NONINTERFERENCE IN EXPRESSIVE LOW-LEVEL LANGUAGES

by

Peter Smith Aldous

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

May 2017

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of Peter Smith Aldous
has been approved by the following supervisory committee members:

Matthew Might, Chair	10 Mar 2017 Date Approved
Matthew Flatt, Member	24 Feb 2017 Date Approved
Eric Mercer, Member	24 Feb 2017 Date Approved
Zvonimir Rakamaric, Member	24 Feb 2017 Date Approved
John Regehr, Member	24 Feb 2017 Date Approved

and by Ross Whitaker, Director of the School of Computing
and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Early work in implicit information flow detection applied only to flat, procedureless languages with structured control-flow (e.g., if statements, while loops). These techniques have yet to be adequately extended and generalized to expressive languages with interprocedural, exceptional, and irregular control-flow behavior. This work presents an implicit information flow analysis suitable for languages with conditional jumps, dynamically dispatched methods, and exceptions. Specifically, this analysis operates on Dalvik bytecode, the substrate for Android.

In order to capture information flows across interprocedural and exceptional boundaries, this analysis uses a projection of a small-step abstract interpreter's rich state graph instead of the control-flow graph typically used for such purposes in weaker linguistic settings. This technique proves termination-insensitive noninterference.

An optimized variant of this analysis performs taint tracking after abstract interpretation instead of combining the two. It does so by removing the additional components in each state and instead performs the same analysis a posteriori. The a posteriori analysis dramatically outperforms its augmented-state counterpart. In addition to improving performance, this independence broadens the applicability of the underlying approach to information-flow analysis.

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

Although my diploma will contain only my name, I would not have finished a doctoral degree—or even begun one—without the contributions of many people.

First, I'm grateful to my parents, who raised me to work hard and who encouraged my curiosity. They taught me to do anything well that was worth doing and insisted on attention to detail and logical rigor. They raised my siblings and me in a home where Standard English was our native tongue and enrolled us in language programs. Both of these decisions helped me immensely as a writer.

As I studied Latin with Keenan Davis in junior high and German with Lili Floyd in high school, grammar finally began to make sense to me. All of the grammatical facts I'd learned finally made sense. The insight I gained into the mechanics of grammar has helped me not just to write but instilled in me a fascination with the inner workings of language. These language studies started me on a path that led to the study of programming languages.

I'm also grateful to people who believed in me and in my potential through the years. Scott and Steve Purves consistently believed that I could do great things. Although they may not have thought much of comments they made to me, their words have remained with me through the years. Similarly, I'm grateful to Kent Pinkerton for hiring me to work in his lab—and then for sincerely wanting me to join him in the sciences when I planned at the time to work in industry. "We'll make a scientist out of you yet," he said. Thanks in part to his encouragement, I have become a scientist.

Friends and colleagues during my undergraduate career also helped me to be my best self. I learned to read the manual from Peter Henderson. The enthusiasm of my love for truth grew because of my association with Seth Stewart. Bill McGreaham listened carefully and gave great advice—and was a model of determination and hard work.

I'm also grateful to Jay McCarthy for mentoring me as I took my first forays into research. Among other things, Jay taught me that uniqueness is an asset in academe.

My colleagues in graduate school were invaluable to me. Chris Earl welcomed me warmly into the lab from the day of my recruiting visit. As we studied together and talked, he became a trusted friend. Other people joined the ranks, both helping me with my studies and becoming valued friends. Among those friends are Steven Lyde and Thomas Gilray, whose help with both research and writing was invaluable to me. I am also grateful for Michael Ballantyne's insight and especially his enthusiasm as he helped me optimize the analyzer.

The question that led to my research was first posed to me by Matthew Might, my adviser. I'm also grateful to him for pushing me to do a little more and a little better. His enthusiasm for research and for communication are infectious and observing his presentations in various situations contributed to my own ability to teach effectively.

I'm also grateful to Jason Hills, T. J. Rakitan, and Brent Kerby for patiently listening to and correcting my attempts create a suitable statistical analysis of the accuracy of my work.

My years in graduate school were among some of the most difficult and discouraging I have experienced. During this time, I had countless family members and friends believe in me and love me when I had trouble doing either for myself. Their support helped me to continue on and, eventually, to find success.

I am blessed to have many more people I could thank for their contributions to my work and to my life. To everyone mentioned here and everyone not mentioned here, I express my gratitude.

CHAPTER 1

THESIS

It is feasible and useful to track implicit information flows soundly in imperative languages with conditional jump statements, virtual methods, and exceptional control flow.

CHAPTER 2

INTRODUCTION

With limited resources available for security, many modern software developers make a rational choice to focus on finding and fixing bugs. Every bug that they fix is a bug that cannot affect users or hurt the developers' brand, so individual bugs are useful intermediate results. In contrast, a partial proof is no proof at all. Given these limitations, it is perfectly reasonable to focus on the identification of bugs—and this focus contributes to software quality.

However, there is always another bug. Despite the fact that humankind has known about overflow bugs for 45 years [1] and despite the fact that many tools have been developed that help programmers to identify and prevent them, buffer overflow vulnerabilities continue to be found in modern software systems. The same is true of many other classes of bugs; they have been known for some time and yet they continue to appear.

Since many developers choose (understandably) not to pursue assurance, end users frequently find themselves without truly secure options for computing. For some users, this is particularly troublesome. For example, administrators of military bases or government offices need particularly strong security guarantees. If personnel can use phones or computers on military bases or in diplomatic buildings and if those devices can be compromised, malware could exfiltrate information that would endanger the people those buildings serve. Malware could also be used for corporate espionage or to compromise the security of nations. In these and other cases, the potential costs incurred by inadequate security are high enough to justify considerable investment in security.

Users who are willing to devote large portions of their resources might choose to perform analyses that provide assurance, even though these analyses may require time and effort from experts to handle false positives. The ideal analysis for these users is automated, operates without cooperation by the software's developers, and proves the absence of the classes of bugs that concern them. This analysis must also be suitable for application to low-level languages, as most source code is not made available to users. Moreover, the analysis must be applicable to *expressive* low-level languages; in other words, it must be able to handle language features like functions and exceptional control flow.

2.1 Contributions

To the end of solving this problem, this dissertation makes the following contributions:

- The design of an original analysis for information flow tracking in expressive low-level languages;
- a proof that the analysis demonstrates noninterference;
- the design of an equivalent, optimized analysis; and
- an empirical evaluation of the performance of both analyses.

These contributions serve as a demonstration that guarantees can be made about information flows in real-world programs, as well as a demonstration of how these guarantees can be obtained.

2.2 Background

This work uses small-step abstract interpretation to model programs' behaviors. Its goal is to prove noninterference, or the property that information does not leak. Section 2.2.1 describes small-step abstract interpretation. Section 2.2.2 describes postdominance, which is used to prove noninterference. Section 2.2.3 explains noninterference.

2.2.1 Small-step abstract interpretation

The CESK [11] evaluation model represents states in an interpreter's execution as tuples of control (C), environment (E), store (S), and continuations (K). Each component captures part of the state of an interpreter at a moment during execution. The control represents where the interpreter is in the program; for example, it may be a line number or label. The environment maps variables to addresses and the store maps addresses to values; this indirection simplifies interpretation in the presence of mutation. The continuation contains the information used to return from a function. In an imperative program, these terms roughly approximate (respectively) the program counter, the frame pointer, the heap, and the stack. Crucially, models like CESK produce states that are self-contained. Being self-contained, the successor ς' to any state ς may be calculated knowing nothing more than the information contained in ς. Concrete, or ordinary, interpretation can be accomplished by calculating the transitive closure of succession from some initial state ς_0.

The semantics of the CESK interpreter use production rules to describe its behavior. Different instructions or forms in the language get different production rules. For example, the `Move` instruction is a summary of the various move instructions in Dalvik bytecode. Each one copies a datum from one register to another. Figure 2.1 contains the concrete semantics of the `Move` instruction. It makes use of a statement lookup metafunction $\mathcal{I}$, a metafunction *next* that retrieves the next code point, and an appropriately defined state space. As the semantics of `Move` demonstrate, control passes from *cp* to the following instruction *next* (*cp*), the frame pointer ϕ is unchanged, the store σ is updated so that the address for the destination register r_d contains the value stored at the address for the source register r_s, and the continuation κ is unchanged. Addresses for registers are pairs of a frame pointer and a register name. Section 3.2.3 contains the semantics for every instruction in a small-step concrete interpreter, including the semantics of the taint tracking mechanism.

Section 2.2.1.1 describes the abstraction of values to guarantee finiteness. Section 2.2.1.2 describes the abstraction of addresses. Section 2.2.1.3 describes the process of abstract state exploration. Finally, Section 2.2.1.4 describes widening, which decreases precision but generally increases speed in abstract interpretation.

2.2.1.1 Abstract values

Van Horn and Might [35] demonstrated that concrete CESK interpreters can be turned into **small step abstract interpreters** by abstracting their state spaces and

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Move(}r_d\texttt{, }r_s\texttt{)}}{\left(cp, \phi, \sigma, \kappa\right) \rightarrow \left(\mathit{next}\left(cp\right), \phi, \sigma', \kappa\right)}, \text{ where}$$

$$\begin{aligned} sa_d &= \left(\phi, r_d\right) \\ sa_s &= \left(\phi, r_s\right) \\ \sigma' &= \sigma[sa_d \mapsto \sigma\left(sa_s\right)] \end{aligned}$$

Figure 2.1: Semantics for a move instruction in a concrete interpreter

by modifying their transition rules. An abstract state space combines concrete values into abstract values in order to guarantee that the state space is finite, which makes a proof of termination possible. In general, the choice of which abstractions to use is left to the analyst. More precise abstractions expand the size of the state space, making the asymptotic complexity of the analysis larger. However, increased precision can diminish the portion of the state space that is explored, leading to faster runtime. For the sake of illustration, this section describes some commonly used abstractions. By convention, variables and sets that describe abstract values are designated with the ˆ diacritic; concrete frame pointers are given names such as ϕ and ϕ' while abstract frame pointers are given names such as $\hat{\phi}$ and $\hat{\phi}'$.

The numeric values used by many programming languages are already finite because of their limited bit width. However, even the modest 32-bit integer or floating point value is frequently precise enough to make abstract interpretation intractable. Because primitive values in many languages have literal representations in syntax, an abstract interpreter generally requires an abstraction metafunction. This section will present several possible abstraction metafunctions for integer values, each of which will have a name of the form $\alpha_x : \mathsf{INT32} \rightarrow \widehat{\mathsf{INT32}}$, where $\widehat{\mathsf{INT32}}$ differs from abstraction to abstraction. Similar strategies can be used for other primitive data types.

One trivial abstraction for numeric values is to discard all precision; in other words, to abstract all integers to $\top$. $\top$ is an abstract value that represents any value; in this case, it represents any integer. This is done with $\alpha_\top : \mathsf{INT32} \rightarrow \{\top\}$, whose definition is straightforward:

$$\alpha_\top(\mathit{int32}) = \top$$

It is worth noting that the literature uses $\top$ inconsistently; some works use $\top$ for an unspecified value and others use $\bot$. These two values are not interchangeable; in fact, they are each other's opposite. However, lattices are defined such that each one is defined and either one may be used for this purpose in an abstract interpretation as long as that usage is consistent. Similarly, this work uses the lattice join operation $\sqcup$. Works that use $\bot$ where this work uses $\top$ use the meet operation $\sqcap$ instead of $\sqcup$.

Another equally trivial abstraction is to discard no precision. While easy to implement, this abstraction rarely allows abstract interpretation to terminate in a reasonable amount of time. Were this "abstraction" to operate on $\mathbb{Z}$, it would produce an infinite set and would allow the interpreter to diverge. This is done with $\alpha_{\mathsf{INT32}} : \mathsf{INT32} \to \mathsf{INT32}$.

$$\alpha_{\mathsf{INT32}}(\mathit{int32}) = \mathit{int32}$$

A much more common abstraction is the abstraction to a value's sign. The abstraction $\alpha_+ : \mathsf{INT32} \to \mathcal{P}(\{-, 0, +\})$ accomplishes this in three cases:

$$\alpha_+(\mathit{int32}) = \begin{cases} \{-\} & \mathit{int32} < 0 \\ \{0\} & \mathit{int32} = 0 \\ \{+\} & \mathit{int32} > 0 \end{cases}$$

Another abstraction is to use a predefined set I of values that are not to be abstracted, while all other values are abstracted with a different strategy. In this example, all values not in I are abstracted to $\top$. $\alpha_I : \mathsf{INT32} \to I \cup \{\top\}$ is defined as follows:

$$\alpha_I(\mathit{int32}) = \begin{cases} \mathit{int32} & \mathit{int32} \in I \\ \top & \mathit{int32} \notin I \end{cases}$$

There is another abstraction which, in some cases, can be problematic. It uses a counter c to determine if a predefined number of primitives has been allocated. If not, it decrements the counter, adds the value to its set of values I that should not be abstracted, and returns the value. Otherwise, it returns $\top$. Another finite set, such as signs, could be used in this second case. $\alpha_c : \mathsf{INT32} \rightarrow \mathsf{INT32} \cup \{\top\}$ requires three slightly more complicated cases. In the second, “adds” refers to a mutating set union.

$$\alpha_c\left(\mathit{int32}\right) = \begin{cases} \mathit{int32} & \mathit{int32} \in I \\ \mathit{dec}\left(c\right); I \text{ adds } \mathit{int32}; \mathit{int32} & c > 0 \\ \top & c = 0 \end{cases}$$

In some cases, stateful abstractions can contribute to nondeterminism in analysis result. This is discussed in Section 7.2.

2.2.1.2 Abstract addresses

In general, abstract addresses are not treated in the same way as abstract primitive values, despite the fact that many languages use 32-bit or 64-bit integer values to store addresses. Also, addresses are not included in syntax but are runtime properties of the interpreter. As such, there is no need to define abstractions for addresses. Instead, addresses are generated with **allocators**, which act like abstracted versions of `malloc`. Unlike `malloc`, whose range is effectively infinite, abstract allocators have much more restricted ranges. At a minimum, small-step abstract interpretation requires *allock*, an allocator for continuation addresses. Any other allocation modeled in the language, such as array or object instantiation, also requires an allocator. These allocation strategies may be identical; indeed, they may all use the same metafunction *alloc*. They may also be completely disparate. In this section, *alloc* may be used for any type of allocation.

As with primitive value abstractions, several species of allocators may be used. One trivial abstraction discards all precision: $\mathit{alloc}_\top : \hat{\Sigma} \rightarrow \{\top\}$

$$\mathit{alloc}_\top\left(\hat{\varsigma}\right) = \top$$

It is possible to use an allocation strategy that uses the entire allocating state as an address. However, states contain addresses. If addresses contain states, abstract interpretation can diverge. So although $\mathit{alloc}_{\hat{\varsigma}} : \hat{\Sigma} \rightarrow \hat{\Sigma}$ is possible to define, it fails to guarantee finiteness:

$$\mathit{alloc}_{\hat{\varsigma}}\left(\hat{\varsigma}\right) = \hat{\varsigma}$$

A much more common allocation strategy is called 0CFA, which uses the currently executing function as an address. $alloc_{0CFA} : \hat{\Sigma} \to \mathsf{Method}$ is defined as follows:

$$alloc_{0CFA}(\hat{\varsigma}) = m, \quad \text{where } \hat{\varsigma} = \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \text{ and } cp = (ln, m)$$

0CFA is the least precise of the family of allocation strategies called k-CFA. The more precise variants, like 1CFA and 2CFA, require states to contain additional information. Specifically, they must contain information about the last function to be called. Notably, there is some ambiguity, even in the original k-CFA paper: this can be the last function to be called or the function that called the current function. A 1CFA allocator $alloc_{1CFA} : \hat{\Sigma} \to \mathsf{Method} \times \mathsf{Method}$ is defined:

$$alloc_{1CFA}(\hat{\varsigma}) = (m, m_c), \quad \text{where } \hat{\varsigma} = \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}, m_c\right) \text{ and } cp = (ln, m)$$

Similar to 0CFA is the pointwise allocator, which uses the program location as an address. $alloc_{cp} : \hat{\Sigma} \to CodePoint$ is simply:

$$alloc_{cp}(\hat{\varsigma}) = cp, \quad \text{where } \hat{\varsigma} = \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right)$$

As long as finiteness is preserved, arbitrary additional information may be added to states in order to tune allocators [13].

