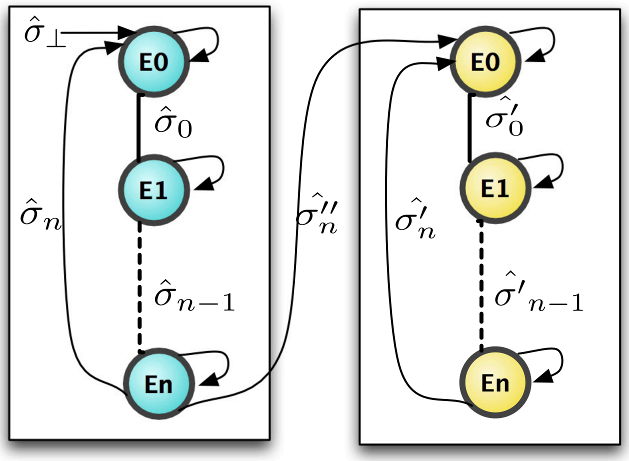

Figure 9.1: Entry point saturation.

So, now we can see that EPS is not based on permutations of entry points and their interleaving execution. It models such behavior by passing the store resulting from analyzing one entry point as the initial store for the next entry point. This gives the “saturated” store for a given component. The saturated store for a component is similarly passed as the initial store for the next component. The final store models every execution path without needing to enumerate every combination, and the effect of every execution path is “stabilized” eventually. In brief, EPS has several advantages:

1. It approximates multiple entry point executions without enumerating all the permutations and analyzing each one of them.
2. It computes the fixed point from bottom up, intraentry point, interentry point, and interunit, in an efficient manner and significantly simplifies the static analysis;
3. In the same spirit, it *can* compute the analysis of thread interleaving executions via a more aggressive widening—global widening. It uses a single $\widehat{Store}$ across each statement, entry point and component. The technique can diminish precision but the performance gain can be considerable [8, 15].
4. The soundness is easy to prove, because the soundness proof for widening on the abstract configurations follows the same structure as shown in [15], thus I omit it here.

CHAPTER 10

STATIC TAINT FLOW ANALYSIS IN SMALL-STEP ABSTRACT INTERPRETATION

According to the open web application security project (OWASP) [24], cross site scripting (XSS) and structured query language (SQL) injection have been two of the top ten most common security threats for several years. Both are caused by unsanitized user input flowing into unsafe or privileged sinks such as system calls, database queries or, in modern times, client-side HTML. In mobile apps, particularly Android malware, security analysis mostly concerns the flows of sensitive private user information. The taint source is not necessarily just user input, but can be from sensors, GPS, local file system or SD cards, *etc.*

Taint analysis has been proposed to solve this problem [25, 26], by tracking and detecting whether tainted values (usually unsanitized user input) may flow into security sinks. This can be done dynamically and statically. Generally speaking, the main strength of dynamic taint analysis is that it can track tainted information through direct data dependencies in an efficient and precise way [27, 28, 29, 30, 31, 32]. However, strictly dynamic analyses cannot prevent soft failure of the application when an unexpected taint violation happens [33, 34, 35].

Static taint analysis can prove the absence of taint violations, or at least delimit the regions in which they may happen. However, it is subject to false positives. Therefore, a precise static taint analysis is of particular importance.

The strategy to achieve a highly precise static *taint* analysis in this work is to retrofit the principled and highly precise analysis framework foundation established in previous chapters. However, the principled analysis framework does not deal with taint values as it does for different types of data in a specific programming paradigm, not to mention the flows of the taint values.

This chapter describes the technique to empower small-step abstract interpretation with security analysis—taint-flow analysis. Section 10.1 describes the technique to integrate taint analysis in small-step abstract interpretation. It presents necessary details of refactoring on abstract semantics for taint-flow analysis in both traditional abstract interpretation and pushdown exception-flow abstract interpretation. Section 10.2 describes how to form taint lattice, including the simplest two-valued taint lattice and multivalued subset-based taint lattice for Android malware. To demonstrate how the static taint-flow precisely track tainted data in abstract interpretation, Section 10.3 details some walk throughs on some challenging running examples.

10.1 Static taint-flow analysis in small-step abstract interpretation

The finite-state abstract interpretation—classical k-CFA described in Chapter 5, Section 5.2.1—shows a context-, object- and field-sensitive analysis. A static taint analysis built upon this analysis can retrofit these advantages. The pushdown abstract interpretation in Chapter 6 and Chapter 7 presents an analysis free of the limited level of context- and object-sensitivity. A static taint analysis based on this framework can gain even more precision. This section describes the generalized technique of integrating taint-flow analysis in both analysis schemes.

The core of integrating taint-flow analysis in these small-step abstract interpretation is to track the tainted values in a "small-step" fashion. In other words, whenever a state transits to next possibles state(s), the analysis will extend the effects of tainted values.

Therefore, here we only need to add a "taint" component into state space $\widehat{State}$ to record this change, formulated as follows:

$$\hat{\sigma}^T \in \widehat{\boldsymbol{TaintStore}} = \widehat{Addr} \rightharpoonup \widehat{D}.$$

The definition of $\widehat{D}$ now include the abstract tainted values:

$$\hat{d} \in \widehat{D} = \mathcal{P}\left(\widehat{ObjectValue} + \widehat{String} + \widehat{\mathcal{Z}} + \widehat{\mathcal{B}} + \widehat{TaintVal}\right)$$

The taint values are opaque for now. This issue will be resolved in Section 10.2.

Further refactoring specific to the state space of two kinds of abstract interpretation:

- In finite-state small-step abstract interpretation:

$$\widehat{Conf} = \mathsf{Stmt}^* \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{\boldsymbol{TaintStore}} \times \widehat{KontAddr} \times \widehat{Time}$$

- In pushdown abstract interpretation:

$$\widehat{Conf} = \mathsf{Stmt}^* \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{\boldsymbol{TaintStore}} \times \widehat{Kont} \times \widehat{Time}$$

We also need to refine the atomic expression evaluator $\hat{\mathcal{A}}$ and $\hat{\mathcal{A}}_{\mathcal{F}}$ to get tainted values for variables and fields, respectively:

$$\hat{\mathcal{A}}^T : \mathsf{AExp} \times \widehat{FramePointer} \times \widehat{TaintStore} \rightharpoonup \widehat{D} : \quad \hat{\mathcal{A}}^T(name, \hat{fp}, \hat{\sigma}^T) = \hat{\sigma}^T(\hat{fp}, name)$$

$$\hat{\mathcal{A}_{\mathcal{F}}}^T : \mathsf{AExp} \times \widehat{FramePointer} \times \widehat{TaintStore} \times Store \times \mathsf{FieldName} \rightharpoonup \widehat{D} :$$

$$\hat{\mathcal{A}_{\mathcal{F}}}^T(\text{æ}, \hat{fp}, \hat{\sigma}^T, \hat{\sigma}, \mathit{field\text{-}name}) = \hat{\sigma}^T(\widehat{op}, \mathit{field\text{-}name}), \text{ where } (\widehat{op}, \mathit{class\text{-}name}) \in \hat{\mathcal{A}}(\text{æ}, \hat{fp}, \hat{\sigma}).$$

When looking up taint values for fields, we need the normal store $\hat{\sigma}$ to first find the base pointer for the field, then look into the taint store $\hat{\sigma}^T$ to get the taint values for the fields.

With the refactoring defined for state space of taint-flow analysis, the transition relation $\leadsto$ can propagate tainted values without any modification of the existing mechanics. In other words, all the transition rules have an additional $\hat{\sigma}^T$ added, operations resemble the ones on $\hat{\sigma}$, except that the taint values are *monotonically* propagated. For example, in a function call, tainted values in arguments are bound to the formal parameters of the functions (using the $\sqcup$ operation in the taint store $\hat{\sigma}^T$), returning abstract taint values bound to a register address with ret, *etc.* The detailed formalism is omitted to save space due to much of the duplication.

What is worth pointing out is that, the taint-flow analysis derived in this way needs *no* modification to the (enhanced) abstract garbage collection (Chapter 7). Abstract garbage collection related semantics are mainly computing *reachable* addresses, where the taint store $\hat{\sigma}^T$ and original store $\hat{\sigma}$ have the same domain, which are the set of abstract addresses defined in $\widehat{Addr}$. This nonintrusive approach simplifies the construction of the taint analysis, more importantly, it significantly eases the exploitation of underlying analysis techniques.

Section 10.3 exemplifies how the taint-flow analysis tracks tainted data precisely. Chapter 13, Section 13.3 reports the overall effectiveness of the taint-flow analysis to identify Android malware in pushdown exception-flow analysis framework.

10.2 Taint lattice

Section 10.1 introduces the idea of integrating taint-flow analysis in small-step abstract interpretation. It leaves the taint values opaque. This section illustrates two kinds of lattices as the abstract taint values.

10.2.1 Simple taint lattice

One of the simplest form of taint lattice is two-valued lattice with the values of "tainted" and "untainted," as shown in Figure 10.1.

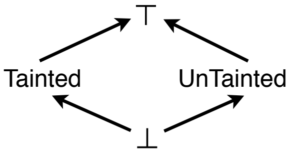

Figure 10.1: Simple taint lattice.

The lattice is useful in specifying unsanitized user input in web apps. It has been used to detect XSS attacks for Python web apps [36].

10.2.2 Multivalued flat taint lattice

The two valued lattice is insufficient to specify the taint values in Android apps. Any data that involve user private information can be tainted values, such as contacts, SMS, pictures in local file system, *etc.* To have a fine-grained analysis of taint values, I have refined the simple taint lattice to be a multivalued subset-based taint lattice:

$$\begin{aligned}
\hat{d} \in \widehat{\mathit{Val}} &= \mathcal{P}\left(\widehat{\mathit{ObjectValue}} + \widehat{\mathit{String}} + \widehat{\mathcal{Z}} + \mathit{TaintVal}\right) \\
\mathit{TaintVal} &= \mathrm{Location} + \mathrm{FileSystem} + \mathrm{Sms} + \mathrm{Phone} + \mathrm{Voice} + \mathrm{Contact} \\
&\quad + \mathrm{DeviceID} + \mathrm{Network} + \mathrm{ID} + \mathrm{TimeOrDate} + \mathrm{Sensor} \\
&\quad + \mathrm{Display} + \mathrm{Reflection} + \mathrm{BrowserBookmark} + \mathrm{IPC} + \mathrm{Thread} \\
&\quad + \mathrm{BrowserHistory} + \mathrm{SdCard} + \mathrm{Picture} + \mathrm{Account} + \mathrm{Media}.
\end{aligned}$$

10.3 Examples of precise taint-flow analysis in abstract interpretation

Since semantics of the taint-flow analysis resembles much of the existing semantics omitted in this chapter, it would be helpful to exemplify how to track taint values by leveraging underlying analysis framework. Listing 10.1 shows some condensed code snippet that could trigger security violation reports.

Listing 10.1: Example code to show precise static taint-flow analysis in abstract interpretation

```
void main() {
  A a = new A();
  B b = a.g;
  foo(source(), a);
  sink(b.f);
  A c = new A();
  foo("clean", c);
  B d = c.g;
```

```
 9    sink(d.f);
10  }

12  void foo(String str, A z) {
13    B x = z.g;
14    String w = str;
15    x.f = w;
16  }
```

I will show how analysis runs in both normal store and taint store in particular. In taint store, the tainted values are indicated with $tainted_i$. v is the prefix to the name of the variables and formal parameters. Frame pointer $\hat{fp}$ is abbreviated with the subscript indicating last one call-site with function name if applied. Object pointer $\widehat{op}$ is abbreviated with the subscript indicating last one allocation site (line number) with the type name.

For the purpose of demonstrating the capability of my analysis, I start from the 0CFA and 1-object sensitivity. So initial states and the ones after executing lines 1-2 (suppose the nested field object "g" is instantiated somewhere with $\{(\widehat{op}_{B0}, B)\}$):

$$(\hat{fp}_{main}, v_a) \mapsto \{(\widehat{op}_{A2}, A)\}, \quad (\widehat{op}_{A2}, \text{“g”}) \mapsto \{(\widehat{op}_{B0}, B)\}$$
$$(\hat{fp}_{main}, v_b) \mapsto \{(\widehat{op}_{B0}, B)\}, \quad (\widehat{op}_{B0}, \text{“f”}) \mapsto \{\}.$$

10.3.1 Propagating tainted values

In line 4, source returns tainted data. When calling foo, the formal parameter of str is tainted at this point:

$$(\hat{fp}_{foo}, v_{str}) \mapsto \{tainted_0\},$$
$$(\hat{fp}_{foo}, v_z) \mapsto \{(\widehat{op}_{A2}, A)\}, \quad (\hat{fp}_{foo}, v_x) \mapsto \{(\widehat{op}_{B0}, B)\}$$
$$(\hat{fp}_{foo}, v_w) \mapsto \{tainted_0\}, \quad (\widehat{op}_{B0}, \text{“f”}) \mapsto \{tainted_0\}.$$

When executing line 5, the sink will report the risk of leaking sensitive data, because now $(\widehat{op}_{B0}, \text{“f”})$ is mapping to $\{tainted_0\}$.

10.3.2 Traditional k-CFA and object sensitivity to refine taint information

In line 6 of Listing 10.1, the program instantiates another object of type A, since I use one level of object-sensitivity, I can distinguish the two objects of type A. Suppose the nested field object "g" is instantiated somewhere with $\{(\widehat{op}_{B1}, B)\}$:

$$(\hat{fp}_{main}, v_c) \mapsto \{(\widehat{op}_{A6}, A)\}, \quad (\widehat{op}_{A6}, \text{“g”}) \mapsto \{(\widehat{op}_{B1}, B)\}$$
$$(\hat{fp}_{main}, v_d) \mapsto \{(\widehat{op}_{B1}, B)\}, \quad (\widehat{op}_{B1}, \text{“f”}) \mapsto \{\}.$$

In line 7 of Listing 10.1, it calls foo again. Without any level of call-context-sensitivity, the following happens:

$$\begin{aligned}
(\hat{fp}_{foo}, v_{str}) &\mapsto \{tainted_0, \text{“clean”}\} \;\; \text{// values merging!}\\
(\hat{fp}_{foo}, v_z) &\mapsto \{(\widehat{op}_{A2}, A), (\widehat{op}_{A6}, A)\} \;\; \text{//values merging!}\\
(\hat{fp}_{foo}, v_x) &\mapsto \{(\widehat{op}_{B0}, B), (\widehat{op}_{B1}, B)\} \;\; \text{//values merging!}\\
(\widehat{op}_{B0}, \text{“f”}) &\mapsto \{tainted_0\}\\
(\hat{fp}_{foo}, v_w) &\mapsto \{tainted_0, \text{“clean”}\} \;\; \text{//values merging!}\\
(\widehat{op}_{B1}, \text{“f”}) &\mapsto \{tainted_0, \text{“clean”}\} \;\; \text{//values merging!}.
\end{aligned}$$

The merging records in the taint store and normal value store are annotated. In particular, the imprecision in str causes the imprecision of v_w and so $(\widehat{op}_{B1}, \text{“f”})$. Its mapped tainted value will falsely flow into sink in line 9.

With 1-call-site context-sensitivity, we can distinguish call site of foo by one level. That is, the call site in line 4 is distinguished from the call site in line 7. So:

$$\begin{aligned}
(\hat{fp}_{foo7}, v_{str}) &\mapsto \{\text{“clean”}\}\\
(\hat{fp}_{foo7}, v_z) &\mapsto \{(\widehat{op}_{A6}, A)\}\\
(\hat{fp}_{foo7}, v_x) &\mapsto \{(\widehat{op}_{B1}, B)\}\\
(\hat{fp}_{foo7}, v_w) &\mapsto \{\text{“clean”}\}\\
(\widehat{op}_{B1}, \text{“f”}) &\mapsto \{\text{“clean”}\}.
\end{aligned}$$

We can see that value merging problem for v_{str}, v_z, v_x, v_w is solved and so that the field “f” does not have any tainted value. So, sink in line 7 does not report a violation.

10.3.3 Abstract garbage collection to refine taint-flow precision without context-sensitivity

What is worth pointing out is that context-sensitivity is easily subject to program constructs, that is, a function can be easily nested in an arbitrary level, any fixed number of level of call-site context-sensitivity can fail at some point. Therefore, the abstract garbage collection in object-oriented setting becomes very useful to refine precision without any level of context-sensitivity and field-sensitivity.

So, rewind to the point of line 7. Before executing this line, if abstract garbage collection is applied, even with no context-sensitivity (0CFA), we can get clean stores as shown below:

$$
\begin{aligned}
(\hat{fp}_{foo}, v_{str}) &\mapsto \{\} \\
(\hat{fp}_{foo}, v_z) &\mapsto \{\} \\
(\hat{fp}_{foo}, v_x) &\mapsto \{\} \\
(\hat{fp}_{foo}, v_w) &\mapsto \{\} \\
(\widehat{op}_{B1}, \text{“f”}) &\mapsto \{\}.
\end{aligned}
$$

With these cleaned stores, when applying foo in line 7, the precise result is shown as follows:

$$
\begin{aligned}
(\hat{fp}_{foo}, v_{str}) &\mapsto \{\text{“clean”}\} \\
(\hat{fp}_{foo}, v_z) &\mapsto \{(\widehat{op}_{A6}, A)\} \\
(\hat{fp}_{foo}, v_x) &\mapsto \{(\widehat{op}_{B1}, B)\} \\
(\hat{fp}_{foo}, v_w) &\mapsto \{\text{“clean”}\} \\
(\widehat{op}_{B1}, \text{“f”}) &\mapsto \{\text{“clean”}\}.
\end{aligned}
$$

This analysis result at this point will not report false positive in the call site sink in line 9.

10.3.4 Abstract garbage collection to refine field taint precision in alias case

Listing 10.2 is a code snippet that involves an alias. For brevity, I keep the foo the same as Listing 10.1. Following the previous sections, I demonstrate how garbage collection can help refine field taint information in particular. So, aa and a are aliases. As Section 10.3.2 shows, line 6 will reports violation, since after the call of foo in line 5, field f is tainted, and there is a record in taint store: $(\widehat{op}_{B0}, \text{“f”}) \mapsto \{tainted_0\}$.

Listing 10.2: Example code to show precise static taint-flow analysis in abstract interpretation-alias case

```
1 void main() {
2     A a = new A();
3     A aa = a;
4     B b = a.g;
5     foo(source(), a);
6     sink(b.f);
7     foo("clean", aa);
8     sink(b.f);
9 }
```

The Listing 10.2 calls foo again in line 7 with the argument value of “clean” and the alias of a, aa, sink in line 8 should not report a violation.

Conservative analysis can falsely report if it is unable to track the alias, or if it is unable to distinguish “active” values of an alias that is of concern in the current calling frame.

Tracking an alias is not an issue in my framework, with any arbitrary level of field reference. What is more worth noting is that, with abstract garbage collection applied right before applying the call foo in line 7, the values of the entry indexed by the field address $(\widehat{op}_{B0}, \text{“f”})$ are eliminated. So when applying foo in line 7 again, $(\widehat{op}_{B0}, \text{“f”}) \mapsto \{\text{“}clean\text{”}\}$ and thus line 8 will not report violation.

10.3.5 Pushdown framework to refine taint-flows in the presence of exceptions

Listing 10.3 is a snippet code that shows source information can flow out of the sinks via exception handlers. It is a simplified version of the work [6]. As shown in the Listing 10.3, function foo calls bar. The bar is a function that can throw exception. The bar can be called elsewhere, for instance, baz.

Listing 10.3: Example code to show precise static taint-flow analysis in abstract interpretation-exception case

```
 1 foo() {
 2   try{
 3     String str = source();
 4     bar(); //may throw exceptions.
 5   }
 6   catch(Exception e) {
 7     sink(str);
 8   }
 9 }

11 baz() {
12   try{
13     String str = "clean";
14     bar(); // may throw exceptions
15   }
16   catch(Exception e){};
17 }
```

Analysis with 1-call-site context-sensitivity is insufficient in this case since bar is nested inside functions. The level of context-sensitivity should be 2 to distinguish the two different calls to bar. However, the real challenge for existing analysis is the exceptions. Ignoring exception handling code is unsound, although it is not trivial to figure out where the control-flow should go, in particular when the handlers are located in deep upper caller chains. Imprecise analysis of exceptions usually work this way: when calling either bar in line 14 or the one in line 4 encounters an exception, conservative analysis might go to handler in line

16 as well as the one in line 6-8. This kind of problem is documented in [37].

This over-approximation can be refined in the pushdown exception analysis framework (Chapter 6). When bar in line 4 is called and an exception is thrown, it “jumps” out of the current call frame of bar, locates its caller foo and finds the exception handler that was installed in the caller. Thus, the analysis is able to track the tainted source flows into sink in line 7. On the other hand, when the function baz is called in line 14, it would not falsely resolve the handler in line 6-8 and report false positives.

At this point, I have demonstrated how to do precise and sound (with respect of analyzing exceptions) taint-flow analysis by retrofitting the underlying analysis framework. Chapter 13, Section 13.3 shall report the overall effectiveness of the taint-flow analysis in identifying Android malware in a pushdown exception-flow analysis framework.

CHAPTER 11

GÖDEL HASHES FOR ACCELERATING STATIC ANALYSIS

Computing static analysis as an abstract interpretation is much like a graph search, except that instead of checking whether the current state has visited, the algorithm checks whether the current state is subsumed by a state which has already been visited. Either the finite-state-based analysis or the pushdown version, the subsumption comparison is the core of the fix-point computation.

For example, in classical control-flow analysis, the fix-point computation is as follows [15]:

```
$Todo := \{(call, \bot_{\widehat{Store}})\}$
$Seen := \emptyset$
while $(Todo \neq \emptyset)$
    $\hat{\varsigma} := choose(Todo)$
    $Todo := Todo - \{\hat{\varsigma}\}$
    if $(\{\hat{\varsigma}\} \not\sqsubseteq Seen)$
        $Seen := Seen \cup \{\hat{\varsigma}\}$
        $next := \left\{\hat{\varsigma}' : \hat{\varsigma} \leadsto \hat{\varsigma}'\right\}$
        $Todo := next \cup Todo$
```

For a path-sensitive k-CFA, it is an implementation of flow-sensitive control-flow analysis with unrestricted heap-cloning. This kind of k-CFA could perform on the order of $O(n^2((2^n)^n)^2)$ subsumption tests [8, 38, 15]: programs as small as 100 lines can easily visit well over a hundred thousand states, requiring over a billion subsumption tests between *abstract heaps*.

In the pushdown exception-flow analysis, which is derived from the polynomial complexity algorithm in [16, 18], the subsumption test between *abstract heaps* accounts for a lot in the fix-point computation of pushdown exception-flow analysis. The Alg. 2 in Chapter 8, Section 8.2 illustrates this.

Gödel hashes are developed to transform the expensive operation during fixed point computation into numerical operations for the purpose to speed up the analysis.

11.1 Outline

The rest of the section in this chapter is organized as follows: Section 11.2 introduces the idea of Gödel hashing and its characteristics of such encoding scheme. Section 11.3 reviews some preliminary mathematics and notations.

Then in Section 11.4, I first explore Gödel hashes on sets by defining a perfect hashing function that exploits the fundamental theorem of arithmetic; it maps set-theoretic operations on values (*e.g.*, union, intersection, subset-inclusion) into arithmetic operations on hashed values (*e.g.*, lcm, gcd, divisibility).

I conduct a formal analysis of the worst- and average-case space usage of the set hashes, and then provide a *space-optimality* result when the distribution of elements amongst sets is known *a priori.* Following the analysis of space usage, I analyze the speed of operations on Gödel hashes, which reveals that, asymptotically, some operations on Gödel hashes are *worse* than the equivalent operations on the original values.

However, by "unhiding" the hidden constant factors—the speed-up from word-sized arithmetic operations for the hashes and the cost of cache misses for the original values—I find the potential for operations on Gödel hashes to be significantly faster (a point later validated by my empirical trials).

With Gödel sets defined, Section 11.5 shows that many recursively constructed, partially ordered data structures have order-preserving (monotonic) Gödel hashes. I do so by finding a condition on partial orders, factorability, which implies the existence of an order-preserving Gödel hash, and show that common set constructors preserve factorability.

Since other data structures are less relative to static analysis, the extension of Gödel hashes for maps, relations, and graphs, multisets, and lists are arranged in Appendix C.

I then briefly discuss computing the prime numbers necessary for Gödel hashing to work. This is described in Section 11.6.

The evaluation on Gödel hashes sets with respect to compactness in space and efficiency, and the application to speedup static analysis is discussed in Chapter 13, Section 13.2 and Section 13.2.4, respectively.

11.2 The idea of Gödel hashes

Gödel's incompleteness proofs [39] used clever strategies for encoding complex mathematical structures as individual natural numbers. With occasional help from Cantor [40], we focus on one of the strategies Gödel used for encoding propositions, and I extend it to encode sets, multisets, lists, maps, relations, graphs, and partial orders. I term these encodings Gödel *hashes* because they are compact and because inverting the encoding is impossible.[1] Compared to traditional hashing strategies, Gödel hashes possess six attractive properties:

1. **Gödel hashes are perfect hashes.** For an ordinary hash function, inequality of the hashes implies inequality of the original objects, but equality of hashes does not imply the equality of the original objects. For a perfect hash function, the hashes of two objects are equal if and only if the objects are equal.
2. **Gödel hashes are dynamic.** Unlike other perfect hashing schemes, it is not necessary to know all of the items that may be hashed in advance in order to compute a Gödel hash. Moreover, if the probability distribution of objects to be hashed *is* known in advance, then the average Gödel hash will be minimal.
3. **Gödel hashes are structurally incremental.** Given a value and its hash, it is efficient to incrementally update the hash without recomputation when the value is extended. It is also straightforward to compute the Gödel hash of a complex value from the Gödel hash of its components.
4. **Gödel hashes are compact.** Because Gödel hashes are dynamic and perfect, they are necessarily unbounded in size. Even so, it is straightforward to reason about their size in advance in both worst and average cases, and even their worst case is remarkably compact under pessimistic assumptions. For example, *in the worse case*, the Gödel hash of a set achieves a density of more than one element per 64-bit word until 2^{58} elements are in the universe. The average case is slightly more compact.
5. **Gödel hashes are efficient.** Operations on Gödel hashes are efficient, and they are easy to implement in most modern programming languages, thanks to their built-in support for arbitrary-precision integer arithmetic. In particular, multiple precision arithmetic can be efficiently implemented by modern CPUs' single instruction, multiple data (SIMD), instructions.
6. **Gödel hashes are partial-order-preserving.** Gödel hashes are order-preserving (monotonic) for a variety of partial orderings. For instance, if:

[1] This statement is false. It is actually *computationally intractable* to invert.

$$H : \mathbb{X} \to \mathbb{H}$$

computes the Gödel hash of an element in the set $\mathbb{X}$, and the relation $(\subseteq)$ orders $\mathbb{X}$ while the relation $(\sqsubseteq)$ orders $\mathbb{H}$, then:

$$x \subseteq y \text{ iff } H(x) \sqsubseteq H(y).$$

Critically, the relation $\sqsubseteq$ has a fast, efficient arithmetic implementation. I show that all *factorable* partial orders (defined in Section 11.5) have a Gödel hashing scheme, and I show that factorability is preserved across many set-construction operations. Thus, even complex partially-ordered data structures have an order-preserving Gödel hash. Existing hashing techniques preserve *total* orders, so this opens up new possibilities for the use of hashes (Chapter 13, Section 13.2.4) on pushdown exception-flow for object-oriented programs).

11.2.1 Key idea

Even though the precise algorithm for constructing a Gödel hash differs from one kind of structure to another, all Gödel hashing techniques that I discuss in detail exploit the same principle—the fundamental theorem of arithmetic:

> *Every natural number has a unique decomposition as the product of prime factors.*

I find links between insertion and multiplication, between removal and division, and between subsumption and divisibility. I generalize these links to cover additional structures, such as partial orders.

The message is that: Gödel hashes are feasible and useful for applications in which hashes need to compactly preserve structure.

11.3 Preliminaries: background and notations

The set $\mathbb{N} = \{0, 1, 2 \ldots\}$ is the set of natural numbers. The set $\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$ is the set of integers. The set $\mathbb{P} = \{2, 3, \ldots\}$ is the set of prime numbers. The number p_i is the ith smallest prime number, where $p_0 = 2$.

To enhance readability and notational symmetry, I use the operations greatest common divisor (gcd) and least common multiple (lcm) in infix form. There are many equivalent definitions of these operations, but this work exploits a particular (and uncommon) interpretation of these operations: as functions which minimize or maximize the exponents of two natural numbers in a factor-wise fashion; the greatest common divisor of two natural numbers $n = p_0^{m_0} \cdots p_n^{m_n}$ and $n' = p_0^{m'_0} \cdots p_n^{m'_n}$ is:

$$n \text{ gcd } n' = p_0^{\min(m_0, m'_0)} \cdots p_n^{\min(m_n, m'_n)},$$

and their least common multiple is:

$$n \text{ lcm } n' = p_0^{\max(m_0, m'_0)} \cdots p_n^{\max(m_n, m'_n)}.$$

I also use the divisibility relation:

$$a \mid b \text{ iff } b \bmod a = 0.$$

My proofs often employ characteristic functions. The characteristic function of a set A is the function $\chi_A : A \to \{0, 1\}$:

$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A. \end{cases}$$

I use the bar brackets to denote set cardinality: $|A|$ is the cardinality of the set A.

The function ln denotes the natural logarithm ($\log_e$) and the function lg denotes the binary logarithm ($\log_2$).

I make use of Cantor's bijection, $C^* : \mathbb{Z}^* \to \mathbb{N}$. There are many such bijections (but Cantor first proved their existence [40]). Any such bijection will work. For example, for the simple case of assigning pairs of naturals to a unique natural, the function $C_2^+ : \mathbb{N}^2 \to \mathbb{N}$ works:

$$C_2^+(x, y) = \sum_{i=0}^{x} i + \sum_{j=x+2}^{y+2} j$$

11.3.1 Gödel hashing notations

For each hashable structure X that I study, I will define a Gödel-hashing function $G_X : X \to \mathbb{N}$, so that $G_X(x)$ is the Gödel encoding of the value x. To unclutter notation, I will often shorthand $G_X(x)$ as $\|x\|_X$ or just $\|x\|$ when it is clear what X is.

11.4 Sets

Sets—unordered collections of elements—abound in functional (and nonfunctional) programming. For set-intensive applications, performance hinges on two factors: (1) the space-efficiency of the underlying data structure and (2) the time-efficiency of operations on such structures: membership testing, inclusion testing, intersection, insertion, union, deletion and difference. A Gödel strategy for encoding sets delivers pragmatic efficiency in both dimensions.

I devote more details to the study of Gödel hashes of sets, because many of the results on sets (*e.g.*, correctness, efficiency, optimality) generalize to other data structures.

To construct the Gödel encoding of a set, first assume that every potential element has been assigned a unique prime number; then compute the product of the primes assigned to each element; the result is the Gödel hash of the set. It is not necessary to preconstruct the assignment from elements to primes: new elements may be assigned fresh primes as they are encountered for the first time.

Example 1 *If the potential elements of a set are A, B, C and D, then I can assign these elements the primes 2, 3, 5 and 7, respectively. Thus, the Gödel hash of the set $S = \{A, C\}$ is the natural number* $2 \times 5 = 10$.

On the Gödel hash of a set, familiar number-theoretic operations become set-theoretic operations: modulo tests both membership *and* subset-inclusion; union becomes the least common multiple; and intersection becomes the greatest-common divisor.

11.4.1 Formal definition of Gödel encoding for sets

A universe of discourse, denoted $\mathbb{U}$, which may be either finite or infinite, is a collection of all the elements that may appear in a set. A **prime map** for a universe $\mathbb{U}$ is an injective function $P_{\mathbb{U}} : \mathbb{U} \to \mathbb{P}$, which maps every element in the universe $\mathbb{U}$ to a unique prime number. In practice, the implementation may assign primes dynamically while memoizing them; or if the elements of the universe themselves have a perfect hash map, $H : \mathbb{U} \to \mathbb{N}$,[2] then a purely functional prime map may be used:

$$P_{\text{pure}}(u) = p_{H(u)},$$

which lends itself to a recursive strategy for constructing prime maps.

Definition 11.1 *The function $G_{\mathcal{P}(\mathbb{U})} : \mathcal{P}(\mathbb{U}) \to \mathbb{N}$ computes the* ***Gödel hash of a set:***

$$G_{\mathcal{P}(\mathbb{U})}\{u_1, \ldots, u_n\} = \| \{u_1, \ldots, u_n\} \| = P(u_1) \times \cdots \times P(u_n).$$

Equivalently, the Gödel hash of a set may also be defined through its characteristic function:

$$G_{\mathcal{P}(\mathbb{U})}(A) = \prod_{u \in \mathbb{U}} P(u)^{\chi_A(u)}.$$

[2] Created, perhaps, by Gödel hashes in the internal structure of elements.

(When only one universe is under consideration, the subscripts may be left off.)

11.4.2 Set-theoretic operations and relations

I can construct the standard set-theoretic operations and relations out of arithmetic.