Of note is pushdown control flow analysis, or PDCFA [9]. Instead of guaranteeing a finite state space, PDCFA guarantees that a finite portion of its infinite state space will be explored. PDCFA guarantees perfect stack precision; that is, that an abstract interpreter will not return to a location that did not call the returning function. PCDFA uses a different structure from the family of small-step abstract analyses described here.

Recent work by Gilray et al. [14] showed that perfect stack precision may be obtained with an allocator. Their allocation strategy is called pushdown control flow analysis for free or P4F. P4F uses the fact that continuation addresses are only ever created upon function invocation. It requires the program location and environment *of the state being created.* Accordingly, $alloc_{P4F} : CodePoint \times FP \to CodePoint \times FP$ is defined as follows:

$$alloc_{P4F}(cp', \phi') = (cp', \phi')$$

Allocated components in an abstract interpreter may already be in use because they come from finite sets. For example, the set of abstract frame pointers might be defined as the set of program locations. New frame pointers are generated upon function invocation, so new frame pointers could be generated with the code point where the function is invoked. With this allocation strategy, recursive calls to the same function would receive the same abstract frame pointer. This phenomenon is called **merging**.

2.2.1.3 Abstract state exploration

Abstracting values leads to situations where the interpreter may follow multiple paths; for example, a branch whose condition is $\top$ cannot prove that its condition is exclusively true or false. In this case, the abstract interpreter follows both paths. The self-contained nature of CESK states allows interpretation to proceed along both paths independently by simply adding each successor to a queue. States may be enqueued in any order. Since states may have multiple successors, an abstract CESK interpreter does not produce a linear program trace, but instead produces a graph of abstract states that model all possible executions from a given initial abstract state.

The abstract store is updated weakly; that is, new values are combined with existing values. This is necessary because of merging in the address space; two (or more) unrelated instructions that use two different concrete addresses may use the same abstract address, so an update to one value must not interfere with the other. Abstract values are defined on a lattice, so the lattice's join operation $\sqcup$ is used to combine them; in many cases, this is simply set union. Weak updates guarantee monotonicity, which contributes to the proof of termination.

As their name suggests, small-step abstract interpreters proceed in small steps. Their semantics are derived from those of analogous small-step concrete interpreters. Like their concrete analogues, abstract interpreters use production rules to specify behavior for each instruction or syntactic form. The production rule for the `Move` instruction in a small-step abstract interpreter is included in Figure 2.2. Section 3.2.4.2 contains abstract semantics for every instruction with taint tracking.

Abstraction makes it impossible for an interpreter to exhibit only those program behaviors that can occur during concrete interpretation. This means that abstract

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Move(}r_d\texttt{, }r_s\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \to \left\{\left(\mathit{next}\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)\right\}}, \text{ where}$$

$$\begin{aligned} \widehat{sa}_d &= \left(\hat{\phi}, r_d\right) \\ \widehat{sa}_s &= \left(\hat{\phi}, r_s\right) \\ \hat{\sigma}' &= \hat{\sigma}[\widehat{sa}_d \mapsto \hat{\sigma}\left(\widehat{sa}_d\right) \sqcup \hat{\sigma}\left(\widehat{sa}_s\right)] \end{aligned}$$

Figure 2.2: Semantics for a move instruction in an abstract interpreter

interpreters are, in one sense, incorrect. We say that they do not have perfect **precision**. They are, however, correct in another sense: they model all possible behaviors in all corresponding concrete interpretations. This correctness property is called **soundness**. Typically, soundness is proven via simulation (illustrated in Figure 2.3). Simulation proofs show that abstract interpretation simulates concrete interpretation by showing that the relationship between a concrete state ς and its abstraction $\hat{\varsigma}$ holds for their respective successors.

An abstraction relation $\alpha \subseteq \Sigma \times \hat{\Sigma}$ formalizes the relationship between ς and its abstraction $\hat{\varsigma}$, as follows: $\alpha\left(\varsigma, \hat{\varsigma}\right)$. The concrete and abstract succession relations ($\to \subseteq \Sigma \times \Sigma$ and $\leadsto \subseteq \hat{\Sigma} \times \hat{\Sigma}$, respectively) indicate that one state succeeds another: $\to \left(\varsigma, \varsigma'\right)$ and $\leadsto \left(\hat{\varsigma}, \hat{\varsigma}'\right)$. With these relations, simulation can be defined: If ς' is the concrete successor to ς, there must be some abstract successor $\hat{\varsigma}'$ to $\hat{\varsigma}$ such that $\alpha\left(\varsigma', \hat{\varsigma}'\right)$. Given some initial concrete state ς_0 and an initial abstract state $\hat{\varsigma}_0$ whose abstraction includes ς_0 and a proof of this inductive property, we can conclude that the abstract state graph includes all possible behaviors that the concrete trace could exhibit.

$$\begin{tikzcd}
\varsigma_0 \arrow[r, "\to"] \arrow[d, "\alpha"] & \ldots \arrow[r, "\to"] & \varsigma \arrow[r, "\to"] \arrow[d, "\alpha"] & \varsigma' \arrow[r, "\to"] \arrow[d, "\alpha"] & \ldots \\
\hat{\varsigma}_0 \arrow[r, "\leadsto"] & \ldots \arrow[r, "\leadsto"] & \hat{\varsigma} \arrow[r, "\leadsto"] & \hat{\varsigma}' \arrow[r, "\leadsto"] & \ldots
\end{tikzcd}$$

Figure 2.3: An illustration of the simulation property

More formally, the inductive property states that if $\rightarrow (\varsigma, \varsigma')$ and $\alpha(\varsigma, \hat{\varsigma})$ then there exists an abstract state $\hat{\varsigma}'$ such that $\rightsquigarrow (\hat{\varsigma}, \hat{\varsigma}')$ and $\alpha(\varsigma', \hat{\varsigma}')$.

2.2.1.4 Widening

Many abstract interpreters relax precision to improve performance by **widening**, or allowing multiple states to share the same component. Small-step abstract interpreters commonly widen their stores. When stores are widened, states no longer contain stores. Instead, there is a way to find the appropriate store for a state. In global widening, all states share the same store. In pointwise widening, all states at the same program location share the same store.

Widening changes the fixed point state exploration slightly; states are only considered previously visited if they have been visited more recently than their respective store has been updated. Typically, timestamps are used to determine whether or not a state has been visited since its store has been updated.

2.2.2 Postdominance

The concept of postdominance (sometimes spelled “post-dominance” in other publications) is defined in a directed graph with a unique exit node. Directed graphs may not have a unique exit node. This is illustrated in Figure 2.4, with exit nodes depicted as squares. In this case, an exit node may safely be added as a successor to each of the multiple exit nodes, as in Figure 2.5. As such, this work refers to directed graphs and assumes without loss of generalization that a unique exit node exists. It further assumes that the exit node is reachable from all nodes; while not true of graphs in general, it is true of the graphs used in this work because of how the graphs are generated. This assumption is somewhat analogous to the assumption that the entry node dominates all other nodes in a control flow graph because the graph is generated by reaching program locations after starting from the entry node.

Postdominance refers to the property that all paths from a postdominated node to the unique exit node pass through all postdominating nodes. In Figure 2.6, G postdominates A; in other words, every path from A to J passes through G, even though there are infinitely many paths from A to J. Although there are infinitely many such paths, each of them has finite length. The same is true of nodes B through

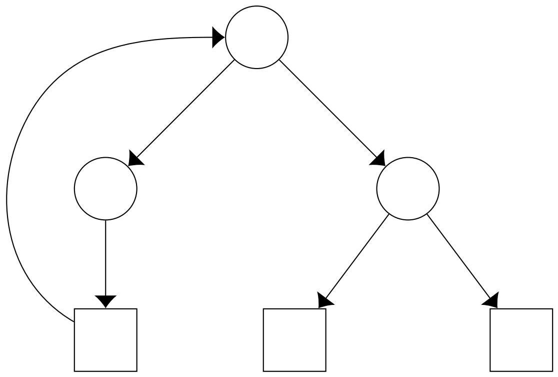

Figure 2.4: A directed graph with no unique exit node

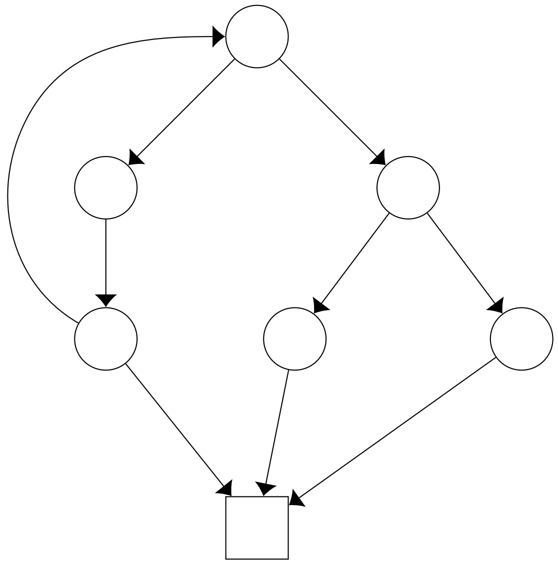

Figure 2.5: The graph from Figure 2.4 with a unique exit node

F; each is postdominated by G. All nodes except the exit node J are postdominated by J. When examining a directed graph, postdominators look like constriction points where paths converge, including the trivial case where a node is postdominated by its unique child.

Dominance is the property that all paths from a unique entry node to a dominated node pass through all dominating nodes. Inverting the graph, including turning the exit node into an entry node, turns a postdominance problem into a dominance problem. This isomorphism is particularly useful because the literature addresses dominance much more frequently than postdominance. In accordance with this isomorphism, this work references the literature about dominance as if it addressed postdominance directly.

Every node except the exit node in a directed graph is postdominated by the exit node. Nodes may also be postdominated by additional nodes. Each node has a unique immediate postdominator, which is the postdominator that does not postdominate any other of the node's postdominators. Intuitively, an immediate postdominator P of a node N is the first of the postdominators of N along a path from N to the exit node.

The uniqueness of a node's immediate postdominator is demonstrable by contradiction. Assume for contradiction that some node W has at least two immediate postdominators X and Y. By the definition of immediate postdominator, neither postdominates the other. It is already known that the exit node Z postdominates all other nodes. This relationship is illustrated in Figure 2.7; solid arrows indicate postdominance, while dashed arrows indicate the absence of a postdominance relation. Because postdominance is transitive, an arrow between W and Z could be shown but is omitted for clarity.

There must be some path from W to X that does not include Y because X does not postdominate Y (but X does postdominate W). Similarly, there must be some path from X to Z that does not include Y because Y does not postdominate X. As such, there must be a path from W through X to Z that does not include Y, which contradicts the given that Y postdominates W.

Algorithms for the calculation of immediate dominators (and, therefore, immediate

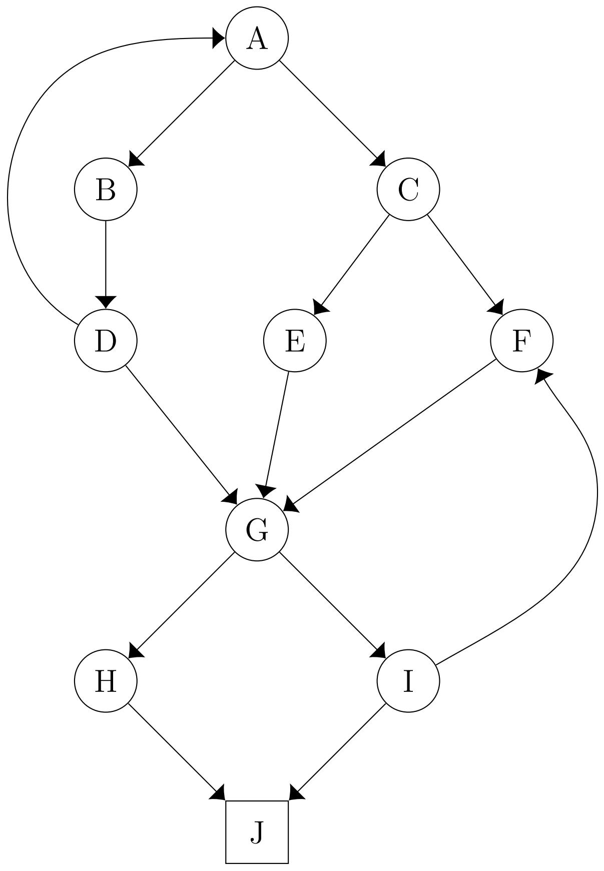

Figure 2.6: A directed graph

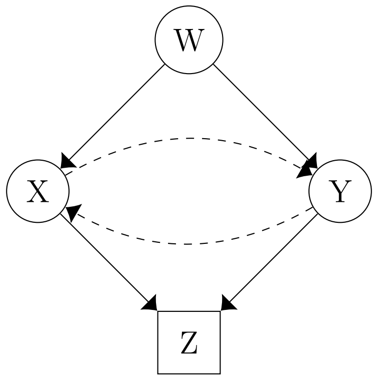

Figure 2.7: Postdominance relationships in the proof by contradiction

postdominators) exist in the literature; for example, the algorithm presented by Lengauer and Tarjan [22].

2.2.3 Noninterference

Traditional **taint tracking** mechanisms apply a security type or label, also called a taint, to sensitive values, such as a phone's location or a user's password. Whenever a new value is written, it derives its security type from the values upon which it depends. Denning demonstrated that security types may be lattices [7]. In practice, many analyses use simple binary labels: the "high" label applies to values with sensitive information and the "low" label applies to other values.

These techniques are effective for **explicit** information flows, in which values propagate directly (e.g., via assignment). However, they fail to detect **implicit** information flows, which depend on control flow to leak information. Figure 2.8 shows a Java snippet that prints `true` when `secret` is true and `false` otherwise. This is an implicit information leak because `secret` affects what is printed but is not copied or used directly. `switch` statements, function calls, function returns, and exceptional control flow can also change control flow. As such, these control flows can also change values in ways not detectable by traditional taint tracking mechanisms.

In order to track implicit information flows, taints can also be applied to the program's context, per Denning and Denning [8]. Denning and Denning further claim that a static analysis of postdominance in the control flow graph would allow context tainting to apply to languages with arbitrary `goto` statements. In the case of Figure 2.8, line 5 postdominates line 1, so the `println` call on line 6 is safe. However, Denning and Denning do not prove noninterference. Furthermore, their analysis does not include function calls or exceptional control flow.

Figure 2.9 contains a program that successfully leaks a bit without detection by postdominance in the (interprocedural) control flow graph. It does so by creating a situation where control returns from a function when a sensitive value is true. As a result, control flow reaches line 11 regardless of the sensitive value but the level of the stack reflects that value. As such, the value of `secret` is printed to the terminal without detection by the proposed analysis.

Noninterference is a formalization used to demonstrate that information cannot

```
if (secret) {
  System.out.println("true");
} else {
  System.out.println("false");
}
System.out.println("safe");
```

Figure 2.8: An implicit information leak

```
private boolean secret;

void printSecret(int frame) {
  if (frame == 0) {
    printSecret(1);
  } else {
    if (secret) {
      return;
    }
  }
  if (frame == 1) {
    System.out.print("not ");
    return;
  }

  System.out.println("true");
}
```

Figure 2.9: An information leak through the stack

leak. More formally, noninterference is the property that if a program is executed twice with different sensitive data but otherwise identical inputs, an attacker must observe identical behaviors. Both of the Java snippets in this section do not preserve noninterference; in both cases, the output to the terminal reflects the value of `secret`.

This work proves **termination-insensitive noninterference**, a weaker form of noninterference that allows attackers to observe whether or not a program terminates, even if some information can be inferred from the program's convergence or divergence. Termination analyses are well understood and are beyond the scope of this work. In the spirit of termination-insensitive noninterference, exceptional control flow that propagates to the top level is not included in the postdominance calculation. Since some programs do not satisfy the requirement of noninterference, the proof of noninterference is a proof that any interference that can occur will be identified.

CHAPTER 3

AUGMENTED-STATE INFORMATION FLOW TRACKING

Small step abstract interpretation can be modified by the addition of components to the state space. This has the advantage of coupling the operational semantics of the language with the propagation of information flows, which results in a straightforward presentation of a proof of noninterference.

Leaks such as the one demonstrated in Figure 2.9 depend on the stack to mask the information flow. In order to address these leaks, the analysis proposed by Denning and Denning can be modified to calculate postdominance the **execution point graph**, which is similar to but richer than than the control flow graph. Nodes in an execution point graph (called **execution points**) are pairs of a code point and a natural number, which is the depth of the stack. An abstract execution point has either a natural number or a value that represents an indeterminate stack height, which can occur in any finite abstraction of a state space. Execution points without exact stack heights are not considered postdominators, as they represent multiple concrete execution points.

This work uses implicit taint values to defer computation about whether or not a taint should propagate until after the execution point graph can be created. Section 3.1 explains in more detail why implicit taint values exist and how they are constructed. Section 3.2 presents a language that summarizes the features of Dalvik bytecode, gives semantics for the language, and describes the abstraction of this dynamic analysis to a static analysis. Section 3.3 proves that this analysis demonstrates termination-insensitive noninterference.

3.1 Implicit taint values

Augmented-state information flow tracking requires special implicit taint values distinct from typical taint values. Implicit taint values act as deferred taints. Once the abstract interpretation is complete, the execution point graph can be consulted to determine if they are **valid**. Valid taints are those that, in retrospect, should have been propagated. In contrast, invalid taints are taints that can be safely removed. Taint validity is formalized in Section 3.3.4. Implicit taint values are generated from context and are formalized in Section 3.2.2.

Section 3.1.1 describes the structure of implicit taint values and Section 3.1.2 describes how implicit taint values are created from context taint.

3.1.1 Implicit taint value structure

In its simplest (but incomplete) form, an implicit taint value contains a location at which a branch occurred, a subsequent location at which an assignment occurred, and a taint value. If the postdominance calculation on the execution point graph shows that the assignment occurs independently of the branch, the taint value is invalid and can be removed. If, on the other hand, the branch influences whether or not the assignment will execute, the implicit taint value is valid and remains. Optionally, a valid implicit taint may be replaced with the taint value it contains.

However, a recursive structure is cumbersome and potentially infinite in size. This recursion is manifest in the analysis of programs like the one in Figure 3.1. In this example, execution points (formalized in Section 3.2.2) are constructed with the stack height h. If some explicit taint value etv_r exists on `secret`, then an implicit taint value is created for `x`. Since the branch in question occurred at line 2 and the assignment in question occurs on line 3, the implicit taint value is $((2, h), (3, h), etv_r)$. The analysis applies this implicit taint value to `x`.

Subsequently, the analysis branches at line 5 and then assigns at line 6. Since `x` has a taint value, a new implicit taint value is created that includes it and applies it to `y`. The nested implicit taint value is the following:

$$\Big((5, h), (6, h), \big((2, h), (3, h), etv_r\big)\Big)$$

```
1   boolean x = false;
2   if (secret) {
3     x = true;
4   }
5   boolean y = false;
6   if (x) {
7     y = true;
8   }
```

Figure 3.1: Chained implicit taint values

Both of these branch/assignment pairs are necessary for precision; if either assignment occurs independently of its respective branch, the taint is invalid and may be discarded. Their order, however, is unimportant, and duplicate values serve no purpose. As such, the branch/assignment pairs are accumulated in a set and the set is paired with a single explicit taint value. Instead of the recursive value given above, the implicit taint value assigned to `y` is instead:

$$(\{((5,h),(6,h)),((2,h),(3,h))\},etv_r)$$

With this finite structure, validity of an implicit taint value is ascertained by determining whether each assignment executes conditionally upon its respective branch.

3.1.2 Creation of implicit taint values

In order to create implicit taint values, it is necessary to know which branches influence control flow to various parts of a program. Each state contains a context taint map *ct* that shows which taint values were applied to values that affected control flow at execution points. Context taint maps are formally defined in Section 3.2.2.

Since the execution point graph can only be created after abstract interpretation is complete, it is unsafe to remove taints from context during abstract interpretation. Accordingly, all updates to a context taint map are weak. Weak updates to the context taint map guarantee that no context taint is removed during interpretation.

When the context taint map is nonempty, assignments create implicit taint values from the context taint map and from the current state's execution point. Every branch whose condition has taint applied to it creates a context taint that persists for the duration of abstract interpretation. Many of these context taints, however, can be removed in retrospect once the execution point graph has been constructed.

Formal details of context taint and implicit taint values are included in Section 3.2.3.

3.2 Language and semantics

3.2.1 Syntax

The abstract syntax for the summarized bytecode language is given in Figure 3.2. See Section 5.1 for the differences between this language and Dalvik bytecode.

In conjunction with this syntax, following metafunctions are necessary:

- $\mathcal{M} : \mathsf{MName} \to \mathsf{Method}$ for method lookup
- $\mathcal{I} : \mathit{CodePoint} \to \mathsf{Stmt}$ for statement lookup
- $\mathit{next} : \mathit{CodePoint} \rightharpoonup \mathit{CodePoint}$ gives the syntactic successor to the current code point
- $\mathcal{H} : \mathit{CodePoint} \rightharpoonup \mathit{CodePoint}$ gives the target of the first exception handler defined for a code point in the current function, if there is any.
- $\mathit{init} : \mathsf{Method} \to \mathit{CodePoint}$ gives the first code point in a method.
- $\mathit{jump} : \mathit{CodePoint} \times \mathsf{LineNumber} \rightharpoonup \mathit{CodePoint}$ gives the code point in the same method as the given code point and at the line number specified.

3.2.2 State space

A state $\varsigma \in \Sigma$ contains six members:

1. A code point cp.
2. A frame pointer ϕ. All registers in the machine are frame-local. Frame addresses are represented as a pair consisting of a frame pointer and an index.
3. A store σ, which is a partial map from addresses to values.
4. A continuation κ.

$$
\begin{aligned}
\mathit{prgm} \in \mathsf{Program} &= \mathsf{ClassDef}^{*} \\
\mathit{classdef} \in \mathsf{ClassDef} &::= \texttt{Class}\ \mathit{className}\ \{\mathit{field}_1, \ldots, \mathit{field}_n, m_1, \ \ldots, m_m\} \\
m \in \mathsf{Method} &::= \texttt{Def}\ \mathit{mName}\ \{\mathit{handler}_1, \ \ldots, \mathit{handler}_n, \\
&\qquad \mathit{stmt}_1, \ \ldots, \ \mathit{stmt}_m\} \\
\mathit{handler} \in \mathsf{Handler} &::= \texttt{Catch}(\mathit{ln}, \ \mathit{ln}, \ \mathit{ln}) \\
\mathit{stmt} \in \mathsf{Stmt} &::= \ \mathit{ln}\ \texttt{Const}(r, \ c) \\
&\mid \ \mathit{ln}\ \texttt{Move}(r, \ r) \\
&\mid \ \mathit{ln}\ \texttt{Invoke}(\mathit{mName}, \ r_1, \ \ldots, \ r_n) \\
&\mid \ \mathit{ln}\ \texttt{Return}(r) \\
&\mid \ \mathit{ln}\ \texttt{IfEqz}(r, \ \mathit{ln}) \\
&\mid \ \mathit{ln}\ \texttt{Add}(r, \ r, \ r) \\
&\mid \ \mathit{ln}\ \texttt{NewInstance}(r, \ \mathit{className}) \\
&\mid \ \mathit{ln}\ \texttt{Throw}(r) \\
&\mid \ \mathit{ln}\ \texttt{IGet}(r, \ r, \ \mathit{field}) \\
&\mid \ \mathit{ln}\ \texttt{IPut}(r, \ r, \ \mathit{field}) \\
r \in \mathsf{Register} &= \{\texttt{result}, \texttt{exception}, 1, 2, \ldots\} \\
\mathit{ln} \in \mathsf{LineNumber} &\ \text{is a set of line numbers} \\
\mathit{mName} \in \mathsf{MName} &\ \text{is a set of method names} \\
\mathit{field} \in \mathsf{Field} &\ \text{is a set of field names} \\
\mathit{cp} \in \mathit{CodePoint} &::= (\mathit{ln}, m)
\end{aligned}
$$

Figure 3.2: Abstract syntax

5. A taint store ts, which is a partial map from addresses to taint values. Explicit taint values store the execution point at which they originated, while implicit taint values also contain a set of pairs of execution points; the first execution point of each pair indicates where a branch occurred and the other indicates where an assignment happened. The taint store is updated in parallel with the store. When the taint store is read at an undefined address, it returns the empty set.

6. A context taint map ct, which is a map from execution points to taint values. The execution points are where control-flow has branched before reaching the current state. The taint values indicate the taints on the values that contributed to the branching.

The concrete state space is formally defined in Figure 3.3.

3.2.3 Semantics

Section 3.2.3.1 defines metafunctions and shorthand notations used in the definition of the formal semantics. Section 3.2.3.2 uses these notations to define the semantics of the concrete interpreter.

3.2.3.1 Helpers for concrete semantics

These semantics require projection metafunctions. $SH : Kont \to \mathbb{Z}$ calculates stack height:

$$SH\,(\kappa) = \begin{cases} 1 + SH\,(\kappa') & \text{if } \kappa = \mathbf{retk}(cp, \phi, ct, \kappa') \\ 0 & \text{if } \kappa = \mathbf{halt} \end{cases}$$

$p : \Sigma \to EP$ uses SH to create execution points:

$$p\,(\varsigma) = \begin{cases} \mathbf{ep}\,(cp, z) & \text{if } \varsigma = (cp, \phi, \sigma, \kappa, ts, ct) \text{ and } z = SH\,(\kappa) \\ \mathbf{endsummary} & \text{if } \varsigma = \mathbf{endstate} \\ \mathbf{errorsummary} & \text{if } \varsigma = \mathbf{errorstate} \end{cases}$$

In some cases, an implicit taint value may exist inside of the context taint map. In this case, the set of branch/assignment pairs is extended to form a single implicit taint value. Combining them ensures that the implicit taint value space is finite.

$$
\begin{aligned}
\varsigma \in \Sigma &::= (cp, \phi, \sigma, \kappa, ts, ct) \mid \mathbf{errorstate} \mid \mathbf{endstate} \\
\phi \in \mathit{FP} & \text{ is an infinite set of frame pointers} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightarrow \mathit{Value} \\
\mathit{val} \in \mathit{Value} &= \mathsf{INT32} + \mathit{ObjectAddress} \\
\kappa \in \mathit{Kont} &::= \mathbf{retk}(cp, \phi, ct, \kappa) \mid \mathbf{halt} \\
ts \in \mathit{TaintStore} &= \mathit{Addr} \rightarrow \mathcal{P}\left(\mathit{TaintValue}\right) \\
tv \in \mathit{TaintValue} &= \mathit{ExplicitTV} + \mathit{ImplicitTV} \\
etv \in \mathit{ExplicitTV} &= \mathit{EP} \\
itv \in \mathit{ImplicitTV} &= \mathcal{P}\left(\mathit{EP} \times \mathit{EP}\right) \times \mathit{ExplicitTV} \\
ct \in \mathit{ContextTaint} &= \mathit{EP} \rightarrow \mathcal{P}\left(\mathit{TaintValue}\right) \\
ep \in \mathit{EP} &::= \mathbf{ep}\left(cp, z\right) \mid \mathbf{errorsummary} \mid \mathbf{endsummary} \\
z \in \mathbb{Z} & \text{ is the set of integers} \\
a \in \mathit{Addr} &::= sa \mid fa \mid oa \mid \texttt{null} \\
sa \in \mathit{StackAddress} &= \mathit{FP} \times \mathsf{Register} \\
fa \in \mathit{FieldAddress} &= \mathit{ObjectAddress} \times \mathsf{Field} \\
oa \in \mathit{ObjectAddress} & \text{ is an infinite set of addresses}
\end{aligned}
$$

Figure 3.3: Concrete state space

Accordingly, new implicit taint values are created with the metafunction $\mathit{implicit} : \mathit{EP} \times \mathit{EP} \times \mathit{TaintValue} \rightarrow \mathit{ImplicitTV}$, which is defined in two cases:

$$\mathit{implicit}\left(ep_b, ep_a, tv\right) = \begin{cases} \left(\left\{\left(ep_b, ep_a\right)\right\}, tv\right) & \text{if } tv \in \mathit{ExplicitTV} \\ \left(\left\{\left(ep_b, ep_a\right)\right\} \cup B, etv_i\right) & \text{if } tv = \left(B, etv_i\right) . \end{cases}$$

The concrete semantics for the language are defined by the relation $(\rightarrow) \subseteq \Sigma \times \Sigma$. In its transition rules, the following shorthand is useful: ep_ς is a state's execution point and itv_ς is the set of implicit taint values generated at a state. For a state ς with context taint map $ct = [ep_1 \rightarrow \{tv_{1,1}, \ldots, tv_{1,i}\}, \ldots, ep_n \rightarrow \{tv_{n,1}, \ldots, tv_{n,j}\}]$,

$$ep_\varsigma = p\left(\varsigma\right)$$

and

$$\begin{aligned} itv_\varsigma = \{&\mathit{implicit}\left(ep_1, ep_\varsigma, tv_{1,1}\right), \ldots, \mathit{implicit}\left(ep_1, ep_\varsigma, tv_{1,i}\right), \ldots, \\ &\mathit{implicit}\left(ep_n, ep_\varsigma, tv_{n,1}\right), \ldots, \mathit{implicit}\left(ep_n, ep_\varsigma, tv_{n,j}\right)\} . \end{aligned}$$

Updates to the context taint map are always **weak**. That is, values are only ever added, leaving existing values in place. It becomes unwieldy to write updates of the form

$$ct' = ct\left[ep \rightarrow ct\left(ep\right) \cup \{tv\}\right] .$$

Instead, this work abbreviates weak updates with the following shorthand, which is equivalent to the long form above:

$$ct' = ct\left[ep \overset{\sqcup}{\mapsto} \{tv\}\right] .$$

It is also convenient to merge context taint maps together. Merging context taint maps involves finding execution points mapped to taints in both maps and combining the sets of taints. In the event that only one context taint map contains a mapping for a given execution point, that mapping is preserved. For example, if $ct_1 = [ep_1 \rightarrow \{tv_1\}, ep_2 \rightarrow \{tv_2\}]$ and $ct_2 = [ep_1 \rightarrow \{tv_3\}]$,

$$ct_1 \sqcup ct_2 = [ep_1 \rightarrow \{tv_1, tv_3\}, ep_2 \rightarrow \{tv_2\}] .$$

3.2.3.2 Concrete semantics

The `Const` instruction writes a constant value to a register. Destination addresses of a constant assignment cannot be tainted explicitly but may be tainted implicitly via context. It proceeds to the next code point and leaves the frame pointer, continuation stack, and context taint unchanged but updates the store so that the stack address sa contains the constant c and the taint store so that it contains any relevant taint from context.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Const(}r\texttt{, }c\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \to \left(next\left(cp\right), \phi, \sigma', \kappa, ts', ct\right)}, \text{ where}$$

$$\begin{aligned} sa &= \left(\phi, r\right) \\ \sigma' &= \sigma\left[sa \mapsto c\right] \\ ts' &= ts\left[sa \mapsto itv_{\varsigma}\right] . \end{aligned}$$

The `Move` instruction simulates all of Dalvik bytecode's move instructions. Taints may propagate both explicitly from the source address and implicitly via context. This instruction varies from the `Const` instruction only in that it copies the value stored at sa_d instead of a constant.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Move(}r_d\texttt{, }r_s\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \to \left(next\left(cp\right), \phi, \sigma', \kappa, ts', ct\right)}, \text{ where}$$

$$\begin{aligned} sa_d &= \left(\phi, r_d\right) \\ sa_s &= \left(\phi, r_s\right) \\ \sigma' &= \sigma\left[sa_d \mapsto \sigma\left(sa_s\right)\right] \\ ts' &= ts\left[sa_d \mapsto ts\left(sa_s\right) \cup itv_{\varsigma}\right] . \end{aligned}$$