Lemma 11.1 *Union reduces to least common multiple:*

$$\|A \cup B\| = \|A\| \operatorname{lcm} \|B\|.$$

Proof. Let $A, B \subseteq \mathbb{U}$.

$$\begin{aligned} \|A \cup B\| &= \textstyle\prod_{u \in \mathbb{U}} P(u)^{\chi_{A \cup B}(u)} \\ &= \textstyle\prod_{u \in \mathbb{U}} P(u)^{\max(\chi_A(u), \chi_B(u))} \\ &= \operatorname{lcm}\left(\textstyle\prod_{u \in \mathbb{U}} P(u)^{\chi_A(u)}, \textstyle\prod_{u \in \mathbb{U}} P(u)^{\chi_B(u)}\right) \\ &= \|A\| \operatorname{lcm} \|B\|. \end{aligned}$$

■

Lemma 11.2 *Intersection reduces to the greatest common divisor:*

$$\|A \cap B\| = \|A\| \gcd \|B\|.$$

Proof. By argument analogous to the previous proof. ■

Lemma 11.3 *Set difference reduces to division:*

$$\|A - B\| = \frac{\|A\|}{\|A\| \gcd \|B\|.}$$

Proof. By extension of the previous result. ■

Lemma 11.4 *Membership reduces to divisibility:*

$$u \in A \textit{ iff } P(u) \mid G(A)$$

Proof.

$$\begin{aligned} u \in A &\text{ iff } 1 = \chi_A(u) \\ &\text{ iff } P(u) = P(u)^{\chi_A(u)} \\ &\text{ iff } P(u) \mid P(u)^{\chi_A(u)} \\ &\text{ iff } P(u) \mid \prod_{u \in \mathbb{U}} P(u)^{\chi_A(u)} \\ &\text{ iff } P(u) \mid G(A). \end{aligned}$$

■

Lemma 11.5 *Inclusion reduces to divisibility:*

$$A \subseteq B \textit{ iff } G(A) \mid G(B).$$

Proof. By an argument similar to the proof for membership. ■

Lemma 11.6 *Insertion reduces to divisibility and multiplication:*

$$\|A \cup \{u\}\| = \begin{cases} \|A\| & P(u) \mid \|A\| \\ \|A\| \times P(u) & \textit{otherwise.} \end{cases}$$

Proof. By cases in $u \in A$, $u \notin A$. ■

Lemma 11.7 *Deletion reduces to divisibility and division:*

$$\|A - \{u\}\| = \begin{cases} \|A\|/P(u) & P(u) \mid \|A\| \\ \|A\| & \textit{otherwise.} \end{cases}$$

Proof. By cases in $u \in A$, $u \notin A$. ■

11.4.3 Space-efficiency of Gödel hashes on sets

I can derive upper bounds on the size of a Gödel hash for a set. Let U be the cardinality of the universe, $U = |\mathbb{U}|$. Using the prime number theorem, I can approximate the value of the prime p_U:

$$p_U \approx U \ln(U)$$

From this, I can approximate the number of bits required to represent p_U:

$$E_{\mathbb{U}} = \textit{size}(p_U) \leq \lceil \lg(U \ln(U)) \rceil$$

When an n-bit number and an m-bit number are multiplied, the result is an (at most) $(n+m)$-bit number. From this, I can bound the bit size of a hash with k elements:

$$\textit{size}(\|\{u_1, \ldots, u_k\}\|) \leq k \times \lceil \lg(U \ln(U)) \rceil.$$

One immediate observation, is that on 32-bit hardware, until the universe of discourse exceeds a cardinality of 193,635,250 (roughly 2^{27}), each word will hold more than one element. On 64-bit hardware, the equivalent threshold for the universe of discourse is a cardinality of 415,828,534,307,634,000 (roughly 2^{58}).

I can also predict the minimum number of elements which will fit in a single word of W bits as a function of the size of the universe:

$$k_{\min}(U) = \frac{W}{\lceil \lg(U \ln(U)) \rceil}.$$

Example 2 *For example, with a universe of* 2^{10} *elements, a 64-bit machine would be able to roughly fit five elements in each word. Translated to a more concrete example, if 0CFA [38] were to compute the flow sets (where each flow set contains the lambda terms that might flow to an expression) for a program with a thousand functions, most flow sets (which tend to be very sparse) would fit in single word—a single register. Using the bitmap formulation, each flow set would consume 16 words of contiguous memory; using the traditional bucket-based hash set, each flow set would consume about ten words; and a balanced-tree sorted set implementation would consume roughly 15 words of (noncontiguous) memory per flow set.*

The average case is slightly better. For the average case, I assume that the elements of a set are uniformly distributed throughout the universe. In this case, the average size (in bits) of a set with k elements will be:

$$\frac{k}{(U-1)}\sum_{i=2}^{U}\lceil \lg(i\ln(i))\rceil.$$

Example 3 *For instance, on average, on 64-bit hardware, each word will hold a little over five elements on average assuming a universe with* 2^{10} *elements. Or, in a universe with* 2^{16} *elements, each word will now hold about three elements on average.*

11.4.3.1 Optimizing the prime map for space usage

If we know *a priori* the probability distribution of elements in the universe, then we can assign primes so as to minimize the number of bits per set. If the probability of an element u appearing in a set is $f(u)$, then we can construct the vector $\vec{u}^\star \in \mathbb{U}^*$ in which elements are sorted according to decreasing frequency:

$$f(u_i^\star) \geq f(u_{i+1}^\star).$$

The optimal prime map is $P^\star : \mathbb{U} \to \mathbb{P}$:

$$P^\star(u_i^\star) = p_i.$$

The expected size (in bits) of a random set is:

$$\sum_{u\in\mathbb{U}} f(u)\lceil \lg(P^\star(u))\rceil,$$

which leads to a space-optimality result:

Theorem 11.1 (Space optimality) *The prime map* $P^\star$ *minimizes the expected bit-size of a random set.*

Proof. Straightforward. (By contradiction.) ■

Example 4 *If the universe has infinite size, but its elements are distributed according to a geometric distribution with parameter $r = 1/2$, then the average set size will be roughly six bits. (Intuition: Half of all the elements will be two, which adds only one bit to a set; a quarter of all elements will be three, which adds only two bits; an eighth of all elements will be 5,* etc.*)*

11.4.4 Time efficiency of operations on Gödel hashes for sets

As it turns out, some operations on Gödel hashes have slightly worse asymptotic complexity than other data structures for sets. I will need to perform a more detailed accounting of their cost with respect to modern hardware, chiefly with respect to the CPU word size, to unearth their pragmatics. To make the constant factors stand out, I will assume 64-bit hardware. I will also assume that the underlying implementation of arbitrary-precision natural number is an array of unsigned integers (64-bit words).

I discuss the cost of operations on two sets, A and B. I assume the universe contains no more than 2^{58} elements so that a single element hash can fit into a machine word. Such a universe size is sufficient in practice. Let m be the sum of the cardinality of these sets: $m = |A| + |B|$. I will refer to the maximum number of bits in the Gödel hashes of these sets: $n = mE_{\mathbb{U}}$.

- Intersection needs to compute greatest common divisor. The Euclidian algorithm requires up to $n/64$ modulo operations on two multiple precision naturals (the Gödel hashes of two sets, respectively), each of which has $O(n^2)$ time complexity. Thus, the complexity of intersection is cubic: $O(n^3)$. However, the modulo operation is performed in chunks of the word-size , which means the $O(n^2)$ complexity has a hidden $1/64^2$ constant speedup factor. Considering the other $1/64$ constant in the number of times of the modulo operations, the cubic intersection complexity actually has a $1/64^3$ constant speedup factor! On a pipelined 64-bit CPU, the back-of-the-envelope cost of the computation is up to $\left\lceil \frac{n^3}{64^3} \right\rceil = \left\lceil \frac{n^3}{262144} \right\rceil$ clock cycles!
- The cost for union is the same as intersection.
- The cost for set difference is the same as intersection, plus a division operation with two multiple precision numbers.
- The cost for element insertion is a modulo operation and a multiplication operation, both of which operate on a multiple precision number (the Gödel hash of a set) with

a word-size (the Gödel hash of an element) divisor or multiplicand, so the total time complexity is $O(n)$, with a 1/64 constant speedup factor.

- The cost for element deletion is: a modulo and a division, both of which operate on a multiple precision number (the Gödel hash of a set) with a word-size (the Gödel hash of an element) divisor. Same as insertion, it is $O(n)$ time complexity with a 1/64 constant speedup factor.
- The complexity of membership-testing is quite efficient too: one modulo operation on a multiple precision number (the Gödel hash of a set) with a word-size (the Gödel hash of an element) divisor, which costs $O(n)$ time with a 1/64 constant speedup factor.
- The cost of subset inclusion testing is a modulo operation with two multiple precision numbers (the Gödel hashes of two sets, respectively), which is $O(n^2)$ time complexity, with a 1/4096 constant speedup factor.
- The complexity of set enumeration is equivalent to integer factorization, which is believed to be intractable[3].

In practice, multiple precision arithmetic can be further accelerated via SIMD instructions such as streaming SIMD extensions (SSE) and advanced vector extensions (AVX) [41]. These instructions can operate on 256 or even 512 bits data with a single instruction. Hence, all the operations above can benefit from a much larger constant factor. What is more, compared with other common set implementations such as tree-based or bucket-based hash sets, the natural implementation of Gödel hashes as a small array of unsigned integers, which can minimize cache misses, is better-suited to modern hardware for cache efficiency. The empirical evaluation in Section 13.2.2 on the GNU (the name GNU is a recursive acronym for GNU's Not Unix!) GNU multiple precision (GMP) based Gödel hashes validates both of these.

11.5 Partial orders

The ability of Gödel hashes to accelerate testing for subsumption under a partial order, *i.e.*, whether $x \sqsubseteq y$, is perhaps their greatest strength. Partially ordered sets (posets) play an important role in fields such as static analysis and artificial intelligence. In static analysis in particular, subsumption testing in large, complex lattices can easily consume the bulk of the runtime for an analysis. (See Chapter 13, Section 13.2.4 for experimental results in accelerating static analysis with Gödel hashing.)

[3] Except that primes are sufficiently small in most cases.

I am able to provide a structurally recursive condition for when a partially ordered set has an order-perserving Gödel hash. Specifically, given a poset $(S, \sqsubseteq)$, I can formulate Gödel hash analogs of join ($\sqcup$), meet ($\sqcap$) and subsumption ($\sqsubseteq$) if the poset S has a *prime basis*. A poset has a prime basis if every (nonbottom) element has a unique decomposition as the least upper bound of a finite number of basis elements.

Definition 11.2 *For a poset $(S, \sqsubseteq)$, the set $B \subseteq S$ is a* ***prime basis*** *if:*

$$C_1 \subseteq B \text{ and } C_2 \subseteq B,$$

and

$$\bigsqcup C_1 = \bigsqcup C_2,$$

implies

$$C_1 = C_2; \text{ and}$$

for any element $s \neq \bot \in S$, there exists a set $C \subseteq B$ such that:

$$s = \bigsqcup C.$$

I will call a partially ordered set that has a prime basis a **factorable** poset. A factorable poset does not need to have a weakest element, but if it does, the weakest element (denoted $\bot$) is not in the prime basis. (This is analogous to excluding one from the set of prime numbers.)

Definition 11.3 *The function $G_S : S \to \mathbb{N}$ computes the* ***order-preserving Gödel hash of a partially ordered set*** *$(S, \sqsubseteq)$ with prime basis B under the prime map $P_B : B \to \mathbb{P}$:*

$$G_S\,(b_1 \sqcup \cdots \sqcup b_n) = P_B(b_1) \times \cdots \times P_B(b_n).$$

11.5.1 Operations on factorable partial orders

As with previous Gödel hashes, common operations and relations reduce to natural arithmetic.

Lemma 11.8 *Join reduces to the least common multiple:*

$$\|s_1 \sqcup s_2\| = \|s_1\| \operatorname{lcm} \|s_2\|.$$

Proof. Factorability allows us to construct "characteristic functions" on the prime basis of partial orders, where $\chi_s : B \to \{0, 1\}$:

$$\chi_s(b) = \begin{cases} 1 & b \sqsubseteq s \\ 0 & b \not\sqsubseteq s. \end{cases}$$

Clearly, $\chi_{s_1 \sqcup s_2}(b) = \max(\chi_{s_1}(b), \chi_{s_2}(b))$. The proof is analogous for union over sets. ■

Lemma 11.9 *Meet operation reduces to the greatest common divisor:*

$$\|s_1 \sqcap s_2\| = \|s_1\| \text{ gcd } \|s_2\|.$$

Proof. By an argument similar to the previous proof. ■

Lemma 11.10 *Subsumption reduces to divisibility:*

$$s_1 \sqsubseteq s_2 \textit{ iff } G(s_1) \mid G(s_2).$$

Proof. By an argument analogous to the subset-inclusion test for Gödel hashes on sets. ■

Warning: My definition of prime basis does not ensure that a partially ordered set defines a join (nor a meet) for any two elements. As a result, there are cases where $s_1 \sqcup s_2$ will not exist, but of course, the reduction to Gödel hashes will still assign it a number, and there is no way for the Gödel hash to know that this element does not exist in the partial order. Thus, the Gödel hash reduction for partial orders is only sound under join for join-semilattices, and under meet for meet-semilattices. It is therefore advisable to promote a partial order to a lattice before working with its Gödel hash encoding to ensure soundness. (This is not an issue for application domains such as static analysis.)

11.5.2 Recursively constructed factorable posets

I can show that the standard set-construction operators preserve factorability.

Factorable flat orders: a poset $(S, \sqsubseteq)$ whose order is flat is trivially factorable: its prime basis is the set S.

Factorable power sets: a partially ordered power set $(\mathcal{P}(S), \subseteq)$ where the order is inclusion is easily factorable: its prime basis, $B_{\mathcal{P}(S)}$, consists of the singleton sets over S: $B_{\mathcal{P}(S)} = \{\{s\} : s \in S\}$

Products of factorable posets: the Cartesian product of factorable partial orders is itself factorable under its product ordering. Let $(A_1, \sqsubseteq_1)$ and $(A_2, \sqsubseteq_2)$ be factorable posets with prime bases B_1 and B_2, respectively. Then the poset $(A_1 \times A_2, \sqsubseteq_{A_1 \times A_2})$ is defined so that:

$$(a_1, a_2) \sqsubseteq_{A_1 \times A_2} (a_1', a_2') \text{ iff } a_1 \sqsubseteq_1 a_1' \text{ and } a_2 \sqsubseteq_2 a_2'.$$

The prime basis for the product, $B_{A_1 \times A_2}$, is the product of the prime bases: $B_{A_1 \times A_2} = B_1 \times B_2$.

Disjoint unions of factorable posets: if two posets $(A_1, \sqsubseteq_1)$ and $(A_2, \sqsubseteq_2)$ have prime bases B_1 and B_2, then the prime basis for the natural ordering of the disjoint sum $A_1 + A_2$ is the disjoint sum of the prime bases: $B_{A_1 + A_2} = B_1 + B_2$.

Function spaces into factorable partial orders: the natural partial ordering of functions leads to a factorable space of partially ordered functions when the range of the function space is factorable. That is, if $(Y, \sqsubseteq_Y)$ has prime basis B_y, then a space of *finite* functions $(X \to Y, \sqsubseteq_{X \to Y})$ is also factorable under the natural ordering:

$$f \sqsubseteq_{X \to Y} g \text{ iff for each } x \in \mathit{dom}(f) : f(x) \sqsubseteq_Y g(x).$$

The prime basis for this function space, $B_{X \to Y}$, is the set of functions that map just one element of x into a prime basis element of the set Y:

$$B_{X \to Y} = \{\bot_{X \to Y}[x \mapsto b] : x \in X, b \in B_Y\},$$

where the bottom function $\bot_{X \to Y}$ maps every element to the bottom of Y: $\bot_{X \to Y}(x) = \bot_Y$, unless the poset Y has no bottom, in which case $\bot_{X \to Y}$, is the everywhere undefined function: $\lambda x.\mathit{undefined}$. Partial function spaces are identical except that the prime basis elements do not extend the bottom map:

$$B_{X \rightharpoonup Y} = \{[x \mapsto b] : x \in X, b \in B_Y\}.$$

11.5.3 Function spaces into countable total orders

While factorability is a sufficient condition for Gödel hashing, there are partial orders which are not factorable, yet which have an order-preserving Gödel hash. An important instance of this is a function space that maps into a countable total order. (In static analysis, such posets are used for must-alias and environment analysis [17, 15].) If $(Y, \leq)$ is a totally ordered set, then the space of *finite*, partial functions $X \rightharpoonup Y$ has the natural partial order $(\sqsubseteq_{X \rightharpoonup Y})$: $f \sqsubseteq_{X \rightharpoonup Y} g$ iff $f(x) \leq g(x)$ for all $x \in X$.

Because Y is a countable total order, there exists an order-preserving measure function $M : Y \to \mathbb{N}$.

Definition 11.4 *Given a totally ordered set $(Y, \leq)$, the* ***order-preserving Gödel hash of the function*** *$f : X \rightharpoonup Y$ under the prime map $P : X \to \mathbb{P}$ is computed by the function $G : (X \rightharpoonup Y) \to \mathbb{N}$:*

$$G(f) = \|f\| = \prod_{x \in \mathit{dom}(f)} P(x)^{M(f(x))}.$$

Under this definition, I have the usual reductions:

Lemma 11.11 *Join reduces to the least common multiple:*

$$\|f \sqcup g\| = \|f\| \operatorname{lcm} \|g\|,$$

Proof. The argument proceeds by constructing a multiset-like characteristic function $\chi_f : X \to \mathbb{N}$:

$$\chi_f(x) = M(f(x)).$$

The rest of the argument is analogous to union on multisets. ■

Lemma 11.12 *Meet reduces to the greatest common divisor:*

$$\|f \sqcap g\| = \|f\| \text{ gcd } \|g\|,$$

Proof. By an argument similar to the previous proof. ■

Lemma 11.13 *Subsumption reduces to divisibility:*

$$f \sqsubseteq g \textit{ iff } G(f) \mid G(g).$$

Proof. By an argument analogous to that of subset-inclusion testing for multisets. ■

A Gödel encoding for sequences is actually required in the proofs of incompleteness.