The `Invoke` instruction simulates Dalvik's invoke instructions. See Section 5.1.3 for a discussion of virtual method resolution, calling convention, etc. in Dalvik bytecode. In Dalvik bytecode, the receiver of the method, if there is one, is in the first register. In this summarized bytecode language, that first argument is r_1. Although these semantics do not show virtual dispatch, they do show context tainting as in the presence of virtual dispatch. To do so, context taint is created from any taint values at the stack address sa_{s1} of the first argument and at the object address, if there is one, stored at sa_{s1}. The updates to the store copy the values of each argument to the new frame pointer.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Invoke(}mName\texttt{, } r_1\texttt{, } \ldots\texttt{, } r_n\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \to \left(cp', \phi', \sigma', \kappa', ts', ct'\right)}, \text{ where}$$

$$\begin{aligned}
cp' &= init\left(\mathcal{M}\left(mName\right)\right) \\
\kappa' &= \mathbf{retk}(cp, \phi, ct, \kappa) \\
\phi' &= \text{ is a fresh frame pointer} \\
&\quad \text{for each } i \text{ from 1 to } n, \\
sa_{di} &= (\phi', i) \text{ and } sa_{si} = (\phi, r_i) \\
\sigma' &= \sigma\left[sa_{d1} \mapsto \sigma\left(sa_{s1}\right), \ldots, sa_{dn} \mapsto \sigma\left(sa_{sn}\right)\right] \\
ts' &= ts\left[sa_{d1} \mapsto ts\left(sa_{s1}\right) \cup itv_{\varsigma}, \ldots, sa_{dn} \mapsto ts\left(sa_{sn}\right) \cup itv_{\varsigma}\right] \\
ct' &= ct\left[ep_{\varsigma} \overset{\sqcup}{\mapsto} T\right] \\
T &= ts\left(sa_{s1}\right) \cup \left(\left\{\sigma\left(sa_{s1}\right)\right\} \cap ObjectAddress\right) .
\end{aligned}$$

The `Return` instruction summarizes Dalvik's return instructions. The `Return` instruction introduces context taint if invocation occurred in a tainted context. In this sense, it acts as a branch.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Return(}r\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \to \left(next\left(cp'\right), \phi', \sigma', \kappa', ts', ct'\right)}, \text{ where}$$

$$\begin{aligned}
\kappa &= \mathbf{retk}(cp, \phi', ct_k, \kappa') \\
sa_d &= (\phi', \texttt{result}) \\
sa_s &= (\phi, r) \\
\sigma' &= \sigma\left[sa_d \mapsto \sigma\left(sa_s\right)\right] \\
ts' &= ts\left[sa_d \mapsto ts\left(sa_s\right) \cup itv_{\varsigma}\right] \\
ct' &= ct \sqcup ct_k .
\end{aligned}$$

This second rule for the `Return` instruction finishes execution when the **halt** continuation is invoked.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Return(}r\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \mathbf{endstate}}, \text{ where}$$

$$\kappa = \mathbf{halt}\,.$$

The `IfEqz` instruction jumps to the given target if its argument is 0 and continues to the next instruction otherwise. Dalvik bytecode represents `null` as 0, so `IfEqz` serves as a null check.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IfEqz(}r\texttt{, }ln\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \left(cp', \phi, \sigma, \kappa, ts, ct'\right)}, \text{ where}$$

$$\begin{aligned} sa_s &= \left(\phi, r\right) \\ cp' &= \begin{cases} jump\left(cp, ln\right) & \text{if } \sigma\left(sa_s\right) = 0 \\ next\left(cp\right) & \text{if } \sigma\left(sa_s\right) \neq 0 \end{cases} \\ ct' &= ct\left[ep_\varsigma \overset{\sqcup}{\mapsto} ts\left(sa_s\right)\right]\,. \end{aligned}$$

The `Add` instruction represents all arithmetic instructions. Since Java uses 32-bit two's complement integers, + represents 32-bit two's complement addition.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Add(}r_d\texttt{, }r_l\texttt{, }r_r\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \left(next\left(cp\right), \phi, \sigma', \kappa, ts', ct\right)}, \text{ where}$$

$$\begin{aligned} sa_d &= \left(\phi, r_d\right) \\ sa_l &= \left(\phi, r_l\right) \\ sa_r &= \left(\phi, r_r\right) \\ \sigma' &= \sigma\left[sa_d \mapsto \sigma\left(sa_l\right) + \sigma\left(sa_r\right)\right] \\ ts' &= ts\left[sa_d \mapsto ts\left(sa_l\right) \cup ts\left(sa_r\right) \cup itv_\varsigma\right]\,. \end{aligned}$$

Object instantiation is done with the `NewInstance` instruction. Calls to constructors are included explicitly in bytecode.

$$\frac{\mathcal{I}\left(cp\right)=\texttt{NewInstance(}r\texttt{, }className\texttt{)}}{\left(cp,\phi,\sigma,\kappa,ts,ct\right)\rightarrow\left(next\left(cp\right),\phi,\sigma',\kappa,ts',ct\right)}, \text{ where}$$

$$\begin{aligned}
&oa \text{ is a fresh object address}\\
&sa=\left(\phi,r\right)\\
&\sigma'=\sigma\left[sa\mapsto oa\right]\\
&ts'=ts\left[sa\mapsto itv_{\varsigma}\right].
\end{aligned}$$

The remaining instructions use an additional metafunction:

$$\begin{aligned}
\mathcal{T}:\ &CodePoint\times FP\times ContextTaint\times Kont\rightharpoonup\\
&CodePoint\times FP\times ContextTaint\times Kont\,.
\end{aligned}$$

$\mathcal{T}$ looks for an exception handler in the current function. If there is a handler, it is returned. If not, searches through the code points in the continuation stack. The accumulation of context taint simulates the accumulation that would happen through successive `Return` instructions.

Formally,

$$\mathcal{T}\left(cp,\phi,ct,\kappa\right)=\begin{cases}\left(cp_h,\phi,ct,\kappa\right) & \text{if } \mathcal{H}\left(cp\right)=cp_h\\ \mathcal{T}\left(cp_k,\phi_k,ct\sqcup ct_k,\kappa_k\right) & \text{if } cp\notin dom\left(\mathcal{H}\right)\\ & \text{and } \kappa=\mathbf{retk}(cp_k,\phi_k,ct_k,\kappa_k)\,.\end{cases}$$

The `Throw` instruction requires two cases. In this case, execution continues at the appropriate error handler.

$$\frac{\mathcal{I}\left(cp\right)=\texttt{Throw(}r\texttt{)}}{\left(cp,\phi,\sigma,\kappa,ts,ct\right)\rightarrow\left(cp',\phi',\sigma',\kappa',ts',ct''\right)}, \text{ where}$$

$$\begin{aligned}
sa_s&=\left(\phi,r\right)\\
sa_d&=\left(\phi',\texttt{exception}\right)\\
\left(cp',\phi',ct'',\kappa'\right)&=\mathcal{T}(cp,\phi,ct',\kappa)\\
\sigma'&=\sigma\left[sa_d\mapsto\sigma\left(sa_s\right)\right]\\
ts'&=ts\left[sa_d\mapsto ts\left(sa_s\right)\cup itv_{\varsigma}\right]\\
ct'&=ct\left[ep_{\varsigma}\overset{\sqcup}{\mapsto}ts\left(sa_a\right)\cup ts\left(\sigma\left(sa_a\right)\right)\right].
\end{aligned}$$

This second case for the `Throw` instruction reaches the error state when no appropriate exception handler is found.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Throw(}r\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \textbf{errorstate}}, \text{ where}$$

$$\left(cp, \phi, ct, \kappa\right) \notin dom\left(\mathcal{T}\right) .$$

The `IGet` instruction represents the family of instance accessor instructions in Dalvik bytecode. It requires three transition rules. In the first, the object address is nonnull:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \left(next\left(cp\right), \phi, \sigma', \kappa, ts', ct'\right)}, \text{ where}$$

$$\begin{aligned}
oa &\neq \texttt{null} \\
sa_d &= \left(\phi, r_d\right) \\
sa_o &= \left(\phi, r_o\right) \\
oa &= \sigma\left(sa_o\right) \\
fa &= \left(oa, \mathit{field}\right) \\
\sigma' &= \sigma\left[sa_d \mapsto \sigma\left(fa\right)\right] \\
ts' &= ts\left[sa_d \mapsto ts\left(sa_a\right) \cup ts\left(oa\right) \cup ts\left(fa\right) \cup itv_\varsigma\right] \\
ct' &= ct\left[ep_\varsigma \overset{\sqcup}{\mapsto} ts\left(sa_a\right) \cup ts\left(oa\right)\right] .
\end{aligned}$$

In the second case, the object address is null and the exception is handled. This instruction implicitly generates an exception. It is useful to think of this instruction as two instructions in one. The first instruction generates an exception and the second throws it.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, } r_o\texttt{, } \mathit{field}\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \to \left(cp', \phi', \sigma', \kappa', ts', ct''\right)}, \text{ where}$$

$$\begin{aligned}
oa &= \texttt{null} \\
\left(cp', \phi', ct'', \kappa'\right) &= \mathcal{T}\left(cp, \phi, ct', \kappa\right) \\
sa_o &= \left(\phi, r_o\right) \\
oa &= \sigma\left(sa_o\right) \\
sa_{ex} &= \left(\phi', \texttt{exception}\right) \\
oa_{ex} & \text{ is a fresh object address} \\
\sigma' &= \sigma\left[sa_{ex} \mapsto oa_{ex}\right] \\
ts' &= ts\left[sa_{ex} \mapsto ts\left(sa_o\right) \cup ts\left(oa\right) \cup itv_{\varsigma},\right. \\
&\qquad \left. oa_{ex} \mapsto ts\left(sa_o\right) \cup ts\left(oa\right) \cup itv_{\varsigma}\right] \\
ct' &= ct\left[ep_{\varsigma} \overset{\sqcup}{\mapsto} ts\left(sa_a\right) \cup ts\left(oa\right)\right] .
\end{aligned}$$

In the last case, the exception reaches the top level:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, } r_o\texttt{, } \mathit{field}\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \to \textbf{errorstate}}, \text{ where}$$

$$\begin{aligned}
oa &= \texttt{null} \\
\left(cp, \phi, ct, \kappa\right) &\notin dom\left(\mathcal{T}\right) \\
sa_o &= \left(\phi, r_o\right) \\
oa &= \sigma\left(sa_o\right) .
\end{aligned}$$

The `IPut` instruction also represents a family of instructions; `IPut` stores values in objects. Like `IGet`, `IPut` requires three transition rules. In this first rule, no exception is thrown.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IPut(}r_s\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \to \left(next\left(cp\right), \phi, \sigma', \kappa, ts', ct\right)}, \text{ where}$$

$$\begin{aligned}
oa &\neq \texttt{null} \\
sa_s &= \left(\phi, r_s\right) \\
sa_o &= \left(\phi, r_o\right) \\
oa &= \sigma\left(sa_o\right) \\
fa &= \left(oa, \mathit{field}\right) \\
\sigma' &= \sigma\left[fa \mapsto \sigma\left(sa_s\right)\right] \\
ts' &= ts\left[fa \mapsto ts\left(sa_s\right) \cup ts\left(sa_o\right) \cup ts\left(oa\right) \cup itv_\varsigma\right] \\
ct' &= ct\left[ep_\varsigma \overset{\sqcup}{\mapsto} ts\left(sa_o\right) \cup ts\left(oa\right)\right].
\end{aligned}$$

In this second case for `IPut`, an exception is thrown and then caught:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IPut(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \to \left(cp', \phi', \sigma', \kappa', ts', ct''\right)}, \text{ where}$$

$$\begin{aligned}
oa &= \texttt{null} \\
\left(cp', \phi', ct'', \kappa'\right) &= \mathcal{T}\left(cp, \phi, ct', \kappa\right) \\
sa_o &= \left(\phi, r_o\right) \\
oa &= \sigma\left(sa_o\right) \\
sa_{ex} &= \left(\phi', \texttt{exception}\right) \\
oa_{ex} &\text{ is a fresh object address} \\
\sigma' &= \sigma\left[sa_{ex} \mapsto oa_{ex}\right] \\
ts' &= ts\left[sa_{ex} \mapsto ts\left(sa_o\right) \cup ts\left(oa\right) \cup itv_\varsigma,\right. \\
&\qquad \left. oa_{ex} \mapsto ts\left(sa_o\right) \cup ts\left(oa\right) \cup itv_\varsigma\right] \\
ct' &= ct\left[ep_\varsigma \overset{\sqcup}{\mapsto} ts\left(sa_o\right) \cup ts\left(oa\right)\right].
\end{aligned}$$

In this last case for `IPut`, an exception is thrown and is uncaught:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IPut(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \textbf{errorstate}}, \text{ where}$$

$$\begin{aligned} oa &= \texttt{null} \\ \left(cp', \phi', ct', \kappa'\right) &\notin dom\left(\mathcal{T}\right) \\ sa_o &= \left(\phi, r_o\right) \\ oa &= \sigma\left(sa_o\right) . \end{aligned}$$

3.2.4 Abstraction

A small-step analyzer as described by Van Horn and Might [35] overapproximates program behavior. Abstraction of taint stores and context taint maps is straightforward: they store execution points, which are code points and stack heights. Code points need no abstraction and the height of abstract stacks is suitable. Any abstraction of continuations (even that of PDCFA [9]) admits indeterminate stack heights; an abstract execution point with an indeterminate stack height cannot be a postdominator. Creating execution points with $\widehat{SH}$ (defined in Section 3.2.4.1) ensures that the generated state space is finite, even though $\mathbb{Z}$ is unbounded.

The abstract interpreter may use any allocators or abstractions for primitives, as long as they are sound. As a result, any form of polyvariance can be employed during abstract interpretation [13].

As in Van Horn and Might's work, continuations are made finite by store allocation. A continuation does not store its successor but an address at which its successor is stored.

The formal definition of the abstract state space is given in Figure 3.4.

3.2.4.1 Helpers for abstract semantics

The abstract semantics also use the shorthand $\hat{\sigma}[\hat{a} \overset{\sqcup}{\mapsto} \widehat{val}]$ for a weak update. Formally,

$$\hat{\sigma}[\hat{a} \overset{\sqcup}{\mapsto} \widehat{val}] = \hat{\sigma}[\hat{a} \mapsto \hat{\sigma}\left(\hat{a}\right) \sqcup \widehat{val}] .$$

Each projection metafunction has an abstract counterpart. $\widehat{SH} : \widehat{Kont} \times \widehat{Store} \times \mathcal{P}\left(\widehat{Kont}\right) \rightarrow \hat{H}$ calculates stack height. Whenever it encounters a loop in the abstract

$$
\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &::= \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \mid \textbf{errorstate} \mid \textbf{endstate} \\
\hat{\phi} \in \widehat{FP} &\text{ is a finite set of frame pointers} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \to \widehat{Value} \\
\widehat{val} \in \widehat{Value} &= \hat{\mathbb{Z}} + \mathcal{P}\left(\widehat{ObjectAddress}\right) + \mathcal{P}\left(\widehat{Kont}\right) \\
\hat{z} \in \hat{\mathbb{Z}} &\text{ is a finite set of abstract integers} \\
\hat{\kappa} \in \widehat{Kont} &::= \widehat{\textbf{retk}}(cp, \hat{\phi}, \widehat{ct}, \widehat{ka}) \mid \textbf{halt} \\
\widehat{ts} \in \widehat{TaintStore} &= \widehat{Addr} \to \mathcal{P}\left(\widehat{TaintValue}\right) \\
\widehat{tv} \in \widehat{TaintValue} &= \widehat{ExplicitTV} + \widehat{ImplicitTV} \\
\widehat{etv} \in \widehat{ExplicitTV} &= \widehat{EP} \\
\widehat{itv} \in \widehat{ImplicitTV} &= \mathcal{P}\left(\widehat{EP} \times \widehat{EP}\right) \times \widehat{ExplicitTV} \\
\widehat{ct} \in \widehat{ContextTaint} &= \widehat{EP} \to \mathcal{P}\left(\widehat{TaintValue}\right) \\
\widehat{ep} \in \widehat{EP} &::= \widehat{\textbf{ep}}\left(cp, \hat{h}\right) \mid \textbf{errorsummary} \mid \textbf{endsummary} \\
\hat{h} \in \hat{H} &= \mathbb{Z} + \{unknown\} \\
z \in \mathbb{Z} &\text{ is the set of integers} \\
\hat{a} \in \widehat{Addr} &::= \widehat{sa} \mid \widehat{fa} \mid \widehat{oa} \mid \widehat{ka} \mid \texttt{null} \\
\widehat{sa} \in \widehat{StackAddress} &= \widehat{FP} \times \mathsf{Register} \\
\widehat{fa} \in \widehat{FieldAddress} &= \widehat{ObjectAddress} \times \mathsf{Field} \\
\widehat{oa} \in \widehat{ObjectAddress} &\text{ is a finite set of addresses} \\
\widehat{ka} \in \widehat{KontAddress} &\text{ is a finite set of addresses}
\end{aligned}
$$

Figure 3.4: Abstract state space for the augmented-state analysis

continuation stack, the stack height cannot be determined. The stack height is also indeterminate in the case where different continuations at the same address along the stack height have different depths.

$$\widehat{SH}\left(\hat{\kappa}, \hat{\sigma}, seen\right) = \begin{cases} unknown & \text{if } \hat{\kappa} \in seen \\ 0 & \text{if } \hat{\kappa} = \textbf{halt} \text{ and } \hat{\kappa} \notin seen \\ 1 + next & \text{if } \hat{\kappa} = \widehat{\textbf{retk}}(cp, \hat{\phi}, \widehat{ct}, \widehat{ka}) \text{ and } \hat{\kappa} \notin seen \\ & \text{and for every } \hat{\kappa}' \text{ in } \hat{\sigma}\left(\widehat{ka}\right), \\ & \quad \widehat{SH}\left(\hat{\kappa}', \hat{\sigma}, seen + \{\hat{\kappa}\}\right) = next \\ unknown & \text{otherwise}. \end{cases}$$

$\widehat{p} : \hat{\Sigma} \to \widehat{EP}$ uses $\widehat{SH}$ to create execution points:

$$\widehat{p}\left(\hat{\varsigma}\right) = \begin{cases} \widehat{\textbf{ep}}\left(cp, \hat{h}\right) & \text{if } \hat{\varsigma} = \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \text{ and } \hat{h} = \widehat{SH}\left(\hat{\kappa}, \hat{\sigma}, \{\}\right) \\ \textbf{endsummary} & \text{if } \hat{\varsigma} = \textbf{endstate} \\ \textbf{errorsummary} & \text{if } \hat{\varsigma} = \textbf{errorstate}. \end{cases}$$

The abstract analogue of *implicit* is $\widehat{implicit} : \widehat{EP} \times \widehat{EP} \times \widehat{ExplicitTV}$, whose definition is straightforward:

$$\widehat{implicit}\left(\widehat{ep}_b, \widehat{ep}_a, \widehat{tv}\right) = \begin{cases} \left(\{(\widehat{ep}_b, \widehat{ep}_a)\}, \widehat{tv}\right) & \text{if } \widehat{tv} \in \widehat{ExplicitTV} \\ \left(\{(\widehat{ep}_b, \widehat{ep}_a)\} \cup \hat{B}, \widehat{etv}_i\right) & \text{if } \widehat{tv} = \left(\hat{B}, \widehat{etv}_i\right). \end{cases}$$

Each shorthand expression from the concrete semantics also has an abstract equivalent. For an abstract state $\hat{\varsigma}$ with context taint map

$$\widehat{ct} = \left[\widehat{ep}_1 \to \left\{\widehat{tv}_{1,1}, \ldots \widehat{tv}_{1,i}\right\}, \ldots, \widehat{ep}_n \to \left\{\widehat{tv}_{n,1}, \ldots, \widehat{tv}_{n,j}\right\}\right],$$

$$\widehat{ep}_\varsigma = \widehat{p}\left(\hat{\varsigma}\right)$$

and

$$\widehat{itv}_\varsigma = \left\{\widehat{implicit}\left(\widehat{ep}_1, \widehat{ep}_\varsigma, \widehat{tv}_{1,1}\right), \ldots, \widehat{implicit}\left(\widehat{ep}_1, \widehat{ep}_\varsigma, \widehat{tv}_{1,i}\right), \ldots, \right.$$
$$\left. \widehat{implicit}\left(\widehat{ep}_n, \widehat{ep}_\varsigma, \widehat{tv}_{n,1}\right), \ldots, \widehat{implicit}\left(\widehat{ep}_n, \widehat{ep}_\varsigma, \widehat{tv}_{n,j}\right)\right\}.$$

Abstract context taint maps merge as expected. Given two abstract context taint sets $\widehat{ct}_1 = \left[\widehat{ep}_1 \to \left\{\widehat{tv}_1\right\}, \widehat{ep}_2 \to \left\{\widehat{tv}_2\right\}\right]$ and $\widehat{ct}_2 = \left[\widehat{ep}_1 \to \left\{\widehat{tv}_3\right\}\right]$,

$$\widehat{ct}_1 \sqcup \widehat{ct}_2 = \left[\widehat{ep}_1 \to \left\{\widehat{tv}_1, \widehat{tv}_3\right\}, \widehat{ep}_2 \to \left\{\widehat{tv}_2\right\}\right].$$

It is also useful to define an abstraction metafunction for values $\alpha_v : Value \rightharpoonup \widehat{Value}$ takes a concrete value and returns its abstraction. The semantics of α_v reflect the choice of abstractions and allocators used by the analysis. Using the abstraction of signs mentioned in Section 3.2.4, α_v would be defined over 32-bit two's complement integers as α_+. Other abstractions might use a different abstraction for integers; α_v is a placeholder for whichever abstraction is chosen.

In practice, abstract addresses are generated during interpretation and not abstracted from concrete addresses. The exception to this rule is `null`, which is equivalent to the literal integer 0 [15] and is abstracted accordingly.

3.2.4.2 Abstract semantics

The semantics of the abstract interpreter use the relation $\leadsto \subseteq \hat{\Sigma} \times \hat{\Sigma}$. This relation is defined case by case, with cases grouped by instruction.

In the case of the `Const` instruction, abstraction means using a weak update and abstracting the concrete value stored in the instruction using the α_v chosen for the analysis.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Const(}r\texttt{, }c\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \left(next\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{ts}', \widehat{ct}\right)}, \text{ where}$$

$$\begin{aligned} \widehat{sa} &= \left(\hat{\phi}, r\right) \\ \hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \alpha_v\left(c\right)\right] \\ \widehat{ts}' &= \widehat{ts}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \widehat{itv_\varsigma}\right]. \end{aligned}$$

Abstracting the `Move` instruction requires only that the concrete semantics be modified to update weakly:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Move(}r_d\texttt{, }r_s\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \left(next\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{ts}', \widehat{ct}\right)}, \text{ where}$$

$$\begin{aligned} \widehat{sa}_d &= \left(\hat{\phi}, r_d\right) \\ \widehat{sa}_s &= \left(\hat{\phi}, r_s\right) \\ \hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_s\right)\right] \\ \widehat{ts}' &= \widehat{ts}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_s\right) \cup \widehat{itv_\varsigma}\right]. \end{aligned}$$

The `Invoke` instruction simulates Dalvik's invoke instructions. For virtual methods, dispatch may discover multiple objects in the store at the given address. In this case, a successor is created for each method that could be dispatched.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Invoke(}mName\texttt{, } r_1\texttt{, ... , } r_n\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \left(cp', \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{ts}', \widehat{ct}'\right)}, \text{ where}$$

$$\begin{aligned}
cp' &= init\left(\mathcal{M}\left(mName\right)\right) \\
\hat{\kappa}' &= \widehat{\mathbf{retk}}(cp, \hat{\phi}, \widehat{ct}, \hat{\kappa}) \\
\hat{\phi}' &= \text{ is a fresh frame pointer} \\
&\quad \text{for each } i \text{ from } 1 \text{ to } n, \\
\widehat{sa}_{di} &= \left(\hat{\phi}', i\right) \text{ and } \widehat{sa}_{si} = \left(\hat{\phi}, r_i\right) \\
\hat{\sigma}' &= \hat{\sigma}[\widehat{sa}_{d1} \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_{s1}\right), \ldots, \widehat{sa}_{dn} \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_{sn}\right)] \\
\widehat{ts}' &= \widehat{ts}[\widehat{sa}_{d1} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_{s1}\right) \cup \widehat{itv}_{\varsigma}, \ldots, \widehat{sa}_{dn} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_{sn}\right) \cup \widehat{itv}_{\varsigma}] \\
\widehat{ct}' &= \widehat{ct}\left[\widehat{ep}_{\varsigma} \overset{\sqcup}{\mapsto} \hat{T}\right] \\
\hat{T} &= \widehat{ts}\left(\widehat{sa}_{s1}\right) \cup \left(\bigcup_{\widehat{oa} \in \widehat{OA}} \widehat{ts}\left(\widehat{oa}\right)\right) \\
\widehat{OA} &= \hat{\sigma}\left(\widehat{sa}_{s1}\right) \cap \widehat{ObjectAddress}\,.
\end{aligned}$$

An abstract continuation address may point to multiple continuations, so $\leadsto$ is defined for each such continuation when handling a `Return` instruction.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Return(}r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \left(next\left(cp'\right), \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{ts}', \widehat{ct}'\right)}, \text{ where}$$

$$\begin{aligned}
\hat{\kappa} &= \widehat{\mathbf{retk}}(cp, \hat{\phi}', \widehat{ct}_k, \widehat{ka}) \\
\hat{\kappa}' &\in \hat{\sigma}\left(\widehat{ka}\right) \cap \widehat{Kont} \\
\widehat{sa}_d &= \left(\hat{\phi}', \texttt{result}\right) \\
\widehat{sa}_s &= \left(\hat{\phi}, r\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_s\right)\right] \\
\widehat{ts}' &= \widehat{ts}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_s\right) \cup \widehat{itv}_{\varsigma}\right] \\
\widehat{ct}' &= \widehat{ct} \sqcup \widehat{ct}_k\,.
\end{aligned}$$

This second case for the `Return` instruction halts at the end of the program:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Return(}r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \rightsquigarrow \textbf{endstate}}, \text{ where}$$

$$\hat{\kappa} = \textbf{halt}\,.$$

The `IfEqz` instruction, like the `Return` instruction, can have multiple successors; the two cases for cp' are not mutually exclusive.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IfEqz(}r\texttt{, }ln\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \rightsquigarrow \left(cp', \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}'\right)}, \text{ where}$$

$$\begin{aligned}
\widehat{sa}_s &= \left(\hat{\phi}, r\right) \\
cp' &= \begin{cases} jump\left(cp, ln\right) & \text{if } \alpha_v\left(0\right) \sqsubseteq \sigma\left(\widehat{sa}_s\right) \\ next\left(cp\right) & \text{if there is an integer } z \text{ such that } z \neq 0 \text{ and } \alpha_v\left(z\right) \sqsubseteq \sigma\left(\widehat{sa}_s\right) \end{cases} \\
\widehat{ct}' &= \widehat{ct}\left[\widehat{ep}_\varsigma \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_s\right)\right]\,.
\end{aligned}$$

The `Add` instruction represents all arithmetic instructions. The $\hat{+}$ operation is the abstraction of 32-bit two's complement addition appropriate to the abstract domain used for integers.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Add(}r_d\texttt{, }r_l\texttt{, }r_r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \rightsquigarrow \left(next\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{ts}', \widehat{ct}\right)}, \text{ where}$$

$$\begin{aligned}
\widehat{sa}_d &= \left(\hat{\phi}, r_d\right) \\
\widehat{sa}_l &= \left(\hat{\phi}, r_l\right) \\
\widehat{sa}_r &= \left(\hat{\phi}, r_r\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_l\right) \ \hat{+}\ \hat{\sigma}\left(\widehat{sa}_r\right)\right] \\
\widehat{ts}' &= \widehat{ts}[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_l\right) \cup \widehat{ts}\left(\widehat{sa}_r\right) \cup \widehat{itv}_\varsigma]\,.
\end{aligned}$$

Abstraction of the `NewInstance` instruction is straightforward:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{NewInstance(}r\texttt{, }className\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \left(next\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{ts}', \widehat{ct}\right)}, \text{ where}$$

$$\begin{aligned}
&\widehat{oa} \text{ is a fresh object address}\\
&\widehat{sa} = \left(\hat{\phi}, r\right)\\
&\hat{\sigma}' = \hat{\sigma}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \widehat{oa}\right]\\
&\widehat{ts}' = \widehat{ts}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \widehat{itv_{\varsigma}}\right] .
\end{aligned}$$

$\mathcal{T}$ has an abstract counterpart, $\hat{\mathcal{T}}$:

$$\hat{\mathcal{T}} : CodePoint \times \widehat{FP} \times \widehat{Store} \times \widehat{ContextTaint} \times \widehat{Kont} \to$$
$$\mathcal{P}\left(CodePoint \times \widehat{FP} \times \widehat{ContextTaint} \times \widehat{Kont}\right) + \{error\} .$$

Because of potential imprecision in the continuation stack, $\hat{\mathcal{T}}$ returns a set of tuples instead of one tuple. The special value *error* is a member of the set whenever an exception reaches the top level. Like $\mathcal{T}$, $\hat{\mathcal{T}}$ is defined by cases. In the first case, a handler is found:

If

$$\mathcal{H}\left(cp\right) = cp_h$$

then

$$\hat{\mathcal{T}}\left(cp, \hat{\phi}, \hat{\sigma}, \widehat{ct}, \hat{\kappa}\right) = \left\{\left(cp_h, \hat{\phi}, \widehat{ct}, \hat{\kappa}\right)\right\} .$$

In the second case, no handler is found at the current stack depth and it must recur. Also, the current continuation is not **halt**. Because continuations are store-allocated, the recursive step may consider multiple continuations. It merges the results of the different calls by set union. Formally,

If

$$\begin{gathered}
cp \notin dom\left(\mathcal{H}\right), \text{ and}\\
\widehat{ct}_k = \emptyset, \text{ and}\\
\hat{\kappa} = \widehat{\mathbf{retk}}(cp_k, \hat{\phi}_k, \widehat{ct}_k, \widehat{ka})
\end{gathered}$$

then

$$\hat{\mathcal{T}}\left(cp, \hat{\phi}, \hat{\sigma}, \widehat{ct}, \hat{\kappa}\right) = \bigcup_{\hat{\kappa}_k \in \hat{K}} \hat{\mathcal{T}}\left(cp_k, \hat{\phi}_k, \widehat{ct} \sqcup \widehat{ct}_k, \hat{\kappa}_k\right), \text{ where}$$

$$\hat{K} = \hat{\sigma}\left(\widehat{ka}\right) \cap \widehat{Kont}\,.$$

In all other cases (notably, when $\hat{\kappa} = \mathbf{halt}$),

$$\hat{\mathcal{T}}\left(cp, \hat{\phi}, \hat{\sigma}, \widehat{ct}, \hat{\kappa}\right) = \{error\}\ .$$

As before, there are two cases for the `Throw` instruction. As with other abstract production rules, the cases are not exclusive. Also, it is possible that $\hat{\mathcal{T}}$ will return multiple tuples, each of which can be used to construct a successor state. This first case demonstrates a caught exception:

$$\frac{\mathcal{I}(cp) = \texttt{Throw(}r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \left(cp', \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{ts}', \widehat{ct}''\right)}, \text{ where}$$

$$\begin{aligned}
\left(cp', \hat{\phi}', \widehat{ct}'', \hat{\kappa}'\right) &\in \hat{\mathcal{T}}\left(cp, \hat{\phi}, \hat{\sigma}, \widehat{ct}', \hat{\kappa}\right)\\
\widehat{sa}_s &= \left(\hat{\phi}, r\right)\\
\widehat{sa}_d &= \left(\hat{\phi}', \texttt{exception}\right)\\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_s\right) \cap \widehat{ObjectAddress}\\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{oa}\right]\\
\widehat{ts}' &= \widehat{ts}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_s\right) \cup \widehat{itv}_\varsigma\right]\\
\widehat{ct}' &= \widehat{ct}\left[\widehat{ep}_\varsigma \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_s\right) \cup \widehat{ts}\left(\widehat{oa}\right)\right]\ .
\end{aligned}$$

In the second case for the `Throw` instruction, no handler is found and the exception reaches the top level:

$$\frac{\mathcal{I}(cp) = \texttt{Throw(}r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \mathbf{errorstate}}, \text{ where}$$

$$error \in \hat{\mathcal{T}}\left(cp, \hat{\phi}, \hat{\sigma}, \widehat{ct}, \hat{\kappa}\right)\ .$$