11.6 Computing primes and prime maps

Gödel hashes rely on being able to generate prime maps. Constructing efficient prime maps requires an efficient method for generating the ith prime number. In functional programming, a global, lazy, internally memoizing stream of prime numbers is convenient, particularly when multiple structures in the program will require their own prime maps. This is the approach that my implementation uses (in the analyzer implementation in Chapter 13, Section 13.2.4). There are pragmatic methods that use a sieve to generate primes deterministically [42]. However, probabilistic primality tests [43, 44, 45] are more efficient, arbitrarily reliable and require no storage of prior primes. The probabilistic tests are particularly fast on word-sized primes. Also, if primes are allocated in an on-demand fashion, word-sized primes are all that are likely to be needed.

CHAPTER 12

IMPLEMENTATION OF A HUMAN-IN-THE-LOOP STATIC ANALYZER FOR MALWARE DETECTION

This chapter describes the implementation of a static malware detection system with a human in the loop for Android apps. The system integrates the techniques presented in the previous chapters with principled soundness to detect common classes of malware in Android apps in a highly precise fashion yet with good-enough performance. It first describes principled soundness in Section 12.1. Then Section 12.2 presents the system architecture and some implementation details.

12.1 Principled soundness

Theoretically, soundness in static analyses usually means the capability to model *all* language's features and approximate *every* concrete executions. Unfortunately, there is no such thing in reality to date, because *complete* soundness in this sense is a very hard problem, theoretically and practically.

In fact, it is not always possible to demand the complete soundness in practice, because (1) every research is constrained to some resources and targeting for a specific goal, and as such it is infeasible to *"do it all"* once; (2) clients of static analysis have resource constraints (such as efficiency) or targets for some specific features, rather than achieve all. However, this does not mean I abandon the pursuit of *soundness*.

Therefore, I describe principled soundness: it is a well-controlled, best-efforts soundness that satisfies the clients, given available resources. principled soundness particularly fits in the malware detection scenario.

The essence of principled soundness is: it is justifiable. It is reasonable to discard soundness in many cases, as summarized below:

- Library code as well as platform code from some large "trustworthy" entities such as Google, *etc.*, for the reasons: (1) the semantics are reasonably defined; (2) it has low risk or probability of malicious code, because maliciousness is “mostly” embedded in application code (100% of all the cases in my experience to date).

- Thread interleavings from concurrent constructs are not explicitly claimed in this work, since concurrency-incurred control-flows and data-flows are very unreliable in being produced or exploited in the context of security vulnerability identification [5] (Chapter 9, Section 9.2).
- Features of no current major concerns: dynamic loading, native codes, reflection, cryptography, *etc.* However, it does not mean discarding them completely. I will need to deal with them given sufficient resources.
- The benefit of analyzing some features does not pay off. This can happen either when complicated features complicate the implementation of the static analyzer or they incur scalability issues due to integration into inefficient legacy analysis model. For example, abstract interpretation of string is not a trivial feature to add in the analyzer. In addition, it can induce high costs. Normally, analyzer provides the option to turn on/off string analysis to focus on the analysis on the property of interest. This can be the same case for analyzing exceptions, and possibly other features.

12.2 *AnaDroid*: the static malware analyzer with a human in the loop

The software artifact is called *AnaDroid*. The analyzer is mainly implemented in Scala in roughly 19K LOC, with subproject implemented in Java and Python. The source code is available at https://github.com/shuyingliang/pushdownoo.

Figure 12.1 briefly sketches the software architecture as well as the work flow. Section 12.2.1 illustrates how *AnaDroid* enables human-in-the-loop with principled soundness.

- ***AnaDroid* frontend:** *AnaDroid* consumes off-the-shelf Android application packages (files with suffix .apk). In Figure 12.1 *JDexSExp* extracts the .dex file by invoking apktool [46] and then disassembles binaries and generates an S-expression IR based on the smali [47] format. The *Parser* parses the IR in S-Expression format and generates Abstract Syntax Tree in the format that is defined in Chapter 2 and Section 3.2.
- ***AnaDroid* analysis backend:** The static malware detection system is built on the principles illustrated in the previous chapters and sections. The pushdown exception-flow analysis (Chapter 6) serves as the foundational platform. Entry point saturation (Chapter 9) is embedded in the platform. The other components, abstract garbage collection and its enhanced version (Chapter 7), static taint-flow analysis (Chapter 10) and Gödel hashing for efficiency (Chapter 11) are “plugged” into the platform. The

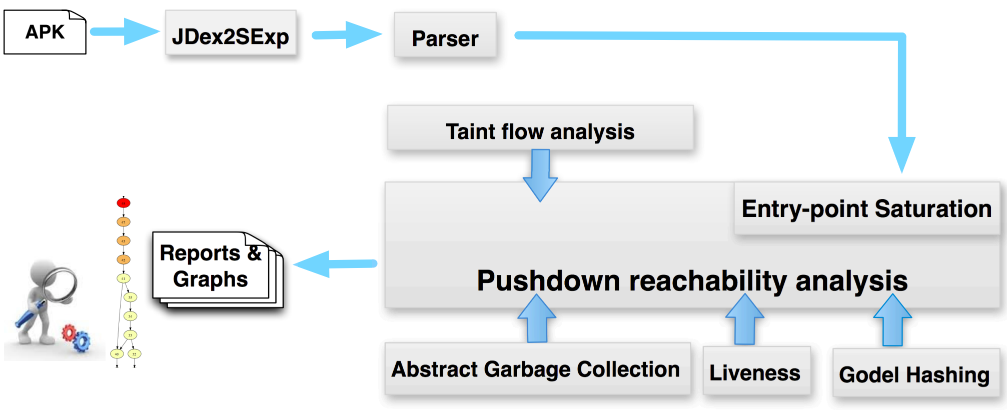

Figure 12.1: Static malware detection with a human in the loop.

following section describes how to involve human analysts in the loop with principled soundness.

12.2.1 Human-in-the-loop analysis

AnaDroid enables human-in-the-loop analysis, by providing a rich user interface for an analyst to configure the analyzer, *i.e.* setting the k value, configuring abstract garbage collection to be used or not, specifying predicates over the state space, *etc.* to trade off precision and performance.

- **Semantic predicates:** To assist analysts, I provide a library of predicates for common patterns. For example, *AnaDroid* allows analysts to specify packages and application programming interfaces (APIs) of interest in regular expression pattern, so that states that are matched will be highlighted with customized color.
- **Visualization of analysis results:** *AnaDroid* generates various reports and state graphs to visualize the analysis results[1]. The following three reports are included: (1) least permissions presents which permissions are requested by an app and which permissions are inferred by *AnaDroid*, reporting whether the app requests more permissions than it actually uses. (2) The information flow report presents triggers (mainly user interface (UI) triggers) and tainted paths that lead from sources to sinks, with contexts such as class files, method names, and line numbers. (3) The heat map report shows rough profiling results of the analyzer, which can be used to help an analyst understand where the analysis has focused its efforts and might indicate where an app developer has attempted to hide malicious behavior. (4) Analysis graphs

[1] Sophisticated technologies to visualize analysis data deserve an orthogonal and decent study, which is not in the scope of the dissertation work.

are presented with an scalable vector graphics (SVG) formatted file as a reachable control-flow graph. It highlights suspicious source and sink states, as well as showing tainted paths between them. In addition, an analyst can click on any state node in the graph for detailed inspection of the abstract execution at this point in the graph.

- **The flags for principled soundness in malware detection:** In addition to the pure analysis configurations, *AnaDroid* provides a list of flags to enable analysts search analysis results of properties of interest. The feature controls *principled soundness* (also precision and efficiency indirectly). The flags are listed in Table 12.1. They are divided into three categories: (1) language features; (2) property of interests, which is related to Android specific functionalities and APIs; (3) analysis truncate, that is to decide whether or not the analysis should proceed after some specified time or number of states. By default, the flags are turned off.

Table 12.1: Flags for principled soundness.

Categories	Language features	Property of Interests	Analysis truncate
Flags	–exceptions, –strings	GPS, IDs, network, display, webview, contact, sensor, camera, account, SMS, media, picture, fileSystem	–states[n], –time[n]

CHAPTER 13

EVALUATION

This chapter presents evaluations on the analysis precision and performance of the analysis techniques and the impact on client security analysis for Android apps. The experiments are divided in three categories. The first category of evaluation (Section 13.1) is on analysis precision and performance of pushdown exception-flow analysis, abstract garbage collection and its enhanced version. The second category of evaluation (Section 13.2) is on space usage and performance of the data structure Gödel hashing sets and its application in accelerating static analysis (Section 13.2.4). The last category of evaluation (Section 13.3) is for client security analysis, to demonstrate impact of analysis precision in helping human analysts to identify malware.

13.1 Evaluation on analysis precision and performance

To evaluate the effectiveness of the pushdown exception-flow analysis in analyzing object-oriented programs, I compare my analysis with one of the well-known finite-state based static analysis frameworks—WALA [48].

WALA adopts co-analysis of control-flow and data-flow analyses, performing call-graph construction and pointer analysis together, by propagating pointer information on the constructed CFG. The framework provides several context-sensitivities [49], including rapid type analysis (RTA), 0-CFA, 0-1-CFA (0-CFA with 1-object sensitivity), vanilla-0-1-CFA (an unoptimized version of 0-1-CFA), and analysis with additional disambiguation of container elements 0-container and 0-1-container. In particular, the 0-1-CFA enables several optimizations for string and thrown objects[1], *etc.* The 0-1-container policy extends the 0-1-CFA with unlimited object-sensitivity for collection objects, which is the most precise default option. My evaluation uses the 0-1-container as the baseline.

[1] http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis

To make the comparison more compelling, I conduct experiments on the DaCapo [50] benchmarks. The suite of benchmarks has been widely accepted and used in the computing community to evaluate Java programs. In addition, it has much larger scale code bases to analyze than ordinary mobile applications presented in the Google market. This allows a more realistic workload to stress-test the analysis.

Since the analyzer works directly on Dalvik bytecode, which is compiled from Java programs in Dalvik virtual machine (DVM), I have successfully compiled 10 out of 11 Java applications in the DaCapo benchmark (v.2006-10.MR2) in DVM with minor changes in the source code (mainly `enum` is changed to another name because `enum` is a keyword in JRE 1.5 or later). I encapsulate the `main` method of each benchmark in the entry point `onCreate` of a class of type `Activity`. These benchmarks are compiled using the built-in tool `dx` in the Android software development kit (SDK). Some graphics user interface (GUI) class references (especially `awt`) in Java programs are resolved by including `rt.jar` in the Android class path. To avoid name space conflicts in packages and classes, I use `jarjar`[2] to repackage some Java standard libraries that are re-implemented in Android. The only Java program that is not ported is `eclipse`, which involves substantial conflicts in Java GUI classes (`awt, swing, swt`). I believe the other ten programs suffice for my purpose.

13.1.1 Metrics for precision

The basis for comparison in precision is the average cardinality of a points-to set, which computes the average number of abstract objects (including exception objects) for pointers that are collected into a single representative in the abstraction. In my evaluation, it has two forms: `VarPointsTo` and `Throws`. `VarPointsTo` refers to the average cardinality of the points-to set for nonexception abstract objects, and `Throws` refers to exception objects specifically. (In Table 13.1, I normalized the two metrics computed in WALA, relative to that in my analysis.)

I adopt this metric because it has been used widely in the literature [37, 7, 51] to measure precision for object-oriented programs. The metric reflects analysis precision by recognizing that the more objects are conflated for a variable, the less precise the analysis. When this metric is a large value, it indicates a negative impact on normal control-flow analysis because it means that virtual method resolution needs to dynamically dispatch to more than one function causing spurious control-flow paths. This same reasoning applies for exception-flow analysis. (The more subtle relationships have been illustrated in Section 1.2).

[2] https://code.google.com/p/jarjar

Table 13.1: Precision comparison. Values in columns `Nodes`, `Edges` and `Methods` are ratios of the number of nodes, edges and methods reached in our analysis, relative to the ones in WALA, respectively. Values in columns `VarPointsTo`* and `Throws`* are ratios of average cardinality of general point-to set and exception points-to set in WALA, relative to the ones in our analysis, respectively. I did not list the results for the benchmark `jython`, since I got OutofMemory error when running WALA after roughly 1 hour, even though I increased the stack and heap space in JVM with the options: `Xms10g -Xss5g -Xmx10g -XX:MaxPermSize=2048m`. `pdxfa+1obj` exists to show the contribution of precision for `pdxfa` and `eagc`, respectively. The table shows that the pushdown exception-flow analysis with enhanced abstract garbage collection `pdxfa+eagc` outperforms finite-state analysis in WALA in precision by 4.5X-11X for `Throws` and up to 7X for general points-to information `VarPointsTo`.

Benchmark	LOC	Opts	Nodes	Edges	Methods	VarPointsTo*	Throws*
antlr	35,000	pdxfa+1obj	4.1x	1.3x	1.2x	1.5x	2.8x
		pdxfa + eagc	3.9x	1x	1x	3x	4.6x
bloat	70,344	pdxfa+1obj	1.9x	1.4x	2.4x	3.3x	2.4x
		pdxfa + eagc	1.2x	1.3x	1.1x	6.3x	6x
chart	217,788	pdxfa+1obj	2.3x	1.3x	1.1x	2x	2.3x
		pdxfa + eagc	2.1x	1.1x	1.2x	6x	4.5x
fop	184,386	pdxfa+1obj	2.1x	1.4x	1.1x	4.2x	5.5x
		pdxfa + eagc	1.9x	1.3x	1.5x	7.3x	11x
hsqldb	155,591	pdxfa+1obj	8.9x	4.4x	3.4x	1x	2.3x
		pdxfa + eagc	5.3x	2.7x	3.3x	3x	4.5x
luindex	38,221	pdxfa+1obj	1.9x	1.9x	1.8x	1x	1.6x
		pdxfa + eagc	3.5x	1.7x	1.2x	1.5x	4x
lusearch	87,000	pdxfa+1obj	1.5x	1.6x	1.6x	1.6x	2.3x
		pdxfa + eagc	1x	1.5x	1.4x	2.5x	4.5x
pmd	55,000	pdxfa+1obj	1.8x	1.3x	1.5x	2.2x	5.2x
		pdxfa + eagc	1.5x	1.1 x	1x	3.7x	7.7x
xalan	159,026	pdxfa+1obj	1.9x	1.3x	1.7x	2.8x	6.2x
		pdxfa + eagc	1.4x	1.2x	1.3x	3.7x	10.3x

Following WALA's heap model[3], I compute the same metric in my pushdown framework.

In addition to the `Throws` metric, the work [37] proposed using exception-catcher links (E-C links) to reflect the precision of handling exceptional flows. I compute the metric in my analysis framework, which is within the range of 1-3 across the DaCapo benchmarks. Because WALA directly computes the catchers intraprocedurally, I do not compute and report the comparison ratio as I do for `VarPointsTo` and `Throws`.

13.1.2 Results

Table 13.1 shows that the pushdown exception-flow analysis with enhanced abstract garbage collection `pdxfa+eagc` outperforms finite-state context-sensitive analysis (repre-

[3] In WALA, the pointer-to relation is computed from `PointerKey` to a set of `InstanceKey`s, where a `PointerKey` may represent a local variable, a static field, or an instance field of objects from a particular allocation site, and an `InstanceKey` may represent all objects of a particular type, all objects from a particular allocation site, all objects from a particular allocation site in a particular context, or other variants.

sented by WALA) with a precision of 4.5-11 times for `Throws` and up to 7 times for general points-to information `VarPointsTo`.

`Nodes` and `Edges` are control-flow graph information. `Methods` denotes the analyzed methods. The values in these columns in Table 13.1 are normalized relative to those reported by WALA 0-1-contain analysis. As is shown in Table 13.1, the pruned, pushdown analysis technique (`pdxfa+eagc`) generally explores more edges and nodes, and explores up to 3.4 times more methods[4].

To evaluate the contribution of each aspect (pushdown exception-flow analysis and enhanced abstract garbage collection) to precision improvement, when comparing with WALA, I also conduct an additional experiment with only the pushdown exception-flow analysis with 1-object sensitivity (as WALA 0-1-container does), denoted as the option `pdxfa+1obj`. The result shows that the `pdxfa` improves the precision more than enhanced abstract garbage collection does.

13.1.3 Analysis time

For completeness, I also report an analysis time comparison. Table 13.2 is the ratio of my analysis time to that of WALA.

WALA reports less analysis time than my analysis. This is not surprising. First, my analysis is derived from the polynomial complexity algorithm in [16, 18]. Even with enhanced garbage collection, it only reduces the complexity by a constant factor. Second, WALA has been significantly optimized by the IBM research lab, particularly with underlying Java (collection) libraries rewritten specifically for its framework. My implementation is based on Scala's default data structure and is not optimized. Last but not least, the analysis time is reasonably acceptable, given the high precision that my analysis technique can provide. For example, for the largest benchmark `chart`, the unoptimized analyzer takes roughly 13 minutes.