The `IGet` requires three nonexclusive transition rules. In the first, the object address is nonnull, so no exception is thrown.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, } r_o\texttt{, } \mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \left(\mathit{next}\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{ts}', \widehat{ct}'\right)}, \text{ where}$$

$$\begin{aligned}
&\exists\, oa \neq \texttt{null} : \alpha_v\left(oa\right) \sqsubseteq \widehat{oa} \\
&\qquad \widehat{sa}_d = \left(\hat{\phi}, r_d\right) \\
&\qquad \widehat{sa}_o = \left(\hat{\phi}, r_o\right) \\
&\qquad \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \cap \widehat{ObjectAddress} \\
&\qquad fa = \left(\widehat{oa}, \mathit{field}\right) \\
&\qquad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{fa}\right)\right] \\
&\qquad \widehat{ts}' = \widehat{ts}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right) \cup \widehat{ts}\left(\widehat{fa}\right) \cup \widehat{itv}_{\varsigma}\right] \\
&\qquad \widehat{ct}' = \widehat{ct}\left[\widehat{ep}_{\varsigma} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right)\right] .
\end{aligned}$$

In this second case for the `IGet` instruction, an exception is thrown and caught:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, } r_o\texttt{, } \mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \left(cp', \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{ts}', \widehat{ct}''\right)}, \text{ where}$$

$$\begin{aligned}
\texttt{null} &\sqsubseteq \widehat{oa} \\
\left(cp', \hat{\phi}', \widehat{ct}'', \hat{\kappa}'\right) &\in \hat{\mathcal{T}}\left(cp, \hat{\phi}, \hat{\sigma}, \widehat{ct}', \hat{\kappa}\right) \\
\widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) \\
\widehat{sa}_{ex} &= \left(\hat{\phi}', \texttt{exception}\right) \\
\widehat{oa}_{ex} &\text{ is a fresh object address} \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{oa}_{ex}\right] \\
\widehat{ts}' &= \widehat{ts}\Big[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right) \cup \widehat{itv}_{\varsigma}, \\
&\qquad\quad \widehat{oa}_{ex} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right) \cup \widehat{itv}_{\varsigma}\Big] \\
\widehat{ct}' &= \widehat{ct}\left[\widehat{ep}_{\varsigma} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right)\right] .
\end{aligned}$$

In the final case for the `IGet` instruction, an exception is thrown and reaches the top level:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \rightsquigarrow \textbf{errorstate}}, \text{ where}$$

$$\begin{aligned} \texttt{null} &\sqsubseteq \widehat{oa} \\ \mathit{error} &\in \hat{\mathcal{T}}\left(cp, \hat{\phi}, \hat{\sigma}, \widehat{ct}, \hat{\kappa}\right) \\ \widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\ \widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) . \end{aligned}$$

Like `IGet`, `IPut` requires three transition rules. The first case shows execution without any exception:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IPut(}r_s\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \rightsquigarrow \left(\mathit{next}\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{ts}', \widehat{ct}'\right)}, \text{ where}$$

$$\begin{aligned} \exists oa \neq \texttt{null} : \alpha_v\left(oa\right) &\sqsubseteq \widehat{oa} \\ \widehat{sa}_s &= \left(\hat{\phi}, r_s\right) \\ \widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\ \widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) \\ \widehat{fa} &= \left(\widehat{oa}, \mathit{field}\right) \\ \hat{\sigma}' &= \hat{\sigma}\left[\widehat{fa} \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_s\right)\right] \\ \widehat{ts}' &= \widehat{ts}\left[\widehat{fa} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_s\right) \cup \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right) \cup \widehat{itv}_\varsigma\right] \\ \widehat{ct}' &= \widehat{ct}\left[\widehat{ep}_\varsigma \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right)\right] . \end{aligned}$$

The second case for `IPut` shows a caught exception:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IPut(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \left(cp', \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{ts}', \widehat{ct}''\right)}, \text{ where}$$

$$\begin{aligned}
\texttt{null} &\sqsubseteq \widehat{oa} \\
\left(cp', \hat{\phi}', \widehat{ct}'', \hat{\kappa}'\right) &\in \hat{\mathcal{T}}\left(cp, \hat{\phi}, \hat{\sigma}, \widehat{ct}', \hat{\kappa}\right) \\
\widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) \\
\widehat{sa}_{ex} &= \left(\hat{\phi}', \texttt{exception}\right) \\
\widehat{oa}_{ex} &\text{ is a fresh object address} \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{oa}_{ex}\right] \\
\widehat{ts}' &= \widehat{ts}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right) \cup \widehat{itv}_{\varsigma},\right. \\
&\qquad \left.\widehat{oa}_{ex} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right) \cup \widehat{itv}_{\varsigma}\right] \\
\widehat{ct}' &= \widehat{ct}\left[\widehat{ep}_{\varsigma} \overset{\sqcup}{\mapsto} \widehat{ts}\left(\widehat{sa}_o\right) \cup \widehat{ts}\left(\widehat{oa}\right)\right] .
\end{aligned}$$

The final case for `IPut` shows an uncaught exception:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IPut(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) \leadsto \textbf{errorstate}}, \text{ where}$$

$$\begin{aligned}
\texttt{null} &\sqsubseteq \widehat{oa} \\
\mathit{error} &\in \hat{\mathcal{T}}\left(cp, \hat{\phi}, \hat{\sigma}, \widehat{ct}, \hat{\kappa}\right) \\
\widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) .
\end{aligned}$$

3.3 Noninterference

The proof of noninterference operates on the concrete semantics, which are soundly overapproximated by the abstract semantics. The proof that any unsafe behavior will be identified in the concrete means that the abstract overapproximation is also guaranteed to identify all unsafe behaviors.

3.3.1 Influence

The **influence** of an execution point ep_0 is the set of execution points that lie along some path from ep_0 to its immediate postdominator ep_n where ep_n appears only at the end of the path. The function $\mathit{ipd} : \mathit{EP} \rightharpoonup \mathit{EP}$ is useful for notation; in this example, $\mathit{ipd}\,(ep_0) = ep_n$.

Given the set V of vertices in the execution point graph and the set E of edges in that same graph, the set P of all paths from ep_0 to ep_n can be defined:

$$P = \left\{ \langle ep_0, \ldots, ep_n \rangle \mid \forall i \in \{0, \ldots, n-1\}, \left(ep_i, ep_{i+1}\right) \in E \;\wedge ep_i \neq ep_n \right\} .$$

With P defined, the influence of ep_0 (when $\mathit{ipd}\,(ep_0) = ep_n$) can be defined as:

$$\mathit{influence}\,(ep_0) = \{ep \in V \mid \exists p \in P \;:\; ep \in p\} - \{ep_0, ep_n\} .$$

3.3.2 Program traces

A **program trace** π is a sequence $\langle \varsigma_1, \varsigma_2, \ldots, \varsigma_n \rangle$ of concrete states such that

$$\varsigma_1 \rightarrow \varsigma_2 \rightarrow \ldots \rightarrow \varsigma_n \text{ and } \varsigma_n \notin \mathit{dom}\,(\rightarrow) .$$

3.3.3 Observable behaviors

Which program behaviors are observable depends on the attack model and is a decision to be made by the user of this analysis. This proof considers the general case: Every program behavior is observable. A more realistic model would be that invocations of certain functions are observable. Accordingly, obs, which must be a (not necessarily proper) subset of Σ, is defined $obs = \Sigma$.

3.3.4 Valid taints

The given semantics has no notion of taint removal; instead, some taints are **valid** and the others are disregarded. Explicit taints are always valid. Implicit taints are created when some assignment is made. An implicit taint is valid if and only if its assignment happens during the influence of its branch. Accordingly, the set of all valid taints is:

$$\mathit{valid} = \mathit{ExplicitTV} \cup \mathit{valid}_i, \text{ where}$$

$$\mathit{valid}_i = \left\{ \, (B, \mathit{etv}_i) \in \mathit{ImplicitTV} \mid \forall\,(ep_b, ep_a) \in B,\; ep_a \in \mathit{influence}\,(ep_b) \, \right\} .$$

3.3.5 Labeled behaviors

A state ς has a **labeled behavior** if and only if it reads values at one or more addresses with valid taint or if it occurs at a state with valid context taint. The functions $inputs : \Sigma \rightarrow Addr^*$ and $labeled : \Sigma \rightarrow \mathcal{P}\left(TaintValue\right)$ are defined so that *inputs* identifies the addresses read by ς's instruction and *labeled* identifies the valid taints at those addresses. The use of itv_ς reflects context taint. The formal definition of *inputs* requires several cases:

$$\begin{aligned}
&\mathcal{I}\left(cp\right) = \texttt{Const(}r\texttt{, }c\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\rangle \\
&\mathcal{I}\left(cp\right) = \texttt{Move(}r_d\texttt{, }r_s\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\left(\phi, r_s\right)\rangle \\
&\mathcal{I}\left(cp\right) = \texttt{Invoke(}mName\texttt{, }r_1\texttt{, }\ldots\texttt{, }r_n\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\left(\phi, r_1\right), \ldots, \left(\phi, r_n\right)\rangle \\
&\mathcal{I}\left(cp\right) = \texttt{Return(}r\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\left(\phi, r\right)\rangle \\
&\mathcal{I}\left(cp\right) = \texttt{IfEqz(}r\texttt{, }ln\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\left(\phi, r\right)\rangle \\
&\mathcal{I}\left(cp\right) = \texttt{Add(}r_d\texttt{, }r_l\texttt{, }r_r\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\left(\phi, r_l\right), \left(\phi, r_r\right)\rangle \\
&\mathcal{I}\left(cp\right) = \texttt{NewInstance(}r\texttt{, }className\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\rangle \\
&\mathcal{I}\left(cp\right) = \texttt{Throw(}r\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\left(\phi, r\right)\rangle \\
&\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, }r_o\texttt{, }field\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\left(\phi, r_o\right), \sigma\left(\left(\phi, r_o\right)\right), \\
& && \qquad\qquad\qquad \sigma\left(\sigma\left(\left(\phi, r_o\right)\right), field\right)\rangle \\
&\mathcal{I}\left(cp\right) = \texttt{IPut(}r_s\texttt{, }r_o\texttt{, }field\texttt{)} && \Rightarrow inputs\left(\varsigma\right) = \langle\left(\phi, r_s\right)\rangle
\end{aligned}$$

The formal definition of *labeled* follows:

$$labeled\left(\varsigma\right) = \Big\{tv \in TaintValue \mid \exists\ a \in inputs\left(\varsigma\right) \cup itv_\varsigma\ :\ tv \in ts\left(a\right)\Big\} \cap valid\,.$$

3.3.6 Similar stores

Two stores are **similar** with respect to two frame pointers, two continuations, and two taint stores if and only if they differ only at reachable addresses that are tainted in their respective taint stores. This definition requires some related definitions, which follow.

Two stores are similar with respect to two addresses and two taint stores if and only if:

- Both stores are undefined at their respective address (Equation 3.1), or
- either store is tainted at its respective address (Equation 3.2), or
- the stores map their respective addresses to the same value (Equation 3.3), or
- the stores map respective addresses to structurally identical objects (Equation 3.4).

Formally,

$$\begin{aligned}
&(\sigma_1, a_1, ts_1) \approx_a (\sigma_2, a_2, ts_2) \\
&\Leftrightarrow \\
&\big(a_1 \notin dom\,(\sigma_1) \wedge a_2 \notin dom\,(\sigma_2)\big) \vee && (3.1)\\
&\big(\exists\ tv \in ts_1\,(a_1)\ :\ tv \in valid\ \vee\ \exists\ tv \in ts_2\,(a_2)\ :\ tv \in valid\big) \vee && (3.2)\\
&\sigma_1\,(a_1) = \sigma_2\,(a_2)\ \vee && (3.3)\\
&\big(\sigma_1\,(a_1) = oa_1\ \wedge\ \sigma_2\,(a_2) = oa_2\ \wedge && (3.4)\\
&\forall\ field \in \mathsf{Field}, (\sigma_1, (oa_1, field)\,, ts_1) \approx_a (\sigma_2, (oa_2, field)\,, ts_2)\big)\,.
\end{aligned}$$

With this definition, it is possible to define similarity with respect to frame pointers. Two stores are similar with respect to two frame pointers and two taint stores if and only if they are similar with respect to every address containing the respective frame pointers.

Formally,

$$(\sigma_1, \phi_1, ts_1) \approx_\phi (\sigma_2, \phi_2, ts_2) \Leftrightarrow \forall r \in \mathsf{Register}, (\sigma_1, (\phi_1, r)\,, ts_1) \approx_a (\sigma_2, (\phi_2, r)\,, ts_2)\ .$$

Two stores are similar with respect to two frame pointers, two continuations, and two taint stores if and only if:

- The stores are similar with respect to the given pair of frame pointers (Equation 3.5), and
- they are recursively similar with respect to the given continuations (Equation 3.6).

Formally,

$$(\sigma_1, \phi_1, \kappa_1, ts_1) \approx_\sigma (\sigma_2, \phi_2, \kappa_2, ts_2)$$

$$\Leftrightarrow$$

$$(\sigma_1, \phi_1, ts_1) \approx_\phi (\sigma_2, \phi_2, ts_2) \wedge \tag{3.5}$$

$$\big(\kappa_1 = \kappa_2 = \textbf{halt} \vee \tag{3.6}$$

$$\kappa_1 = \textbf{retk}(cp_1, \phi_1', ct_1', \kappa_1') \quad \wedge \quad \kappa_2 = \textbf{retk}(cp_2, \phi_2', ct_2', \kappa_2') \wedge$$

$$(\sigma_1, \phi_1', \kappa_1', ts_1) \approx_\sigma (\sigma_2, \phi_2', \kappa_2', ts_2)\big) \, .$$

3.3.7 Similar states

Two states are **similar** if and only if their execution points are identical and their stores are similar with respect to their frame pointers and continuations.

Formally, if $\varsigma_1 = (cp_1, \phi_1, \sigma_1, \kappa_1, ts_1, ct_1)$ and $\varsigma_2 = (cp_2, \phi_2, \sigma_2, \kappa_2, ts_2, ct_2)$, then $\varsigma_1 \approx_\varsigma \varsigma_2 \quad \Leftrightarrow \quad p\,(\varsigma_1) = p\,(\varsigma_2) \quad \wedge \quad (\sigma_1, \phi_1, \kappa_1, ts_1) \approx_\sigma (\sigma_2, \phi_2, \kappa_2, ts_2)$.

3.3.8 Similar traces

Any two traces, which we call $\pi = \langle \varsigma_1, \varsigma_2, \ldots, \varsigma_n \rangle$ and $\pi' = \langle \varsigma_1', \varsigma_2', \ldots, \varsigma_m' \rangle$ without loss of generality, are **similar** if and only if their observable behaviors are identical except for those marked as tainted. The formulation of trace similarity uses a partial function $dual : \pi \rightharpoonup \pi'$. Trace similarity is equivalent to the existence of a function *dual* such that:

- *dual* is injective (Equation 3.7), and
- *dual* maps each state in π to a similar state in π' if such a state exists (Equation 3.8), and
- all states in π not paired by *dual* occur in a tainted context (Equation 3.9), and
- all states in π' not paired by *dual* occur in a tainted context (Equation 3.10), and
- the pairs of similar states occur in the same order in their respective traces (Equation 3.11).

Formally,

$$\pi \approx_\pi \pi' \Leftrightarrow \exists\ dual:$$

$$\forall\ i, j \in 1 \ldots n,\ \ i \neq j \Rightarrow dual\,(i) \neq dual\,(j) \wedge \tag{3.7}$$

$$\varsigma_i \in dom\,(dual) \Rightarrow \varsigma_i \approx_\varsigma dual\,(\varsigma_i) \wedge \tag{3.8}$$

$$\varsigma_i \notin dom\,(dual) \Rightarrow itv_\varsigma \in valid\ \wedge \tag{3.9}$$

$$\varsigma'_j \notin range\,(dual) \Rightarrow itv_\varsigma \in valid\ \wedge \tag{3.10}$$

$$\forall\ i, j \in 1 \ldots n\ \ :\ \ dual\,(\varsigma_i) = \varsigma'_k \wedge dual\,(\varsigma_j) = \varsigma'_l, \tag{3.11}$$

$$i < j \Rightarrow k < l \quad \wedge \quad i = j \Rightarrow k = l \quad \wedge \quad i > j \Rightarrow k > l\,.$$

3.3.9 Transitivity of similarity

3.3.9.1 Lemma

If two states ς and ς' are similar, they have the same execution point ep_0. Without loss of generalization, the immediate postdominator of ep_0 in the execution point graph is ep_{pd}. The first successor of each state whose execution point is ep_{pd} is similar to the other successor.

Formally, if

- ς is similar to ς' (Equation 3.12), and
- each adjacent pair of states in the sequence $\langle \varsigma, \varsigma_1, \ldots, \varsigma_n \rangle$ is a predecessor and a successor (Equation 3.13), and
- each adjacent pair of states in the sequence $\langle \varsigma', \varsigma'_1, \ldots, \varsigma'_m \rangle$ is a predecessor and a successor (Equation 3.14), and
- the respective execution points of ς and ς' are both ep_0 (Equation 3.15), and
- the respective execution points of ς_n and ς'_m are both ep_{pd} (Equation 3.16), and
- ep_{pd} is the immediate postdominator of ep_0 (Equation 3.17), and
- no intermediate state's execution point is ep_{pd} (Equation 3.18),

then ς_n is similar to ς'_m (Equation 3.19).

$$\varsigma \approx_\varsigma \varsigma' \wedge \tag{3.12}$$

$$\varsigma \to \varsigma_1 \to \ldots \to \varsigma_n \wedge \tag{3.13}$$

$$\varsigma' \to \varsigma'_1 \to \ldots \to \varsigma'_m \wedge \tag{3.14}$$

$$p\left(\varsigma\right) = p\left(\varsigma'\right) = ep_0 \wedge \tag{3.15}$$

$$p\left(\varsigma_n\right) = p\left(\varsigma'_m\right) = ep_{pd} \wedge \tag{3.16}$$

$$ipd\left(ep_0\right) = ep_{pd} \wedge \tag{3.17}$$

$$\forall\ \varsigma_i \in \{\varsigma_1, \ldots, \varsigma_{n-1}\} \cup \left\{\varsigma'_1, \ldots, \varsigma'_{m-1}\right\},\ p\left(\varsigma_i\right) \neq ep_{pd} \tag{3.18}$$

$$\Rightarrow \varsigma_n \approx_\varsigma \varsigma'_m \tag{3.19}$$

3.3.9.2 Proof

Without loss of generality,

$$\varsigma = \left(cp, \phi, \sigma, \kappa, ts, ct\right) \text{ and } \varsigma' = \left(cp', \phi', \sigma', \kappa', ts', ct'\right) \text{ and}$$
$$\varsigma_n = \left(cp_n, \phi_n, \sigma_n, \kappa_n, ts_n, ct_n\right) \text{ and } \varsigma'_m = \left(cp_m, \phi_m, \sigma_m, \kappa_m, ts_m, ct_m\right)\ .$$

$\varsigma_1, \ldots, \varsigma_{n-1}$ and $\varsigma'_1, \ldots, \varsigma'_{m-1}$ are **intermediate states**.

It is given that $p\left(\varsigma_n\right) = p\left(\varsigma'_m\right)$. All that remains is to prove that

$$\left(\sigma_n, \phi_n, \kappa_n, ts_n\right) \approx_\sigma \left(\sigma_m, \phi_m, \kappa_m, ts_m\right)\ .$$

By the definitions of influence and of *valid* and by induction on the instructions in the language, all changes to the store between ς and ς_n and between ς' and ς'_m are marked as tainted. Crucially, this includes changes to heap values as well as to stack values. The following four cases cover all possible additions to the stores:

- Addresses added to σ in some intermediate state (Equation 3.20),
- addresses added to σ' in some intermediate state (Equation 3.21),
- addresses changed in σ in some intermediate state (Equation 3.22), and
- addresses changed in σ' in some intermediate state (Equation 3.22).

$$\forall\ a \in \mathit{Addr},$$

$$a \notin \mathit{dom}\,(\sigma) \wedge a \in \mathit{dom}\,(\sigma_n) \;\Rightarrow\; \mathit{ts}_n\,(a) \cap \mathit{valid} \neq \emptyset \tag{3.20}$$

$$a \notin \mathit{dom}\,(\sigma') \wedge a \in \mathit{dom}\,(\sigma_m) \;\Rightarrow\; \mathit{ts}_m\,(a) \cap \mathit{valid} \neq \emptyset \tag{3.21}$$

$$\sigma\,(a) \neq \sigma_n\,(a) \;\Rightarrow\; \mathit{ts}_n\,(a) \cap \mathit{valid} \neq \emptyset \tag{3.22}$$

$$\sigma'\,(a) \neq \sigma_m\,(a) \;\Rightarrow\; \mathit{ts}_m\,(a) \cap \mathit{valid} \neq \emptyset \tag{3.23}$$

The only changes that can occur to the continuation stack in any circumstance are removal of stack frames (`Return`, `Throw`, `IGet`, and `IPut` instructions) and the addition of new stack frames (`Invoke` instructions).

Since `Invoke` uses only fresh stack frames, all stack addresses with frames created in intermediate states (FP_f) are undefined in σ and σ':

$$\forall\ r \in \mathsf{Register}, \phi_f \in \mathit{FP}_f,\ (\phi_f, r) \notin \mathit{dom}\,(\sigma) \cup \mathit{dom}\,(\sigma')\ .$$

This, together with the fact that all updates to heap values are tainted, proves that σ_n and σ_m are similar with respect to any pair of frame pointers if one of those is in FP_f:

$$\phi_n \in \mathit{FP}_f \vee \phi_m \in \mathit{FP}_f \Rightarrow (\sigma_n, \phi_n, \mathit{ts}_n) \approx_\phi (\sigma_m, \phi_m, \mathit{ts}_m)$$

Because $p\,(\varsigma) = p\,(\varsigma')$, the stack heights in ς and ς' are equal. Because of the restrictions of stack operations, ϕ_n is either ϕ, a fresh stack frame, or some stack frame from within κ. Similarly, ϕ_m is either ϕ', a fresh stack frame, or some stack frame from within κ'. If ϕ_n is not fresh, it is identical to some suffix of κ. Crucially, this means that no reordering of existing frame pointers is possible. The same relationship holds between ϕ_m and κ'. As such, ϕ_n and ϕ_m are either ϕ and ϕ', some pair from continuations at the same height from **halt**, or at least one of them is fresh. The same is true of each pair of frame pointers at identical height in κ_n and κ_m. In all of these cases, the two stores must be similar with respect to the frame pointers and their taint stores. Accordingly,

$$(\sigma_n, \phi_n, \kappa_n, \mathit{ts}_n) \approx_\sigma (\sigma_m, \phi_m, \kappa_m, \mathit{ts}_m)\ .$$

3.3.10 Global transitivity of similarity

3.3.10.1 Lemma

Any two finite program traces that begin with similar states are similar.

Formally, if $\pi = \langle \varsigma_1, \ldots, \varsigma_n \rangle$ and $\pi' = \langle \varsigma'_1, \ldots, \varsigma'_m \rangle$, then $\varsigma_1 \approx_\varsigma \varsigma'_1 \Rightarrow \pi \approx_\pi \pi'$.

3.3.10.2 Proof

By induction on transitivity of similarity.

3.3.11 Labeled interference in similar states

3.3.11.1 Lemma

Any two similar states exhibit the same behavior or at least one of them exhibits behavior that is labeled as insecure.

$$\begin{gathered}
\varsigma_1 \approx_\varsigma \varsigma_2 \Rightarrow \mathit{labeled}\,(\varsigma_1) \neq \emptyset \lor \mathit{labeled}\,(\varsigma_2) \neq \emptyset \lor \\
\forall i \in \langle 1, \ldots n \rangle, (\sigma_1, a_i, \mathit{ts}_1) \approx_a (\sigma_2, a'_i, \mathit{ts}_2)\,, \text{ where} \\
\mathit{inputs}\,(\varsigma_1) = \langle a_1, \ldots, a_n \rangle \text{ and } \mathit{inputs}\,(\varsigma_2) = \langle a'_1, \ldots, a'_n \rangle \;\text{ and} \\
\varsigma_1 = (\mathit{cp}_1, \phi_1, \sigma_1, \kappa_1, \mathit{ts}_1, \mathit{ct}_1) \;\text{ and } \varsigma_2 = (\mathit{cp}_2, \phi_2, \sigma_2, \kappa_2, \mathit{ts}_2, \mathit{ct}_2)\;.
\end{gathered}$$

3.3.11.2 Proof

By the definition of similarity, the contents of both states' stores are identical at reachable, untainted addresses. Thus, one of the calls *labeled* must return an address or the calls to *inputs* must match.

3.3.12 Concrete termination-insensitive labeled interference

Any traces that begin with similar states exhibit the same observable behaviors except for those labeled as insecure.

Formally, if $\pi = \langle \varsigma_1, \varsigma_2, \ldots, \varsigma_n \rangle$ and $\pi' = \langle \varsigma'_1, \varsigma'_2, \ldots, \varsigma'_m \rangle$ and $\varsigma_1 \approx_\varsigma \varsigma'_1$, then

$$\forall\ \varsigma_i \in \pi,\ \varsigma_i \notin \mathit{obs}\ \lor\ \mathit{labeled}\,(\varsigma_i) \neq \emptyset\ \lor\ \exists\ \varsigma'_j \in \pi'\ :\ \varsigma_i \approx_\varsigma \varsigma'_j\,.$$

Because the choice of traces is arbitrary, this proof considers π' as well as π.

3.3.12.1 Proof

By global transitivity of similarity, π and π' are similar. Every state in π or π', then, is similar to a state in the other trace or has a valid context taint. By the definition of *labeled*, states with valid context taints report those behaviors.

By the definition of similarity, similar states in similar traces occur in the same order.

3.3.13 Abstract noninterference

3.3.13.1 Lemma

Abstract interpretation with the given semantics detects all possible variances in externally visible behaviors.

3.3.13.2 Proof

Since the abstract semantics are a sound overapproximation of the concrete semantics, they capture the behavior of all possible executions. Since concrete executions are proven to label all termination-insensitive interference, the absence of labels reported by abstract interpretation proves noninterference.

CHAPTER 4

A POSTERIORI INFORMATION FLOW TRACKING

The additional state components in the augmented-state analysis add considerable asymptotic complexity to abstract interpretation. Both implicit taint values and the context taint map are exponential in size in the worst case. It is possible to remove these additional components from abstract states and to perform taint tracking as a separate analysis after abstract interpretation. Performing taint tracking after abstract interpretation makes implicit taint values unnecessary, which removes an exponential term from the asymptotic complexity of the analysis. This section describes a posteriori information flow tracking with augmented-state information flow tracking as its basis.

4.1 Abstract state space

In order to separate taint propagation from abstract interpretation, it is necessary to change the abstract state space. There is motivation for this in the literature; PDCFA [9] separates its continuation stack from the rest of the state in order to use a different guarantee of finiteness for continuations than for the rest of the program. In this case, taint tracking components are separated from abstract states in order to perform the taint tracking strictly after abstract interpretation.

An augmented state is a tuple of the following form:

$$\hat{\varsigma} \in \hat{\Sigma} ::= \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{ts}, \widehat{ct}\right) .$$

Restructuring this tuple preserves all of its information:

$$\hat{\varsigma}_p \in \hat{\Sigma}_p ::= \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right)$$
$$\hat{\varsigma}_t \in \hat{\Sigma}_t ::= \left(\hat{\varsigma}_p, \widehat{ts}, \widehat{ct}\right) .$$

In this restructured state space, $\hat{\Sigma}_p$ is the set of states in the small-step abstract interpreter and $\hat{\Sigma}_t$ is the set of states in the a posteriori taint analysis. The full abstract state space for the small-step abstract interpreter is detailed in Figure 4.1 and the state space for the a posteriori taint tracking analysis is detailed in Figure 4.2.

4.2 Abstract interpretation semantics

Although the a posteriori abstract state space has no additional state components, it does track the addresses read and written at each state. This preserves generality in the presence of nondeterministic allocators, as Might and Manolios [28] showed to be possible. In this case, it is impossible to calculate in retrospect which addresses were used by a particular state.

The abstract interpreter tracks reads and writes in a data structure external to the state space. Any widening scheme may be used. Mapping from addresses written to the addresses whose value influenced the write allows for the special value of $\emptyset$ to indicate that a write occurred. This simplifies the propagation of implicit taints. A rule of the form $\mathit{writes}' = \mathit{writes}\left[\hat{a}_d \overset{\sqcup}{\mapsto} \{\hat{a}_s\}\right]$ indicates that the map *writes* associated with the state in question is updated to indicate that $\hat{a}_s$ influenced $\hat{a}_d$. This data structure records the information necessary to perform taint propagation at a later time. As such, it is appropriate to think of it as storing deferred taints.

Although it is not incorporated into these simplified semantics, it is also useful to record writes that result from a call to a source.

Besides this bookkeeping, the semantics of the a posteriori abstract interpreter are identical to those of the augmented state abstract interpreter with the abstract taint store and abstract context taint map removed.

The `Const` instruction writes to an abstract stack address but reads from the instruction, not from memory:

$$
\begin{aligned}
\hat{\varsigma}_p \in \hat{\Sigma}_p &::= \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \mid \textbf{errorstate} \mid \textbf{endstate} \\
\hat{\phi} \in \widehat{FP} &\text{ is a finite set of frame pointers} \\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \to \widehat{Value} \\
\widehat{val} \in \widehat{Value} &= \hat{\mathbb{Z}} + \mathcal{P}\left(\widehat{ObjectAddress}\right) + \mathcal{P}\left(\widehat{Kont}\right) \\
\hat{z} \in \hat{\mathbb{Z}} &\text{ is a finite set of abstract integers} \\
\hat{\kappa} \in \widehat{Kont} &::= \widehat{\textbf{retk}_p}(cp, \hat{\phi}, \widehat{ka}) \mid \textbf{halt} \\
\hat{a} \in \widehat{Addr} &::= \widehat{sa} \mid \widehat{fa} \mid \widehat{oa} \mid \widehat{ka} \mid \texttt{null} \\
\widehat{sa} \in \widehat{StackAddress} &= \widehat{FP} \times \mathsf{Register} \\
\widehat{fa} \in \widehat{FieldAddress} &= \widehat{ObjectAddress} \times \mathsf{Field} \\
\widehat{oa} \in \widehat{ObjectAddress} &\text{ is a finite set of addresses} \\
\widehat{ka} \in \widehat{KontAddress} &\text{ is a finite set of addresses}
\end{aligned}
$$

Figure 4.1: Abstract state space for the a posteriori analysis

$$
\begin{aligned}
\hat{\varsigma}_t \in \hat{\Sigma}_t &::= \left(\hat{\varsigma}_p, \widehat{ts}, \widehat{ct}\right) \\
\widehat{ts} \in \widehat{TaintStore} &= \widehat{Addr} \to \mathcal{P}\left(\widehat{TaintValue}\right) \\
\widehat{tv}_p \in \widehat{TaintValue}_p &= \widehat{EP} \\
\widehat{ct} \in \widehat{ContextTaint} &= \widehat{EP} \to \mathcal{P}\left(\widehat{TaintValue}_p\right) \\
\widehat{ep} \in \widehat{EP} &::= \widehat{\textbf{ep}}\left(cp, \hat{h}\right) \mid \textbf{errorsummary} \mid \textbf{endsummary} \\
\hat{h} \in \hat{H} &= \mathbb{Z} + \{unknown\} \\
z \in \mathbb{Z} &\text{ is the set of integers}
\end{aligned}
$$

Figure 4.2: State space for a posteriori taint tracking

$$\frac{\mathcal{I}(cp) = \texttt{Const(}r\texttt{, }c\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \left(\mathit{next}(cp), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}, \text{ where}$$

$$\begin{aligned}
\widehat{sa} &= \left(\hat{\phi}, r\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \alpha_v(c)\right] \\
\mathit{writes}' &= \mathit{writes}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \emptyset\right].
\end{aligned}$$

The `Move` instruction reads one address and writes one address:

$$\frac{\mathcal{I}(cp) = \texttt{Move(}r_d\texttt{, }r_s\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \left(\mathit{next}(cp), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}, \text{ where}$$

$$\begin{aligned}
\widehat{sa}_d &= \left(\hat{\phi}, r_d\right) \\
\widehat{sa}_s &= \left(\hat{\phi}, r_s\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}(\widehat{sa}_s)\right] \\
\mathit{writes}' &= \mathit{writes}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \{\widehat{sa}_s\}\right].
\end{aligned}$$

The `Invoke` instruction performs several writes. Because writes update weakly, the order in which they occur makes no difference. Context taint is deferred entirely to the information flow tracking stage.