13.2 Evaluation on Gödel hashing

There are two key questions to answer with experimentation:

1. How big do Gödel hashes get?
2. How fast are operations on Gödel hashes?

The short answer to the question of size is that they are roughly tens of times less than the size of the structure from whence they came for relatively sparse data. With respect to

[4] WALA filters out some common library methods (*e.g.*, I/O API), which can be specified in "AnalysisScope".

Table 13.2: Analysis time.

benchmark	**antlr**	**bloat**	**chart**	**fop**	**hsqldb**	**luindex**	**lusearch**	**pmd**	**xalan**
ratio	11.7x	5.9x	14.9x	10.9x	5.4x	2.7x	8.7x	9.2x	5x

speed, the short answer is that the critical operations of inclusion and equality are orders of magnitude faster at all densities.

I will focus my efforts on Gödel hashes of sets, since these form the basis for the other techniques. Let the set $\mathbb{U}$ be the universe and let $U = |\mathbb{U}|$ be the size of the universe.

13.2.1 Measuring hash size

Figure 13.1 renders the normalized sizes of the following standard data structures, relative to that of the predicted worst-case for Gödel hashes set: (1) the array-backed set; (2) the tree-backed set; (3) the traditional hash-backed set; and (4) the bitmap-backed set.

The varied parameters are (1) the size of the universe, U, and (2) the density of the set relative to the size of the universe, ρ. For instance, with a density of 0.1 and a universe of 10,000 elements, the set size under consideration is $0.1 \times 10,000 = 1,000$.

I compute the exact array size, without consideration of resizing strategy, which is commonly employed in most program languages' standard libraries for better performance

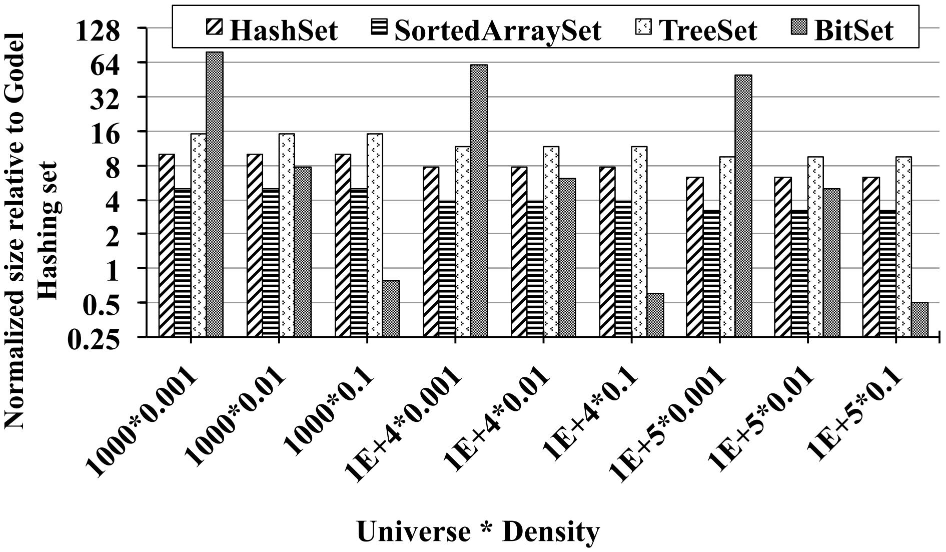

Figure 13.1: Normalized size of traditional hash sets, sorted-array sets, sorted-balanced tree sets and bitmap sets, relative to the size of the predicted worst case of Gödel hashing sets. The (logarithmic scale) vertical axis is the normalized size. The horizontal axis is denoted as $U * \rho$: U is the size of the universe, ρ is the density of the set as a fraction of the universe. The worst case of Gödel hash dominates for compactness—by up to tens of times smaller than that of the common data structures.

(*e.g.*, Java's `ArrayList` increases its capacity by a factor of 1.5). Also, I only compute the elements' size based on a very space-efficient bucket-based hashset implementation (C++'s `unordered_set`), the bucket array's size is ignored.

Even so, Gödel hashes are substantially smaller (tens of times smaller) than all of the other data structures, except the dense bitmap-backed sets with $\rho = 0.1$. Even though the bitmap set is very space-efficient to represent dense data, when it goes to the sparse data with small ρ, the size of bitmap set can be up to 78.4 times larger than that of Gödel hashes!

I focus on sparse sets because higher-order program analysis tends to deal with highly sparse flow sets. For instance, in context-sensitive analysis, bitmap-backed sets could be exponential in the size of the program, whereas the median flow set in practice has size 2.

13.2.2 Measuring speed

In Table 13.3, I measured the slow down ratio of average run time of each single set operation on sorted tree sets, sorted array sets, and bitmap sets relative to that of Gödel hashes. For hashes, fast equality is a critical operation. For Gödel hashes, fast subsumption (subset) is also critical for candidate applications. Table 13.3 shows that the critical hash operations on Gödel hashes of sets can be up to hundreds of times faster! Remarkably, the performance of critical operations on bitmap sets degrades rapidly (by an order of magnitude) when the sets become sparse (when ρ decreases), even though they can be relatively more efficient than the other three standard data structures in dense sets (with $\rho = 0.1$). The performance advantage of Gödel hashing sets on sparse sets (in addition to the size advantage validated in Table 13.1) fits extremely well over other data structures in the case of program analysis, especially for higher-order programs.

13.2.3 Implementation details

Gödel hashes are implemented in C++ using GNU GMP for big integer arithmetic operations. GMP is highly optimized on modern CPUs to operate at very long data types with a single instruction. Instruction sets such as SSE and AVX can do 256 bits or even 512 bits data arithmetic operations [41]. Sorted tree set uses C++ `std::set`, which is red-black tree based. Sorted array set uses C++ `std::vector` to store data, and uses `std`'s binary search and set algorithms for correspondent set operations. Hash set uses `std::unordered_set` that is a bucket-based hash set. Bit set uses C++ `std::bitset`, which is implemented with an array of integers. The evaluation program runs each operation 1,000 times on sets that contain $U * \rho$ number of elements. These elements are randomly fetched from the prime universe U. The evaluation is conducted on a PC with a six-

Table 13.3: Slow down ratio of average run time of each single set operation on sorted tree sets (ST), sorted array sets (SA), hash sets (HS) and bitmap sets (BS), relative to that of Gödel hashes. For the critical hash operations of equality and subsumption, operations on Gödel hashes of sets are up to hundreds of times faster.

U	**density**		$\subseteq$	$=$	$\cup$	$-$	$\cap$	$\in$	deletion	insertion
5,000	0.001	ST	2.667	2.333	1.459	1.179	0.56	1	2.667	3.333
		SA	1.333	1.333	0.324	0.393	0.36	0.75	1	2.333
		HS	4	4.333	1.757	1.607	0.72	1	1.667	2.333
		BS	36.333	35.333	2.216	2.929	3.32	0.25	0.333	0.333
	0.01	ST	17.75	5.667	1.067	0.888	0.372	1.333	3.2	2.25
		SA	6.625	2.333	0.098	0.105	0.072	0.833	1.6	2.5
		HS	14	5.333	1.318	0.82	0.287	0.667	1.2	1
		BS	13.75	23.333	0.129	0.147	0.149	0.333	0.2	0.25
	0.1	ST	89.147	285.455	0.843	0.789	0.399	1.278	1.962	2
		SA	40.941	112.818	0.075	0.07	0.049	0.556	1.423	4.32
		HS	31.294	94.182	0.921	0.551	0.201	1.444	0.923	0.72
		BS	3.235	9.727	0.01	0.011	0.011	0.111	0.038	0.08
10,000	0.001	ST	4.8	9	1.165	1.056	0.395	1.25	3.667	2.333
		SA	1.8	3.5	0.165	0.222	0.158	0.75	1.333	2
		HS	4	11	1.66	1.25	0.421	0.75	2	1.167
		BS	49.4	117.5	1.641	2.319	2.197	0.5	0.667	0.333
	0.01	ST	36.8	84.25	0.952	0.797	0.349	1	2.571	3
		SA	12	26.5	0.081	0.084	0.059	0.636	1.714	4.5
		HS	22.2	55.5	1.168	0.645	0.235	0.273	0.857	1.333
		BS	24.6	59	0.129	0.141	0.141	0.182	0.286	0.5
	0.1	ST	111.882	392.737	0.696	0.626	0.313	2.345	1.352	1.605
		SA	58.853	205.737	0.052	0.048	0.034	0.448	1.759	5.698
		HS	39.235	134.895	0.624	0.39	0.144	0.586	0.407	0.767
		BS	3.559	12.105	0.007	0.007	0.007	0.069	0.037	0.047
50,000	0.001	ST	17.875	40	0.829	0.668	0.262	1	3	3.4
		SA	6.5	14.667	0.077	0.08	0.051	0.714	1.6	4
		HS	14	37.333	1.048	0.586	0.201	0.429	1.2	1.6
		BS	247.875	599.333	1.691	1.739	1.663	0.571	1	1.2
	0.01	ST	76.244	259.333	0.62	0.576	0.241	1.091	1.455	1.769
		SA	35.463	104.667	0.05	0.048	0.032	0.409	0.939	4.038
		HS	27.342	91.917	0.699	0.391	0.14	0.227	0.333	0.5
		BS	48.342	150.667	0.113	0.118	0.115	0.227	0.152	0.154
	0.1	ST	158.884	596.427	0.32	0.339	0.201	0.888	0.369	0.589
		SA	58.382	219.281	0.018	0.018	0.012	0.231	1.65	6.386
		HS	52.693	197.812	0.254	0.238	0.131	0.806	0.136	0.35
		BS	5.565	19.104	0.004	0.004	0.004	0.037	0.016	0.03
100,000	0.001	ST	29.462	68.6	0.747	0.62	0.281	0.917	2.25	2.429
		SA	8.769	20.2	0.062	0.065	0.045	0.583	1.5	3.857
		HS	17	44	0.877	0.501	0.183	0.333	0.875	1.286
		BS	338.077	1,001	1.795	1.868	1.865	0.5	1.125	1.143
	0.01	ST	96.747	341.455	0.483	0.464	0.189	1.457	1.045	1.059
		SA	51.279	179.318	0.035	0.033	0.022	0.314	1.358	3.353
		HS	35.57	123.909	0.494	0.28	0.1	0.286	0.254	0.324
		BS	54.392	179.136	0.093	0.098	0.098	0.2	0.119	0.118
	0.1	ST	141.405	414.681	0.224	0.24	0.146	0.471	0.193	0.246
		SA	51.9	152.039	0.012	0.013	0.008	0.143	1.54	4.825
		HS	68.372	199.84	0.204	0.203	0.116	0.611	0.081	0.173
		BS	5.403	14.858	0.003	0.004	0.004	0.027	0.012	0.015

core Xeon 3.3GHz CPU and 32GB RAM. Gödel-hashing library used in this work are available from: https://github.com/shuyingliang/cgh (in C++) and https://github.com/shuyingliang/godelhash (in Scala).

13.2.4 Applying Gödel hashing to speedup static analysis

As motivated in Chapter 11, a pure pushdown exception-flow analysis is an excellent candidate to validate the benefits of Gödel hashes in improving analysis run time.

To evaluate the impact of Gödel hashes in accelerating static analysis of Android apps, I refactor the pushdown exception-flow analyzer *AnaDroid* and instrument the Gödel hashing domains. Gödel hashing domains mainly refer to Gödel hashing set and Gödel hashing store. The implementation of Gödel hashing sets in Scala in Section 13.2 (Scala version) is reused. Gödel hashing store is implemented by making Gödel hashing sets the type of the range in the $\widehat{Store}$ component. The Gödel domains are subtypes of the abstract domains, in addition to default implementation of set and hash map in Scala. This can make the analyzer accommodate two kinds of abstract domains in the same framework.

For the same reasons as illustrated in Section 13.1, I evaluate the Gödel hashes instrumented analyzer on the DaCapo benchmarks.

Table 13.4 presents the runtime speedup for the DaCapo benchmarks that are compiled in Android DVM. With Gödel hash domains, the analysis can run tens of times faster than the one without.

13.3 Evaluation on static malware detection

This section reports the evaluation on the advantages of using the pushdown exception-flow analysis to help human analysts identify malware. Section 13.3.1 describes the overview of the process and results, followed by sections to illustrate the characteristics of challenge suite (Section 13.3.2), the metrics of accuracy (Section 13.3.3) and the experiment setup (Section 13.3.4). Section 13.3.5 reports the characteristics of the detected malware, which are summarized according to the four categories that are defined in Chapter 1, Section 1.1.

13.3.1 Overview

There were two teams of analysts analyze a challenge suite of 52 Android apps released as part of the DARPA Automated Program Analysis for Cybersecurity (APAC) program. Both teams are composed of the same five people, a mixture of graduate students and undergraduate students, however, the applications are shuffled, ensuring the same app is

Table 13.4: Analysis runtime speedup with Gödel hashes in DaCapo benchmarks.

benchmark name	**lines**	**speed-up**
antlr	35,000	22.2x
bloat	70,344	5.9x
chart	217,788	14.2x
fop	184,316	11.7x
jython	110,867	10.7x
hsqldb	155,591	18.7x
luindex	38,221	33.4x
lusearch	87,00	30.1x
pmd	55,000	17x
xalan	259,026	14.7x

not evaluated by the same analyst. The first team analyzed the apps with a version of *AnaDroid* that uses traditional (finite-state-machine-based) control-flow-analysis used in many existing malware analysis tools; the second team analyzed the apps with a version of *AnaDroid* that uses my enhanced pushdown-based control-flow-analysis. I measure the time the analyzer takes, the time human analysts spent reviewing the results, and the accuracy of the malware identification by a human analyst. All other factors being equal, I found a statistically significant ($p < 0.05$) decrease in time and a statistically significant increase ($p < 0.05$) in accuracy with the pushdown-based analyzer.

13.3.2 The challenge suite

Among the apps, 47 are adapted from apps found on the Android market, Contagio [52], or the developer's source repository. A third-party within the APAC project injects malicious behavior into these apps and uses an antidiffing tool on apps with larger code bases to make it difficult to simply diff the application with the original source code. The remaining five apps are variants of the original 47 apps with different malicious behaviors. For example, *App1* may leak location information to a malicious website while *App2* may not. The apps range in size from 18.7 KB to 10 MB, with 11,600 lines of source code in each app on average.

13.3.3 Accuracy

The accuracy in malware identification is measured by four standards:

- *Is the app is malicious?*
- *Where is the maliciousness?*

  This requires specifying the possible paths with class name, method name and line number.

- *What is the trigger of the maliciousness?*
- *Why is it malicious?*

The results of human-in-the-loop analysis in the four aspects have to be compared with the ones released by DARPA. Failing to answer any one of the four correctly is counted as *inaccurate.*

13.3.4 Experiment setup

The two teams of analysts were given instructions on how to use the tool (both versions of the tool use similar UIs and output forms) and some warm-up exercises on a couple of example apps. Then they were given example from the DARPA-supplied challenge suite. The analysts used *AnaDroid* (deployed as a web application on my server) to analyze each app and then record the run time of the analyzer, the total time the human analysis spent investigating the results, and the answers of the four questions (described in Section 13.3.3), based on the analysis results (and what the apps advertise).

I have made efforts to ensure that other factors remained unchanged so that the only difference was the tool the analyst used to detect malware. This restriction can help us gain insight into whether any improvement is made by my new analysis techniques.

Finally, I compare the analysts' results with the DARPA supplied information on the four standards to check accuracy and run statistical analysis using one-way analysis of variance (ANOVA) to get the mean value and p-value of analyzer time, analyst time, and accuracy. This allows us to see the statistical results of the experiment on finite-state-based-machine versus pushdown-based control-flow analysis. This is shown in Table 13.5.

I found that pushdown malware analysis leads to statistically significant improvements with $p < 0.05$ in both accuracy and analysis time over traditional static analysis.

Table 13.5: Comparison of finite-state based vs. pushdown malware analysis: pushdown malware analysis leads to statistically significant improvements with $p < 0.05$ in both accuracy and analysis time over traditional static analysis method.

Metrics	**Methods**	**mean**	**p-value**
Analyzer Time	Finite	994 sec	**0.003**
	Pushdown	560 sec	
Analyst Time	Finite	1.13 hr	**0.0**
	Pushdown	0.44 hr	
Accuracy	Finite	71%	**0.0005**
	Pushdown	95%	

13.3.5 Case studies

In this section, I report case studies of malware detected by *AnaDroid*. The malicious behavior of the 52 apps are summarized in Table 13.6, based on the four categories that are described in Chapter 1, Section 1.1.

From the experiments, I found the taint-flow analysis to be more useful than the least permissions analysis in identifying these behaviors, since half of these apps are designed to avoid requesting any permissions. In addition, I found that finite-state-based analysis can lead to many spurious execution flows in the control graph when the apps have a lot of exception handling code. The pushdown-based model, on the other hand, produces more precise execution flows, which contributes to the sharp decline in analyst time when using *AnaDroid*.

Table 13.6: Vulnerabilities summarization.

Vulnerabilities	**Percentile**	**Case examples**
Data leakage	57%	location, pictures, SMS, ID, *etc.* ex-filtrated to URL, intents, or predefined local file path.
Data tampering	10%	fill local file system with meaningless data, (recursive) deletion of files
DoS attack	11%	inode exhaustion via log, battery drainage (brightness, WiFi, *etc.*)
Other	28%	random vibration, block or intercept SMS messages

CHAPTER 14

RELATED WORK

This dissertation is inspired by existing related work and technology. I separate them in six categories, and discuss some representative work in each category. The outline is summarized as follows.

Section 14.1 discusses the existing approaches in analyzing exception-flow, and compares the pushdown approach in modeling control-flow in general.

Section 14.2 discusses a representative portion of the points-to analysis literature.

Section 14.3 discusses the work in pushdown analysis in higher-order settings, how they inspire the dissertation work and what techniques have been advanced by this work. Section 14.5 describes a subline of the pushdown analysis research.

For Android security specifically, Section 14.6 presents previous and state of the art technology in static taint-flow analysis and Section 14.7 discusses related work of Android malware detection in general.

14.1 Exception-flow analysis

The bulk of the earlier literature for analyzing Java programs has generally focused on finite-state abstractions, *i.e.*, k-CFA and its variants.

Specifically, for the work that acknowledges exceptional flows, the analysis is based on either context-insensitivity or a limited form of context-sensitivity. Analyzers that use only syntactic, type-based analysis of exceptional flow are extremely imprecise [53, 54, 48]. Propagating exceptions via the imprecise call graphs cause the analysis to result in: (1) inclusion of many spurious paths between exception throw sites and handlers that are not truly realizable at run time; (2) unable to tell and differentiate where an exception comes from.

There are three kinds of approach in modeling exception handling. One approach assigns all exceptions thrown in a program to a single global variable. This variable is then read at an exception catch site. This approach is imprecise since it has no knowledge of which exception propagates to a catch site [55, 56].

The second approach analyzes exceptional control-flow only intraprocedurally, computing only local catch clauses for a try block, with no dynamic propagation of exceptions interprocedurally [48].

The third approach is co-analysis using both control-flow analysis and points-to analysis (a.k.a. on-the-fly control-flow construction) to handle exceptions, which yields reasonable precision, compared to the aforementioned two approaches, as documented in past precision studies [37, 57, 7]. However, even for the best co-analysis, where boosting context-sensitivity improves the analysis of exceptions, it does not improve as much as it does for points-to analysis. It is too easy for exceptions to cross context boundaries and merge. My analysis shares similarities in this line of work, with respect to the co-analysis of both control-flow, exception-flow and points-to analysis. The differences from all the three approaches are two-fold: (1) the pushdown exception-flow can match precisely return flows of function calls and exceptions; (2) the abstract garbage collection and its enhanced version does not have limited context-, object- and field-sensitivity.

The work [58] have proposed a modular refinement to construct control graph to analyze interprocedual exceptions for C++ programs. They avoid points-to analysis, opting to use static type information to decide which catch handlers are invoked. One interesting thing worth pointing out is that the paper mentions that stack unwinding is a major issue in C++, because when an exception escapes out of a function, destructors are invoked on all stack allocated objects between the occurrence of the exception and the catch handler in a process [58]. Intuitively, this issue can be resolved naturally within the pushdown scheme with a stack faithfully models the control-flow and exceptions flows.

14.1.1 Pushdown exception-flow analysis

There is little work on pushdown analysis for object-oriented languages as a whole. Sridharan and Bodik proposed demand-driven analysis for Java that matches reads with writes to object fields selectively, by using refinement [59]. They employ a refinement-based contex-free language (CFL) reachability technique that refines calls and returns to valid matching pairs, but approximates for recursive calls. They do not consider specific applications of CFL-reachability to exception-flow.

14.2 Points-to analysis

Precise and scalable context-sensitive points-to analysis has been an open problem for decades. I describe a portion of the representative work in the literature.

Progress in general has been gradual, with results like object-sensitivity [60, 61] intermittently providing a leap for most programs. Most results target improvements for individual classes of programs. My analyses targets all programs, and it is orthogonal to and compatible with results like object-sensitivity.

Much work in pointer analysis exploits methods to improve performance by strategically reducing precision. Lattner *et al.* show that an analysis with a context-sensitive heap abstraction can be efficient by sacrificing precision under unification constraints [62]. In full-context-sensitive pointer analysis, Milanova *et al.* found that an object-sensitive analysis [61] is an effective context abstraction for object-oriented programs. This is confirmed by the extensive evaluation in the work of [57]. Binary decision diagrams (BDDs) have been used to compactly represent the large amount of redundant data in context-sensitive pointer analysis efficiently [63, 64, 65]. Specifically, the work [65] reduces the redundancy by choosing the right context abstractions. Such advancements could be applied to my pushdown framework, as they are orthogonal to its central thesis. Recently, the work [66] exploits liveness analyses to improve points-to analysis. My work also uses liveness analyses but extends it to work with abstract garbage collection. In fact, to the best of my knowledge, my work is the first work that explores abstract garbage collection in analyzing object-oriented programs and enhances it with liveness analysis to explicitly prune points-to precision.

14.3 Pushdown analysis for the λ-calculus

Vardoulakis and Shivers's CFA2 [67] is the precursor to the pushdown control-flow analysis [21].

CFA2 is a table-driven summarization algorithm that exploits the balanced nature of calls and returns to improve return-flow precision in a control-flow analysis. While CFA2 uses a concept called "summarization," it is a summarization of execution paths of the analysis, roughly equivalent to Dyck state graphs.

In terms of recovering precision, pushdown control-flow analysis [21] is the dual to abstract garbage collection: it focuses on the global interactions of configurations via transitions to precisely match push-pop/call-return, thereby eliminating all return-flow merging. However, pushdown control-flow analysis does not directly avoid the value merging problem. This work directly draws on the work of pushdown analysis for higher-order programs [21] and introspective pushdown system (IPDS) for higher-order programs [18].

I extend the earlier work in two dimensions: (1) I generalize the framework to an object-oriented language; (2) I adapt the Dyck state graph synthesis algorithm to handle the new stack change behavior introduced by exceptions; (3) I formulate necessary details to design and implement a static analyzer even in the exceptions; (4) I integrate static taint-flow analysis in the improved pushdown exception-flow analysis.

14.4 Pushdown analysis for the μ-calculus

The work [68] defines a fixpoint logic over execution trees of structured programs for *mu*-calculus and associates the intra- and intercontrol-flow that is produced from the pushdown fixpoint computation with some useful and interesting properties. The combination admits tractable model-checking. What is notable is that their work uses the k-coloring technique to distinguish multiple-returning points. This enables some refined and expressive property specification at some specific calls and return points of the program. It is beneficial to extend the pushdown system for Android programs with the k-coloring technique to prove some security properties in general.

14.5 CFL- and pushdown-reachability techniques

The work [18] develop a pushdown reachability algorithm suitable for the pushdown systems that I generate. It essentially draws on CFL- and pushdown-reachability analysis [69, 70, 16, 71]. For instance, epsilon closure graphs, or equivalent variants thereof, appear in many context-free-language and pushdown reachability algorithms. Dyck state graph synthesis is an attractive perspective on pushdown reachability because it allows targeted modifications to the algorithm.

CFL-reachability techniques have also been used to compute classical finite-state abstraction CFAs [72] and type-based polymorphic control-flow analysis [73]. These analyses should not be confused with pushdown control-flow analysis, which is computing a fundamentally different kind of CFA.

14.6 Static taint-flow analysis

Static taint analysis is proposed to track and detect whether tainted values (usually unsanitized user input) may flow into security sinks. It is applied a lot in detecting vulnerabilities in web apps. Pixy [74] is a static taint analysis for hypertext preprocessor (PHP) that propagates taint information and implements finely tuned alias analysis. Xie and Aiken designed a more precise and scalable analysis for detecting SQL injection attacks in PHP by using block- and function-summaries [75]. The taint analysis scales well but they are conducted in intraprocedural fashion and can not be used to track taint information in objects. This is also the case for other work, such as [76, 77, 78]. Liang and Might take a different approach in information flow analysis for scripting languages such as Python [36]. The analysis is developed in AAM style [9] in abstract interpretation.

A taint-flow analysis for Java programs was developed by [26]. Realizing tracking data-flow through objects in general is difficult, [26] resolves the problem with "nested taint depth." It is tuned to be, at most, two levels, which can easily exceed the behaviors commonly experienced in modern web applications. The work in ANDROMEDA with on-demand alias analysis was later improved by [79]. It uses "access path" that indicates the levels of object field reference to remember, and distinguish and propagate data-flow information, when performing forward and backward analysis. The semantics involving object related operations are well specified. It is reported to work in commercial use and can analyze Java, Javascript and .NET. The technique of on-demand alias analysis to track taint-flows is used in the work of FlowDroid [80] in tracking private sensitive data in Android apps. It also improves ANDROMEDA with flow-sensitive analysis.

My approach in static taint-flow analysis is orthogonal to this line of work. I adopt abstract interpretation design methodology for object-oriented languages and perform on-the-fly analysis with control-flow, exception-flow and points-to analysis. The advantage of my approach is comparatively high precision with good-enough performance. This is reasonable in malware detection and identification scenarios.

14.7 Malware detection for Android applications

Several analyses have been proposed for Android malware detection.

Dynamic taint analysis has been applied to identify security vulnerabilities at run time in Android applications. TaintDroid [81] dynamically tracks the flow of sensitive information and looks for confidentiality violations. IPCInspection [82], QUIRE [83], and XManDroid [84] are designed to prevent privilege-escalation, where an application is compromised to provide sensitive capabilities to other applications. The vulnerabilities introduced by interapp communication is considered future work. However, these approaches typically ignore implicit flows raised by control structures in order to reduce run-time overhead. Moreover, dynamically executing all execution paths of these applications to detect potential information leaks is impractical. The limitations make these approaches inappropriate for computing information flows for all submitted applications.

Woodpecker [85] uses traditional data-flow analysis to find possible capability leaks. Comdroid [86] targets vulnerabilities related to interapp communications. However, it does not perform deep program analysis as my analysis does, and this results in high false positive rates. SmartDroid [87] targets finding complex UI triggers and paths that lead to sensitive sinks. It addresses imprecision of static analysis by combining dynamic executions to filter out infeasible paths at run time. CHEX [5] focuses on detecting *component hi-jacking* by augmenting existing analysis framework using app-splitting to handle Android's multiple entry points. My tool takes a significantly different approach from it (and other finite-state-based static analysis tools) in two aspects: (1) I use pushdown flow analysis that handles traditional control-flow and exception-flow precisely and efficiently. (2) My tool enables human-in-the-loop analysis by allowing the analyst to supply predicates for the analyzer, allowing it to highlight inspection of deeply disguised malware.

Language-based information flow [88, 89, 90] allows developers to annotate variables with security attributes and compilers use the attributes to enforce information control-flow. The main concern of this approach is that it imposes additional burdens on developers whose major focus and interest is business logic. In addition, it can not support the vast majority of legacy code.

The work [91] proposes enforcing a fine-grained permission system. It limits access to resources that could normally be accessed by one of Android's default permissions. Specifically, the security policy uses a white list to determine which resources an app can use and a black list to deny access to resources. In addition, strings potentially containing URLs are identified by pattern matching and constant propagation is used to infer more specific Internet permissions. The work [85] have also identified unprivileged malicious apps that can exploit permissions on protected resources through a privileged agent (or app component in my test suite) that does not enforce permission checks. *Anadroid* can also identify this malicious behavior.

Stowaway [92] is a static analysis tool identifying whether an application requests more permissions than it actually uses. PScout [93] aims for a similar goal, but produces more precise and fine-grained mapping from APIs to permissions. My least permission report uses the PScout permission map as my analysis's database. However, they use a different approach, adapting testing methodology to test applications and identify APIs that require permissions, while my approach annotates APIs with permissions and statically analyzes all executable paths.

Another approach to enforce security control on mobile devices is delegating the control to users themselves. iOS and Window user account control [94] can prompt a dialog to request permissions from users when applications try to access resources or make security or privacy-related system level changes. Depending on users to enforce security control is putting users at risk. Notification prompts by tools usually provide no insights of how users' private sensitive data are used, and users tend to grant permissions in order to install the apps that they want. Thus, it is desirable to stop potential malware from floating into the market beforehand via strict inspections. My tool is designed with analysts in mind and can help them identify malicious behaviors of submitted applications efficiently.

CHAPTER 15

CONCLUSION

The contribution of this work is two-fold: (1) it develops generic, highly precise static analysis of object-oriented programs with multiple entry points; (2) it constructs an effective malware identification system with a human in the loop.

Specifically, for the primary challenge in analyzing object-oriented programs—mutual dependence between control-flow, exception-flow and points-to analysis—I tackle it from two angles: (1) I develop the pushdown exception-flow analysis to refine the return flows as well as exception flows. (2) I generalize abstract garbage collection in an object-oriented setting to "reclaim" unreachable resources to refine points-to information. Precision is further improved when the abstract garbage collection is enhanced with liveness analysis.

To tackle the Android specific challenge—multiple entry points—I develop the entry point saturation technique to "soundly" approximate the execution of the permutations of asynchronous entry points.

With the foundational analysis constructed for the two challenges, I develop a static taint-flow analysis in abstract interpretation framework for security purpose. When it is built on classical abstract interpretation, the static taint analysis can leverage the context-, object- and field-sensitivity; when it is built on pushdown exception-flow analysis, it can track taint-flow information with even better precision.

To accelerate the speed of static analysis, I develop a compact and efficient encoding scheme, called Gödel hashes and integrate them into the analysis framework.

All the techniques are realized and evaluated in a system, named *AnaDroid*. *AnaDroid* is designed with a human in the loop to specify analysis configuration, the property of interest and then make the final judgment and identify where the maliciousness is, based on analysis results. The analyses results include control-flow graphs with suspiciousness highlighted, security reports of permissions and risk rankings. The experiments show that *AnaDroid* can lead to precise and fast malware identification.

APPENDIX A

THE GENERALIZED DALVIK INSTRUCTION SET

Table A.1: The generalized Dalvik instruction set (00-1e).

OpNumber	Dalvik Instruction	Generalized Instruction
00	nop	nop
01	move	assign
02	move/from16	
03	move/16	
04	move-wide	
05	move-wide/from16	
06	move-wide/16	
07	move-object	
08	move-object/from16	
09	move-object/16	
0a	move-result	
0b	move-result-wide	
0c	move-result-object	
0d	move-exception	
0e	return-void	return
0f	return	
10	return-wide	
11	return-object	
12	const/4	assign
13	const/16	
14	const	
15	const/high16	
16	const-wide/16	
17	const-wide/32	
18	const-wide	
19	const-wide/high16	
1a	const-string	
1b	const-string/jumbo	
1c	const-class	
1d	monitor-enter	
1e	monitor-exit	

Table A.2: The generalized Dalvik instruction set (1f-4a).

OpNumber	Dalvik Instruction	Generalized Instruction
1f	check-cast	check-cast
20	instance-of	assign
21	array-length	
22	new-instance	new
23	new-array	
24	filled-new-array	
25	filled-new-array/range	
26	fill-array-data	
27	throw	throw
28	goto	goto
29	goto/16	
2a	goto/32	
2b	packed-switch	switch
2c	sparse-switch	
2d	cmpl-float	assign
2e	cmpg-float	
2f	cmpl-double	
30	cmpg-double	
31	cmp-long	
32	if-eq	if
33	if-ne	
34	if-lt	
35	if-ge	
36	if-gt	
37	if-le	
38	if-eqz	
39	if-nez	
3a	if-ltz	
3b	if-gez	
3c	if-gtz	
3d	if-lez	
3e..43	(unused)	
44	aget	array-get
45	aget-wide	
46	aget-object	
47	aget-boolean	
48	aget-byte	
49	aget-char	
4a	aget-short	

Table A.3: The generalized Dalvik instruction set (4b-73).

OpNumber	Dalvik Instruction	Generalized Instruction
4b	aput	array-put
4c	aput-wide	
4d	aput-object	
4e	aput-boolean	
4f	aput-byte	
50	aput-char	
51	aput-short	
52	iget	field-get
53	iget-wide	
54	iget-object	
55	iget-boolean	
56	iget-byte	
57	iget-char	
58	iget-short	
59	iput	field-put
5a	iput-wide	
5b	iput-object	
5c	iput-boolean	
5d	iput-byte	
5e	iput-char	
5f	iput-short	
60	sget	field-sget
61	sget-wide	
62	sget-object	
63	sget-boolean	
64	sget-byte	
65	sget-char	
66	sget-short	
67	sput	field-sput
68	sput-wide	
69	sput-object	
6a	sput-boolean	
6b	sput-byte	
6c	sput-char	
6d	sput-short	
6e	invoke-virtual	invoke-virtual
6f	invoke-super	invoke-super
70	invoke-direct	invoke-direct
71	invoke-static	invoke-static
72	invoke-interface	invoke-interface
73	(unused)	

Table A.4: The generalized Dalvik instruction set (74-9f).

OpNumber	Dalvik Instruction	Generalized Instruction
74	invoke-virtual/range	invoke-virtual
75	invoke-super/range	invoke-super
76	invoke-direct/range	invoke-direct
77	invoke-static/range	invoke-static
78	invoke-interface/range	invoke-interface
79..7a	(unused)	
7b	neg-int	*atomic-op*, assign
7c	not-int	
7d	neg-long	
7e	not-long	
7f	neg-float	
80	neg-double	
81	int-to-long	
82	int-to-float	
83	int-to-double	
84	long-to-int	
85	long-to-float	
86	long-to-double	
87	float-to-int	
88	float-to-long	
89	float-to-double	
8a	double-to-int	
8b	double-to-long	
8c	double-to-float	
8d	int-to-byte	
8e	int-to-char	
8f	int-to-short	
90	add-int	
91	sub-int	
92	mul-int	
93	div-int	
94	rem-int	
95	and-int	
96	or-int	
97	xor-int	
98	shl-int	
99	shr-int	
9a	ushr-int	
9b	add-long	
9c	sub-long	
9d	mul-long	
9e	div-long	
9f	rem-long	

Table A.5: The generalized Dalvik instruction set (a0-c9).

OpNumber	Dalvik Instruction	Generalized Instruction
a0	and-long	*atomic-op*, assign
a1	or-long	
a2	xor-long	
a3	shl-long	
a4	shr-long	
a5	ushr-long	
a6	add-float	
a7	sub-float	
a8	mul-float	
a9	div-float	
aa	rem-float	
ab	add-double	
ac	sub-double	
ad	mul-double	
ae	div-double	
af	rem-double	
b0	add-int/2addr	
b1	sub-int/2addr	
b2	mul-int/2addr	
b3	div-int/2addr	
b4	rem-int/2addr	
b5	and-int/2addr	
b6	or-int/2addr	
b7	xor-int/2addr	
b8	shl-int/2addr	
b9	shr-int/2addr	
ba	ushr-int/2addr	
bb	add-long/2addr	
bc	sub-long/2addr	
bd	mul-long/2addr	
be	div-long/2addr	
bf	rem-long/2addr	
c0	and-long/2addr	
c1	or-long/2addr	
c2	xor-long/2addr	
c3	shl-long/2addr	
c4	shr-long/2addr	
c5	ushr-long/2addr	
c6	add-float/2addr	
c7	sub-float/2addr	
c8	mul-float/2addr	
c9	div-float/2addr	

Table A.6: The generalized Dalvik instruction set (ca-ff).

OpNumber	Dalvik Instruction	Generalized Instruction
ca	rem-float/2addr	*atomic-op*, assign
cb	add-double/2addr	
cc	sub-double/2addr	
cd	mul-double/2addr	
ce	div-double/2addr	
cf	rem-double/2addr	
d0	add-int/lit16	
d1	rsub-int	
d2	mul-int/lit16	
d3	div-int/lit16	
d4	rem-int/lit16	
d5	and-int/lit16	
d6	or-int/lit16	
d7	xor-int/lit16	
d8	add-int/lit8	
d9	rsub-int/lit8	
da	mul-int/lit8	
db	div-int/lit8	
dc	rem-int/lit8	
dd	and-int/lit8	
de	or-int/lit8	
df	xor-int/lit8	
e0	shl-int/lit8	
e1	shr-int/lit8	
e2	ushr-int/lit8	
e3..ff	(unused)	

APPENDIX B

ADDITIONAL CONCRETE AND ABSTRACT TRANSITION RELATIONS

B.1 Additional concrete transition relations

- **Stepping over nops and labels:** The simplest instruction nop does not change any component in the state:

  $$(\llbracket(\mathsf{nop})^{\ell} : \vec{s}\rrbracket, fp, \sigma, \kappa, t) \Rightarrow (\vec{s}, fp, \sigma, \kappa, t')$$
  $$\text{where } t' = \mathit{tick}(\ell, t).$$

  label and line statement shares the same transition form.[1]

- **Unconditional jumps:** This kind of statement forces the program to jump to the target statement sequence:

  $$(\llbracket(\mathsf{goto}\ \mathit{label})^{\ell} : \vec{s}\rrbracket, fp, \sigma, \kappa, t) \Rightarrow (S(\mathit{label}), fp, \sigma, \kappa, t')$$
  $$\text{where } t' = \mathit{tick}(\ell, t).$$

  where the function $S : \mathsf{Label} \to \mathsf{Stmt}^*$ maps a label to the statement sequence starting with that label.

- **Conditionals:** The if-goto is not much more complicated than a nop or goto, but it needs to evaluate the conditional expression ($t' = \mathit{tick}(t)$):

  $$(\llbracket(\mathsf{if}\ \text{æ}\ (\mathsf{goto}\ \mathit{label})) : \vec{s}\rrbracket, fp, \sigma, \kappa, t) \Rightarrow \begin{cases} (\vec{s}, fp, \sigma, \kappa, t') & \mathcal{A}(\text{æ}, fp, \sigma) \neq \mathit{false} \\ (S(\mathit{label}), fp, \sigma, \kappa, t') & \text{otherwise} \end{cases}$$

[1] The line statement is mainly for instrumenting *context* information to the statements that are actually interpreted.

- **Atomic assignments:** Atomic assignment statements assign the value of an atomic expression to a variable (register). This involves evaluating the expression, calculating the frame address to modify and then updating the store.

$$(\llbracket(\mathsf{assgin}\ \mathit{name}\ æ) : \vec{s}\rrbracket, \mathit{fp}, \sigma, \kappa, t) \Rightarrow (\vec{s}, \mathit{fp}, \sigma', \kappa, t')$$
$$\text{where } t' = \mathit{tick}(t) \text{ and } \sigma' = \sigma[(\mathit{fp}, \mathit{name}) \mapsto \mathcal{A}(æ, \mathit{fp}, \sigma)].$$

A large set of instruction statements are transformed into assign form. For example, (move-result *name*) is transformed to (assign *name* ret) form.

- **Check cast:** check-cast checks whether the object reference in register name can be cast to an instance of a class referenced by *class-name*. If *class-name′* is any subtype of *class-name*:

$$(\llbracket(\mathsf{assign}\ \mathit{name}\ (\mathsf{check\text{-}cast}\ æ\ \mathit{class\text{-}name}))^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, \kappa, t) \Rightarrow (\vec{s}, \mathit{fp}, \sigma', \kappa, t')$$
$$\text{where } t' = \mathit{tick}(\ell, t), (\mathit{op}, \mathit{class\text{-}name}') = \mathcal{A}(æ, \mathit{fp}, \sigma)$$
$$\sigma' = \sigma[(\mathit{fp}, \mathit{name}) \mapsto (\mathit{op}, \mathit{class\text{-}name}')].$$

If *class-name′* is *not* a subtype of *class-name*, then the statement will throw ClassCastException. This is simulated by making the next statement a fake throw statement, with other components unchanged.

- **Instance-of:**

$$(\llbracket(\mathsf{assign}\ \mathit{name}\ (\mathsf{insance\text{-}of}\ æ\ \mathit{class\text{-}name}))^{\ell} : \vec{s}\rrbracket, \mathit{fp}, \sigma, \kappa, t) \Rightarrow (\vec{s}, \mathit{fp}, \sigma', \kappa, t')$$
$$\text{where } t' = \mathit{tick}(\ell, t), (\mathit{op}, \mathit{class\text{-}name}') = \mathcal{A}(æ, \mathit{fp}, \sigma),$$
$$\sigma' = \begin{cases} \sigma[(\mathit{fp}, \mathit{name}) \mapsto \{\mathsf{true}\})], & \text{if } \mathit{class\text{-}name}' \text{ is any subtype of } \mathit{class\text{-}name} \\ \sigma[(\mathit{fp}, \mathit{name}) \mapsto \{\mathsf{false}\}], & \text{otherwise.} \end{cases}.$$

As we can see, the above rule has some similar logic as check-cast.

B.1.1 Array

Following rules deal with array related instructions. Array pointer $ap \in \mathit{ArrayPointer}$ is similar to object pointer, but it is used to denote array reference, and the *type* is the type for elements. To index array element, we introduce array address *ArrayAddr* into*Addr* :

$$a \in \mathit{Addr} = \mathit{RegAddr} + \mathit{FieldAddr} + \mathit{ArrayAddr}$$
$$\mathit{ArrayAddr} = \mathit{ArrayPointer} \times \mathcal{Z}.$$

To allocate an array address, we need an allocation function: $\mathit{alloc}_a : \mathit{Conf} \rightarrow \mathit{ArrayAddr}$

- **new array:**

$$\begin{aligned}(\overbrace{[\![(\mathsf{assign}\ \mathit{name}\ (\mathsf{new\text{-}array}\ æ\ \mathit{type}))^{\ell} : \vec{s}]\!], \mathit{fp}, \sigma, \kappa, t)}^{c} &\Rightarrow (\vec{s}, \mathit{fp}, \sigma'', \kappa, t')\\ \text{where } t' =& \mathit{tick}(\ell, t)\\ \mathit{ap} =& \mathit{alloc}_a(c)\\ \sigma' =& \sigma[(\mathit{fp}, \mathit{name}) \mapsto (\mathit{ap}, \mathit{type})]\\ \sigma'' =& \sigma'[(\mathit{ap}, \mathsf{len}) \mapsto \mathcal{A}(æ, \mathit{fp}, \sigma)].\end{aligned}$$

- **array length:**

$$\begin{aligned}([\![(\mathsf{assign}\ \mathit{name}\ (\mathsf{array\text{-}length}\ æ))^{\ell} : \vec{s}]\!], \mathit{fp}, \sigma, \kappa, t) &\Rightarrow (\vec{s}, \mathit{fp}, \sigma', \kappa, t')\\ \text{where } t' =& \mathit{tick}(\ell, t)\\ (\mathit{ap}, \mathit{type}) =& \mathcal{A}(æ, \mathit{fp}, \sigma)\\ \sigma' =& \sigma[(\mathit{fp}, \mathit{name}) \mapsto \sigma(\mathit{ap}, \mathsf{len})].\end{aligned}$$

- **filled new array:**

$$\begin{aligned}([\![(\mathsf{filled\text{-}new\text{-}array}\ æ_0 \ldots æ_n\ \mathit{type})^{\ell} : \vec{s}]\!], \mathit{fp}, \sigma, \kappa, t) &\Rightarrow (\vec{s}, \mathit{fp}, \sigma', \kappa, t')\\ \text{where } t' =& \mathit{tick}(\ell, t)\\ (\mathit{ap}, \mathit{type}) =& \sigma(\mathit{fp}, \mathsf{ret})\\ d_i =& \mathcal{A}(æ_i, \mathit{fp}, \sigma), i = 0 \ldots n\\ \sigma' =& \sigma + [(\mathit{ap}, i) \mapsto d_i].\end{aligned}$$

- **aget:**

$$\begin{aligned}([\![(\mathsf{aget}\ \mathit{name}\ æ_a\ æ_i) : \vec{s}]\!], \mathit{fp}, \sigma, \kappa, t) &\Rightarrow (\vec{s}, \mathit{fp}, \sigma', \kappa, t')\\ \text{where } (\mathit{ap}, \mathit{type}) =& \mathcal{A}(æ_a, \mathit{fp}, \sigma)\\ i =& \mathcal{A}(æ_i, \mathit{fp}, \sigma)\\ \sigma' =& \sigma[(\mathit{fp}, \mathit{name}) \mapsto \sigma(\mathit{ap}, i)].\end{aligned}$$

- **aput:**

$$\begin{aligned}([\![(\mathsf{aput}\ æ_v\ æ_a\ æ_i)^{\ell} : \vec{s}]\!], \mathit{fp}, \sigma, \kappa, t) &\Rightarrow (\vec{s}, \mathit{fp}, \sigma', \kappa, t')\\ \text{where } t' =& \mathit{tick}(\ell, t)\\ (\mathit{ap}, \mathit{type}) =& \mathcal{A}(æ_a, \mathit{fp}, \sigma)\\ i =& \mathcal{A}(æ_i, \mathit{fp}, \sigma)\\ \sigma' =& \sigma[(\mathit{ap}, i) \mapsto \sigma(\mathit{fp}, æ_v)].\end{aligned}$$

B.2 Additional abstract transition relations

- **Stepping over nops and labels:**

$$(\llbracket(\mathsf{nop}) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}), \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}')$$
$$\text{where } t' = tick(\ell, t).$$

- **Unconditional jumps:**

$$(\llbracket(\mathsf{goto}\ label) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (S(label), \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}').$$

- **Conditionals:**

$$(\llbracket(\mathsf{if}\ \ae\ (\mathsf{goto}\ label)) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto \begin{cases} (\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}') & \hat{\mathcal{A}}(\ae, \hat{fp}, \hat{\sigma}) \neq false \\ (S(label), \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}') & \text{otherwise} \end{cases}$$
$$\text{where } t' = tick(\ell, t).$$

- **Atomic assignments:**

$$(\llbracket(\mathsf{assgin}\ name\ \ae) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}')$$
$$\text{where } t' = tick(\ell, t) \text{ and } \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, name) \mapsto \hat{\mathcal{A}}(\ae, \hat{fp}, \hat{\sigma})].$$

A large set of instruction statements are transformed into assign form. For example, (move-result *name*) is transformed to (assign *name* ret) form.

- **Check cast:** check-cast checks whether the object reference in register name can be cast to an instance of a class referenced by *class-name*. So, if *class-name′* is any subtype of *class-name*:

$$(\llbracket(\mathsf{assign}\ name\ (\mathsf{check\text{-}cast}\ \ae\ class\text{-}name))^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), (\widehat{op}, class\text{-}name') \in \hat{\mathcal{A}}(\ae, \hat{fp}, \hat{\sigma})$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, name) \mapsto (\widehat{op}, class\text{-}name')].$$

If *class-name′* is *not* a subtype of *class-name*, then the statement will throw a ClassCastException. This is simulated by making the next statement a fake throw statement, with other components unchanged.

- **Instance-of:**

$$(\llbracket(\mathsf{assign}\ name\ (\mathsf{insance\text{-}of}\ \ae\ class\text{-}name))^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), (\widehat{op}, class\text{-}name') \in \hat{\mathcal{A}}(\ae, \hat{fp}, \hat{\sigma})$$
$$\hat{\sigma}' = \begin{cases} \hat{\sigma} \sqcup [(\hat{fp}, name) \mapsto \{\mathsf{true}\})], & \text{if } class\text{-}name' \text{ is any subtype of } class\text{-}name \\ \hat{\sigma} \sqcup [(\hat{fp}, name) \mapsto \{\mathsf{false}\}], & \text{otherwise.} \end{cases}$$

This has similar logic as check-cast.

B.2.1 Array

The abstract semantics for array is not a direct lifting from the concrete semantics, because the array index in abstract semantics can have partial order on it, which can make the array reference and update complicated. For simplicity (and to avoid precision loss), we adopt the genera abstract approach, which is "crush" all array elements. In other words, the analysis becomes index-insensitive, where all the values that are unordered will be constructed and referenced in an array. There is no need to lift the refactoring to arrays defined in concrete semantics.

- **new array:**

$$(\llbracket(\mathsf{assign}\ \mathit{name}\ (\mathsf{new\text{-}array}\ æ\ \mathit{type}))^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}'', \hat{\kappa}, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{\mathit{tick}}(\ell, \hat{t}), \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, \mathit{name}) \mapsto \{\}].$$

- **array length:**

$$(\llbracket(\mathsf{assign}\ \mathit{name}\ (\mathsf{array\text{-}length}\ æ))^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}).$$

- **filled new array:**

$$(\llbracket(\mathsf{filled\text{-}new\text{-}array}\ æ_0 \ldots æ_n\ \mathit{type})^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t})a \Rightarrow (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}')$$
$$\text{where } \hat{t}' = \mathit{tick}(\ell, \hat{t}), d_i = \hat{\mathcal{A}}(æ_i, \hat{fp}, \hat{\sigma}), i = 0 \ldots n, \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, \mathsf{ret}) \mapsto d_i].$$

- **aget:**

$$(\llbracket(\mathsf{aget}\ \mathit{name}\ æ_a\ æ_i) : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}')$$
$$\text{where } t' = \mathit{tick}(\ell, t), \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, \mathit{name}) \mapsto \hat{\sigma}(\hat{fp}, æ_a)].$$

- **aput:**

$$(\llbracket(\mathsf{aput}\ æ_v\ æ_a\ æ_i)^{\ell} : \vec{s}\rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \leadsto (\vec{s}, \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}')$$
$$\text{where } \hat{t}' = \mathit{tick}(\ell, \hat{t}), \hat{\sigma}' = \hat{\sigma} \sqcup [(\hat{fp}, æ_a) \mapsto \hat{\sigma}(\hat{fp}, æ_v)].$$

APPENDIX C

GÖDEL HASHING OTHER DATA STRUCTURES

C.1 Maps, relations and graphs

The strategy for Gödel-hashing maps, relations and graphs are derivatives of the strategy for Gödel-hashing sets.

C.1.1 Hashing maps

Finite maps can be encoded as sets of pairs; thus:

Definition C.1 *The* ***Gödel hash of a map*** $f : X \to Y$ *with respect to prime map* $P_{X \times Y} : X \times Y \to \mathbb{P}$ *is* $G(f)$*, where:*

$$G(f) = \prod_{x \in dom(f)} P_{X \times Y}(x, f(x)).$$

C.1.2 Hashing relations

Relations can also be encoded as sets of pairs; thus:

Definition C.2 *The* ***Gödel hash of a relation*** $R \subseteq X \times Y$ *with respect to prime map* $P_{X \times Y} : X \times Y \to \mathbb{P}$ *is* $G(R)$*, where:*

$$G(R) = \prod_{x \; R \; y} P_{X \times Y}(x, y).$$

C.1.3 Hashing graphs

A directed graph (V, E) is just a set of vertexes and a set of edges.

Definition C.3 *Given two Gödel set-hashing functions:*

$$G_{\mathcal{P}(V)} : \mathcal{P}(V) \to \mathbb{N} \quad \text{, and} \quad G_{\mathcal{P}(E)} : \mathcal{P}(E) \to \mathbb{N},$$

*the **Gödel hash of a graph** (V, E) is $G(V, E)$:*

$$G(V, E) = (G_{\mathcal{P}(V)}(V), G_{\mathcal{P}(E)}(E)).$$

If one needs a natural number instead of a pair of natural numbers, then one can apply Cantor's bijection $C : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ (used in proving the countability of the rationals) to the result [40].

C.2 Multisets

It is not difficult to extend Gödel hashing to multisets. A **multiset** is a collection which allows multiple copies of identical elements: in the multiset $\{u_1^{m_1}, \ldots, u_n^{m_n}\}$, there are m_i copies of the element u_i. A characteristic function for a multiset M maps each element to its multiplicity: $\chi_M : M \to \mathbb{N}$. Even if multisets are not required, they can make a suitable substitution for sets, since some operations are cheaper. (Additive union, for instance, costs just one multiplication.)

Definition C.4 *Given a universe of discourse $\mathbb{U}$ and a prime map $P : \mathbb{U} \to \mathbb{P}$, the function G provides the **Gödel hash of a multiset**:*

$$G\,\{u_1^{m_1}, \ldots, u_n^{m_n}\} = P(u_1)^{m_1} \times \cdots \times P(u_n)^{m_n},$$

or, equivalently:

$$G(M) = \prod_{u \in \mathbb{U}} P(u)^{\chi_M(u)}.$$

The standard generalization of the set operations map to simple operations under the Gödel encoding.

Lemma C.1 *Additive union reduces to multiplication:*

$$\|A \uplus B\| = \|A\| \times \|B\|.$$

Proof. Hence:

$$\begin{aligned}
\|A \uplus B\| &= \prod_{u \in \mathbb{U}} P(u)^{\chi_{A \uplus B}(u)} \\
&= \prod_{u \in \mathbb{U}} P(u)^{\chi_A(u) + \chi_B(u)} \\
&= \left(\prod_{u \in \mathbb{U}} P(u)^{\chi_A(u)}\right)\left(\prod_{u \in \mathbb{U}} P(u)^{\chi_B(u)}\right) \\
&= \|A\| \times \|B\|.
\end{aligned}$$

■

Lemma C.2 *Union reduces to the least common multiple:*

$$\|A \cup B\| = \|A\| \operatorname{lcm} \|B\|.$$

Proof. The argument is identical in form to that of regular sets. ■

Lemma C.3 *Additive insertion reduces to multiplication:*

$$\|A \uplus \{u\}\,\| = G(A) \times P(u).$$

Proof. By the fundamental theorem of arithmetic. ■

Lemma C.4 *Intersection reduces to the greatest common divisor:*

$$\|A \cap B\| = \|A\| \gcd \|B\|$$

Proof. The argument is identical in form to that of regular sets. ■

Lemma C.5 *Difference reduces to division by the intersection:*

$$\|A - B\| = \frac{\|A\|}{\|A\| \gcd \|B\|.}$$

Proof. The argument is identical in form to that of regular sets. ■

Lemma C.6 *Membership reduces to divisibility:*

$$\|u \in A\| \text{ iff } P(u) \mid G(A).$$

Proof. The argument is identical in form to that of regular sets. ■

Lemma C.7 *Testing multiplicity reduces to divisibility:*

$$\|u^k \in A\| \text{ iff } P(u)^k \mid G(A).$$

Proof. By an argument similar to that for membership. ■

Lemma C.8 *Inclusion reduces to divisibility:*

$$\|A \subseteq B\| \text{ iff } GA \mid G(B).$$

Proof. The argument is identical in form to that of regular sets. ■

C.3 Lists

A Gödel encoding for sequences is actually required in the proofs of incompleteness.

C.3.1 Arithmetic Gödel hashes on sequences

There is a strategy for Gödel hashing lists based on the fundamental theorem of arithmetic, but ironically, the hashes produced by this strategy are rarely more efficient than an array-based representation.

Definition C.5 *The* ***arithmetic Gödel hash of a list*** $\vec{s} \in S^*$ *under the perfect hash map* $H : S \to \mathbb{N}$ *is* $G_{S^*}(\vec{s})$*:*

$$G_{S^*}(\langle s_1, \ldots, s_n \rangle) = p_1^{H(s_1)} \times \cdots \times p_n^{H(s_n)}.$$

C.3.2 Gödel β-hashes on sequences

Though the proof itself required no efficiency, the proof of Gödel's first incompleteness theorem contains a far more space-efficient and effective means for encoding a sequence of natural numbers: a β-function encoding. Since these hashes employ the Chinese remainder theorem and his β-function, we term them Gödel β-hashes. Unlike the factorization-based hash of the previous subsection, it *is* possible to efficiently recover the individual elements of the sequence from a β-hash. We leave a detailed study of the practical applications of these kinds of Gödel hashes to future work.

REFERENCES

[1] Strategy Analytics, “Android captured 79% share of global smartphone shipments in 2013.” http://blogs.strategyanalytics.com/WSS/post/2014/01/29/Android-Captured-79-Share-of-Global-Smartphone-Shipments-in-2013.aspx.

[2] Wikipedia, “Android.” http://en.wikipedia.org/wiki/Android_(operating_system).

[3] Tech-thoughts, “Declining application download rate: Impact on iOS vs. Android engagement.” http://www.tech-thoughts.net/2013/06/declining-app-download-rate-ios-android-engagement.html.

[4] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, “A survey of mobile malware in the wild,” in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM ’11, pp. 3–14, ACM, 2011.

[5] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “CHEX: Statically vetting Android apps for component hijacking vulnerabilities,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS ’12, pp. 229–240, ACM, 2012.

[6] S. Liang, M. Might, D. V. Horn, S. Lyde, T. Gilray, and P. Aldous, “Sound and precise malware analysis for android via pushdown reachability and entry-point saturation,” in *Proceedings of the 3rd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ACM, 2013.

[7] M. Bravenboer and Y. Smaragdakis, “Exception analysis and points-to analysis: Better together,” in *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA’09, pp. 1–12, ACM, 2009.

[8] O. G. Shivers, *Control-flow analysis of higher-order languages.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[9] D. Van Horn and M. Might, “Abstracting abstract machines,” in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP ’10, pp. 51–62, ACM, 2010.

[10] M. Felleisen and D. P. Friedman, “Control operators, the SECD-machine, and the lambda-calculus,” in *3rd Working Conference on the Formal Description of Programming Concepts*, Aug. 1986.

[11] M. Felleisen and D. P. Friedman, “A calculus for assignments in higher-order languages,” in *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL ’87, 1987.

[12] 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, 1982.

[13] J. Midtgaard, "Control-flow analysis of functional programs," *ACM Comput. Surv.*, pp. 10:1–10:33, 2012.

[14] O. G. Shivers, *Control-flow analysis of higher-order languages.* PhD thesis, Carnegie Mellon University, 1991.

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

[16] T. Reps, "Program analysis via graph reachability," *Information and Software Technology*, vol. 40, pp. 701–726, Dec. 1998.

[17] M. Might and O. Shivers, "Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting," in *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, ICFP'06, pp. 13–25, ACM, 2006.

[18] C. Earl, I. Sergey, M. Might, and D. Van Horn, "Introspective pushdown analysis of higher-order programs," in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP'12, ACM, 2012.

[19] O. Agesen, D. Detlefs, and J. E. Moss, "Garbage collection and local variable type-precision and liveness in java virtual machines," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI'98, ACM, 1998.

[20] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1988.

[21] C. Earl, M. Might, and D. Van Horn, "Pushdown control-flow analysis of higher-order programs," in *Proceedings of the 2010 Workshop on Scheme and Functional Programming*, Aug. 2010.

[22] R. Chatterjee, B. G. Ryder, and W. A. Landi, "Complexity of points-to analysis of java in the presence of exceptions," *IEEE Trans. Softw. Eng.*, 2001.

[23] Android Fundamentals, "Bytecode for the Dalvik VM." http://developer.android.com/guide/components/fundamentals.html.

[24] OWASP13, "OWASP top 10 for 2013." https://www.owasp.org/index.php/.

[25] Y. W. Huang, F. Yu, C. Hang, C. H. Tsai, D. T. Lee, and S. Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pp. 40–52, ACM, 2004.

[26] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of web applications," in *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, PLDI'09, pp. 87–97, ACM, 2009.

[27] L. A. Meyerovich and B. Livshits, "ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pp. 481–496, IEEE Computer Society, 2010.

[28] L. Wall, T. Christiansen, R. L. Schwartz, and S. Potter, *Programming Perl (2nd Edition).*

[29] "Ruby." http://ruby-doc.org/docs/ProgrammingRuby/.

[30] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *20th IFIP International Information Security Conference*, pp. 372–382, 2005.

[31] D. Kozlov and Petukhov, "Implementation of tainted mode approach to finding security vulnerabilities for python technology," in *Spring/Summer Young Researchers' Colloquium on Software Engineering*, SYRCoSE'07, June 2007.

[32] J. Seo and M. S. Lam, "Invisitype: Object-oriented security policies," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, NDSS'10, Feb. 2010.

[33] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proceedings of the 14th Annual Network and Distributed System Security Symposium*, NDSS'07, Feb. 2007.

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

[35] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J.Sel. A. Commun.*, vol. 21, pp. 5–19, Sept. 2006.

[36] S. Liang and M. Might, "Hash-flow taint analysis of higher-order programs," in *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS '12, ACM, 2012.

[37] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott, "Robustness testing of Java server applications," *IEEE Trans. Softw. Eng.*, vol. 31, pp. 292–311, Apr. 2005.

[38] O. Shivers, "Control-flow analysis in Scheme," in *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pp. 164–174, 1988.

[39] K. Gödel, "Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i.," *Monatshefte fr Mathematik und Physik*, vol. 38, pp. 173–198, 1931.

[40] G. Cantor, "Über eine eigenschaft des inbegriffes aller reellen algebraischen zahlen," *Mathematische Annalen*, 1874.

[41] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1.* Intel Corporation, September 2013.

[42] M. E. O'Neill, "The genuine sieve of eratosthenes," *Journal of Functional Programming*, October 2008.

[43] R. M. Solovay and V. Strassen, "A fast monte-carlo test for primality," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 84–85, 1977.

[44] G. L. Miller, "Riemann's hypothesis and tests for primality," *Journal of Computer and System Sciences*, vol. 13, no. 3, pp. 300–317, 1976.

[45] M. O. Rabin, "Probabilistic algorithm for testing primality," *Journal of Number Theory*, vol. 12, no. 1, pp. 128–138, 1980.

[46] Google, "Android-apktool: A tool for reverse engineering Android apk files." http://code.google.com/p/android-apktool/.

[47] J. Freke, "Smali, an assembler/disassembler for Android's dex format." http://code.google.com/p/smali/wiki/Registers.

[48] IBM Watson Research Center, "T.J. Watson libraries for analysis (WALA)." http://sourceforge.net/projects/wala/.

[49] M. Sridharan, S. Chandra, J. Dolby, S. Fink, and E. Yahav, "Alias analysis for object-oriented programs," in *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, vol. 7850, pp. 196–232, Springer, 2013.

[50] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06*, ACM, 2006.

[51] G. Kastrinis and Y. Smaragdakis, "Hybrid context-sensitivity for points-to analysis," in *PLDI '13*, ACM, 2013.

[52] Contagio, "Contagio." http://contagiodump.blogspot.com.

[53] X. Leroy and F. Pessaux, "Type-based analysis of uncaught exceptions," *ACM Trans. Program. Lang. Syst.*, vol. 22, pp. 340–377, Mar. 2000.

[54] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 12, pp. 191–221, Apr. 2003.

[55] L. Hendren, "Scaling Java points-to analysis using Spark," in *12th International Conference on Compiler Construction*, Springer, 2003.

[56] O. Lhoták, *Program analysis using binary decision diagrams.* PhD thesis, McGill University, 1987.

[57] O. Lhoták and L. Hendren, "Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, pp. 3:1–3:53, Oct. 2008.

[58] P. Prabhu, N. Maeda, and G. Balakrishnan, "Interprocedural exception analysis for c++," in *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pp. 583–608, Springer-Verlag, 2011.

[59] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for Java," in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, PLDI'06, pp. 387–400, ACM, 2006.

[60] A. Milanova, "Light context-sensitive points-to analysis for Java," in *Proceedings of the 7th ACM Workshop on Program Analysis for Software Tools and Engineering*, 2007.

[61] A. Milanova and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for Java," *ACM Trans. Softw. Eng. Methodol*, vol. 14, p. 2005, 2005.

[62] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proceedings of the ACM SIGPLAN 2007 conference on Programming Language Design and Implementation*, PLDI'07, ACM, 2007.

[63] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using BDDs," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation*, PLDI'03, ACM, 2003.

[64] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, PLDI'04, ACM, 2004.

[65] G. Xu and A. Rountev, "Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis," in *ISSTA '08*, pp. 225–236, ACM, 2008.

[66] U. P. Khedker, A. Mycroft, and P. S. Rawat, "Liveness-based pointer analysis," in *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, (Berlin, Heidelberg), pp. 265–282, Springer-Verlag, 2012.

[67] D. Vardoulakis, *CFA2: Pushdown flow analysis for higher-order languages.* PhD thesis, Northeastern University, 2012.

[68] R. Alur, S. Chaudhuri, and P. Madhusudan, "A fixpoint calculus for local and global program flows," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, 2006.

[69] A. Bouajjani, J. Esparza, and O. Maler, "Reachability analysis of pushdown automata: application to model-checking," in *Proceedings of the 8th International Conference on Concurrency Theory*, pp. 135–150, Springer-Verlag, 1997.

[70] J. Kodumal and A. Aiken, "The set constraint/CFL reachability connection in practice," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, PLDI'04, ACM, 2004.

[71] T. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted pushdown systems and their application to interprocedural dataflow analysis," *Science of Computer Programming*, vol. 58, pp. 206–263, Oct. 2005.

[72] D. Melski and T. W. Reps, "Interconvertibility of a class of set constraints and context-free-language reachability," *Theoretical Computer Science*, vol. 248, pp. 29–98, Oct. 2000.

[73] J. Rehof and M. Fähndrich, "Type-based flow analysis: From polymorphic subtyping to CFL-reachability," in *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pp. 54–66, ACM, 2001.

[74] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," in *2006 IEEE Symphosium on Security and Privacy*, pp. 258–263, 2006.

[75] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *USENIX-SS'06: Proceedings of the 15th Conference on USENIX Security Symposium*, USENIX Association, 2006.

[76] Y. Minamide, "Static approximation of dynamically generated Web pages," in *WWW*, pp. 432–441, 2005.

[77] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pp. 32–41, ACM, 2007.

[78] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 171–180, ACM, 2008.

[79] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "ANDROMEDA: Accurate and scalable security analysis of web applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, Springer-Verlag, 2013.

[80] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and M. Patrick, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," in *Proceedings of the 35-th ACM SIGPLAN conference on Programming language design and implementation*, PLDI'14, 2014.

[81] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, USENIX Association, 2010.

[82] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *Security 2011, 20st USENIX Security Symposium* (D. Wagner, ed.), USENIX Association, Aug. 2011.

[83] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems," in *Proceedings of the 20th USENIX Conference on Security*, SEC'11, USENIX Association, 2011.

[84] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, 2012.

[85] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock Android smartphones," in *Proceedings of the 19th Network and Distributed System Security Symposium*, NDSS'12, 2012.

[86] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th International Cconference on Mobile Systems, Applications, and Services*, MobiSys '11, ACM, 2011.

[87] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: An automatic system for revealing UI-based trigger conditions in android applications," in *Proceedings of the Second Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, ACM, 2012.

[88] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, 2003.

[89] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pp. 228–241, ACM, 1999.

[90] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, Oct. 2000.

[91] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, "Dr. Android and Mr. Hide: Fine-grained permissions in android applications," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, ACM, 2012.

[92] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, ACM, 2011.

[93] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: Analyzing the Android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, ACM, 2012.

[94] Microsoft Corp., "What's user account control?." http://windows.microsoft.com/en-us/windows-vista/what-is-user-account-control.