$$\frac{\mathcal{I}(cp) = \texttt{Invoke(}\mathit{mName}\texttt{, }r_1\texttt{, }\ldots\texttt{, }r_n\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \left(cp', \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}, \text{ where}$$

$$\begin{aligned}
cp' &= \mathit{init}(\mathcal{M}(\mathit{mName})) \\
\hat{\kappa}' &= \widehat{\mathbf{retk}}_p(cp, \hat{\phi}, \hat{\kappa}) \\
\hat{\phi}' &= \text{ is a fresh frame pointer} \\
& \quad \text{for each } i \text{ from 1 to } n, \\
\widehat{sa}_{di} &= \left(\hat{\phi}', i\right) \text{ and } \widehat{sa}_{si} = \left(\hat{\phi}, r_i\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_{d1} \overset{\sqcup}{\mapsto} \hat{\sigma}(\widehat{sa}_{s1}), \ldots, \widehat{sa}_{dn} \overset{\sqcup}{\mapsto} \hat{\sigma}(\widehat{sa}_{sn})\right] \\
\mathit{writes}' &= \mathit{writes}\left[\widehat{sa}_{d1} \overset{\sqcup}{\mapsto} \{\widehat{sa}_{s1}\}, \ldots, \widehat{sa}_{dn} \overset{\sqcup}{\mapsto} \{\widehat{sa}_{sn}\}\right].
\end{aligned}$$

The `Return` instruction writes to just one address:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Return(}r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \left(\mathit{next}\left(cp'\right), \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}, \text{ where}$$

$$\begin{aligned}
\hat{\kappa} &= \widehat{\mathbf{retk}_p}(cp, \hat{\phi}', \widehat{ka}) \\
\hat{\kappa}' &\in \hat{\sigma}\left(\widehat{ka}\right) \cap \widehat{Kont} \\
\widehat{sa}_d &= \left(\hat{\phi}', \texttt{result}\right) \\
\widehat{sa}_s &= \left(\hat{\phi}, r\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_s\right)\right] \\
\mathit{writes}' &= \mathit{writes}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \left\{\widehat{sa}_s\right\}\right] .
\end{aligned}$$

This second case for the `Return` instruction does not write at all:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Return(}r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \mathbf{endstate}}, \text{ where}$$

$$\hat{\kappa} = \mathbf{halt} .$$

The `IfEqz` instruction does not write to the store, so its semantics are particularly straightforward. As before, the two cases are not mutually exclusive.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IfEqz(}r\texttt{, }ln\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \left(cp', \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right)}, \text{ where}$$

$$\begin{aligned}
\widehat{sa}_s &= \left(\hat{\phi}, r\right) \\
cp' &= \begin{cases} \mathit{jump}\left(cp, ln\right) & \text{if } \alpha_v\left(0\right) \sqsubseteq \sigma\left(\widehat{sa}_s\right) \\ \mathit{next}\left(cp\right) & \text{if there is an integer } z \text{ such that } z \neq 0 \text{ and } \alpha_v\left(z\right) \sqsubseteq \sigma\left(\widehat{sa}_s\right) . \end{cases}
\end{aligned}$$

The `Add` instruction reads from two addresses and writes to one:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Add(}r_d\texttt{, }r_l\texttt{, }r_r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \left(next\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}, \text{ where}$$

$$\begin{aligned}
\widehat{sa}_d &= \left(\hat{\phi}, r_d\right) \\
\widehat{sa}_l &= \left(\hat{\phi}, r_l\right) \\
\widehat{sa}_r &= \left(\hat{\phi}, r_r\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_l\right) \ \hat{+}\ \hat{\sigma}\left(\widehat{sa}_r\right)\right] \\
writes' &= writes\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \left\{\widehat{sa}_l, \widehat{sa}_r\right\}\right] .
\end{aligned}$$

Abstraction of the `NewInstance` instruction is straightforward:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{NewInstance(}r\texttt{, }className\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \left(next\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}, \text{ where}$$

$$\begin{aligned}
&\widehat{oa} \text{ is a fresh object address} \\
\widehat{sa} &= \left(\hat{\phi}, r\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \widehat{oa}\right] \\
writes' &= writes\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \left\{\widehat{oa}\right\}\right] .
\end{aligned}$$

$\hat{\mathcal{T}}$ can be simplified, although it is possible to use the same metafunction with an empty context taint map.

$$\begin{gathered}
\hat{\mathcal{T}}_p : CodePoint \times \widehat{FP} \times \widehat{Store} \times \widehat{Kont} \rightarrow \\
\mathcal{P}\left(CodePoint \times \widehat{FP} \times \widehat{Kont}\right) + \{error\}
\end{gathered}$$

As with $\hat{\mathcal{T}}$, $\hat{\mathcal{T}}_p$ returns a set of tuples instead of one tuple and may also return the special value *error*.

If

$$\mathcal{H}\left(cp\right) = cp_h$$

then

$$\hat{\mathcal{T}}_p\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) = \left\{\left(cp_h, \hat{\phi}, \hat{\kappa}\right)\right\} .$$

The second case of $\hat{\mathcal{T}}_p$ shows handled exceptions:

If

$$\begin{gathered} cp \notin dom\left(\mathcal{H}\right), \text{ and} \\ \hat{\kappa} = \widehat{\mathbf{retk}_p}(cp_k, \hat{\phi}_k, \widehat{ka}) \end{gathered}$$

then

$$\hat{\mathcal{T}}_p\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) = \bigcup_{\hat{\kappa}_k \in \hat{K}} \hat{\mathcal{T}}_p\left(cp_k, \hat{\phi}_k, \hat{\kappa}_k\right), \text{ where}$$

$$\hat{K} = \hat{\sigma}\left(\widehat{ka}\right) \cap \widehat{Kont}\,.$$

In all other cases (notably, when $\hat{\kappa} = \mathbf{halt}$),

$$\hat{\mathcal{T}}_p\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) = \{error\}\ .$$

There are two nonexclusive cases for the `Throw` instruction. The first case demonstrates a caught exception:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Throw(}r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(cp', \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}, \text{ where}$$

$$\begin{aligned} \left(cp', \hat{\phi}', \hat{\kappa}'\right) &\in \hat{\mathcal{T}}_p\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \\ \widehat{sa}_s &= \left(\hat{\phi}, r\right) \\ \widehat{sa}_d &= \left(\hat{\phi}', \texttt{exception}\right) \\ \widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_s\right) \cap \widehat{ObjectAddress} \\ \hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{oa}\right] \\ writes' &= writes\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \{\widehat{sa}_s, \widehat{oa}\}\right]\ . \end{aligned}$$

In this second case for the `Throw` instruction, no handler is found and the exception reaches the top level:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Throw(}r\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \mathbf{errorstate}}, \text{ where}$$

$$error \in \hat{\mathcal{T}}_p\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right)\ .$$

The `IGet` requires three nonexclusive transition rules. In the first, the abstract frame address, abstract object address, and abstract field address are all read:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \left(\mathit{next}\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}, \text{ where}$$

$$\begin{aligned}
\exists\, oa \neq \texttt{null} : \alpha_v\left(oa\right) \sqsubseteq \widehat{oa} \\
\widehat{sa}_d &= \left(\hat{\phi}, r_d\right) \\
\widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) \cap \widehat{\mathit{ObjectAddress}} \\
fa &= \left(\widehat{oa}, \mathit{field}\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{fa}\right)\right] \\
\mathit{writes}' &= \mathit{writes}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \left\{\widehat{sa}_o, \widehat{oa}, \widehat{fa}\right\}\right] .
\end{aligned}$$

In this second case for the `IGet` instruction, an exception is thrown and caught:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \leadsto \left(cp', \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}, \text{ where}$$

$$\begin{aligned}
\texttt{null} &\sqsubseteq \widehat{oa} \\
\left(cp', \hat{\phi}', \hat{\kappa}'\right) &\in \hat{\mathcal{T}}_p\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \\
\widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) \\
\widehat{sa}_{ex} &= \left(\hat{\phi}', \texttt{exception}\right) \\
\widehat{oa}_{ex} &\text{ is a fresh object address} \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{oa}_{ex}\right] \\
\mathit{writes}' &= \mathit{writes}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \left\{\widehat{sa}_o, \widehat{oa}\right\}, \widehat{oa}_{ex} \overset{\sqcup}{\mapsto} \left\{\widehat{sa}_o, \widehat{oa}\right\}\right] .
\end{aligned}$$

In the final case for the `IGet` instruction, an exception is thrown and reaches the top level. No writes are performed.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, } r_o\texttt{, } \mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \textbf{errorstate}}, \text{ where}$$

$$\begin{aligned}
\texttt{null} &\sqsubseteq \widehat{oa} \\
\mathit{error} &\in \hat{\mathcal{T}}_p\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \\
\widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) .
\end{aligned}$$

Like IGet, IPut requires three transition rules. The first case shows execution without any exception:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IPut(}r_s\texttt{, } r_o\texttt{, } \mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(\mathit{next}\left(cp\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}, \text{ where}$$

$$\begin{aligned}
\exists\, oa \neq \texttt{null} : \alpha_v\left(oa\right) &\sqsubseteq \widehat{oa} \\
\widehat{sa}_s &= \left(\hat{\phi}, r_s\right) \\
\widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) \\
\widehat{fa} &= \left(\widehat{oa}, \mathit{field}\right) \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{fa} \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_s\right)\right] \\
\mathit{writes}' &= \mathit{writes}\left[\widehat{fa} \overset{\sqcup}{\mapsto} \left\{\widehat{sa}_o, \widehat{oa}, \widehat{sa}_s\right\}\right] .
\end{aligned}$$

The second case for IPut shows a caught exception:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IPut(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(cp', \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}, \text{ where}$$

$$\begin{aligned}
\texttt{null} &\sqsubseteq \widehat{oa} \\
\left(cp', \hat{\phi}', \hat{\kappa}'\right) &\in \hat{\mathcal{T}}_p \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \\
\widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) \\
\widehat{sa}_{ex} &= \left(\hat{\phi}', \texttt{exception}\right) \\
\widehat{oa}_{ex} &\text{ is a fresh object address} \\
\hat{\sigma}' &= \hat{\sigma}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{oa}_{ex}\right] \\
\mathit{writes}' &= \mathit{writes}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \left\{\widehat{sa}_o, \widehat{oa}\right\}, \widehat{oa}_{ex} \overset{\sqcup}{\mapsto} \left\{\widehat{sa}_o, \widehat{oa}\right\}\right] .
\end{aligned}$$

The final case for `IPut` shows an uncaught exception:

$$\frac{\mathcal{I}\left(cp\right) = \texttt{IPut(}r_d\texttt{, }r_o\texttt{, }\mathit{field}\texttt{)}}{\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \textbf{errorstate}}, \text{ where}$$

$$\begin{aligned}
\texttt{null} &\sqsubseteq \widehat{oa} \\
\mathit{error} &\in \hat{\mathcal{T}}_p \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \\
\widehat{sa}_o &= \left(\hat{\phi}, r_o\right) \\
\widehat{oa} &\in \hat{\sigma}\left(\widehat{sa}_o\right) .
\end{aligned}$$

4.3 Information flow tracking

Information flow tracking over a completed small-step abstract interpretation state graph is the process of performing all of the deferred writes recorded in the abstract interpretation phase along with context taints.

The state graph, together with the annotations about addresses that are read and written at each state, contain enough information to reconstruct all of the program's behaviors. As such, it is possible to construct the same information flows a posteriori as in the augmented-state model except that some spurious context taints do not occur. This is equivalent to constructing only the valid information flows in the augmented-state model.

Because this a posteriori taint propagation occurs after abstract interpretation has terminated, it has access to the entire state graph (and, therefore, the entire execution point graph). With the execution point graph, the taint tracking analysis can prune context taints at each step of the analysis, preventing spurious context taints from propagating. At each step, pruning occurs by removing any invalid implicit taint values. $EPG\left(\widehat{ep}\right)$ is the set of abstract execution points that succeed $\widehat{ep}$ in the execution point graph.

Using some widening scheme, a taint store is associated with each state (any widening scheme can be used). A rule such as $\widehat{ts}' = \widehat{ts}\left[\hat{a} \overset{\sqcup}{\mapsto} \left\{\widehat{tv}\right\}\right]$ indicates that the abstract taint store $\widehat{ts}$ for the state in question is updated to $\widehat{ts}'$. $\widehat{ts}'$ is not stored in place but is stored at each successor state to the state in question. If a taint store is already there, the two states merge with $\sqcup$. Timestamps are updated accordingly.

Abstract taint stores merge with the $\sqcup$ operator. It merges the taint sets at each address in the domain of either abstract taint store. Formally,

$$\widehat{ts}_1 \sqcup \widehat{ts}_2 = \widehat{ts},$$

where

$$\widehat{ts}\left(\hat{a}\right) = \begin{cases} \widehat{ts}_1\left(\hat{a}\right) & \text{when } \hat{a} \in dom\left(\widehat{ts}_1\right) \text{ and } \hat{a} \notin dom\left(\widehat{ts}_2\right) \\ \widehat{ts}_2\left(\hat{a}\right) & \text{when } \hat{a} \notin dom\left(\widehat{ts}_1\right) \text{ and } \hat{a} \in dom\left(\widehat{ts}_2\right) \\ \widehat{ts}_1\left(\hat{a}\right) \cup \widehat{ts}_2\left(\hat{a}\right) & \text{when } \hat{a} \in dom\left(\widehat{ts}_1\right) \text{ and } \hat{a} \in dom\left(\widehat{ts}_2\right) . \end{cases}$$

Similarly, there is a single, global map CTs from abstract execution points to context taint values, which are themselves maps from abstract execution points to taint values. This is equivalent to an execution-pointwise widening of context taint maps. A rule such as $CTs' = CTs\left[\widehat{ep}_c \overset{\sqcup}{\mapsto} \left(\widehat{ep}_b, \left\{\widehat{tv}\right\}\right)\right]$ indicates that the context taint map CTs is updated so that the context taint map at $\widehat{ep}_c$ now maps $\widehat{ep}_b$ to $\widehat{tv}$ in addition to any values it mapped to previously. In other words, states at $\widehat{ep}_c$ now see a context taint from a branch at $\widehat{ep}_b$ due to $\widehat{tv}$.

Pruning occurs before these updates happen; $\widehat{ep}_c$ is outside the influence of $\widehat{ep}_b$, no update happens. Pruning before writing ensures that the context taint map is monotonic and always contains only valid context taints. Formally, $CTs\left[\widehat{ep}_c \overset{\sqcup}{\mapsto} \left(\widehat{ep}_b, \left\{\widehat{tv}\right\}\right)\right]$ is equivalent to

$$CTs\left[\widehat{ep}_c \overset{\sqcup}{\mapsto} CTs\left(\widehat{ep}_c\right)\left[\widehat{ep}_b \overset{\sqcup}{\mapsto} \left\{\widehat{tv}\right\}\right]\right]$$

when $\widehat{ep}_c \in \mathit{influence}\left(\widehat{ep}_b\right)$ and equivalent to CTs otherwise.

Taint propagation would be totally uniform across states except that it is necessary to calculate which context taints are introduced at a state. For most instructions, no context taints are introduced. However, the `IfEqz`, `Throw`, `Return`, `IGet`, and `IPut` instructions can introduce context taints. In the presence of virtual dispatch, the `Invoke` instruction may also introduce context taint.

Taint propagation rules will use much of the same shorthand as abstract interpretation production rules. T_c refers to all taints from context at the state's execution point. Formally, when $\widehat{EP}_\varsigma = dom\left(CTs\left(\widehat{ep}_\varsigma\right)\right)$,

$$T_c = \bigcup_{\widehat{ep} \in \widehat{EP}_\varsigma} CTs\left(\widehat{ep}_\varsigma\right)\left(\widehat{ep}\right) .$$

Each rule operates over an abstract state $\hat{\varsigma} = \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right)$.

As with the semantics of the abstract interpretation, the appropriate map *writes* exists for each state. $dom\left(\mathit{writes}\right)$ gives the set of addresses at which *writes* is defined. For convenience, the shorthand T_w indicates the combined taints at each address in *writes* at a particular address. An abstract address $\hat{a}$, an abstract taint store $\widehat{ts}$, and a writes map *writes* all exist in context.

$$T_w = \bigcup_{\hat{a}_s \in \mathit{writes}(\hat{a})} \widehat{ts}\left(\hat{a}_s\right)$$

When a state's instruction is `Const`, all that is necessary is to add taint values to the taint store for each abstract execution point in the abstract context taint map (that is, T_c). In fact, the taint propagation rules for all instructions that do not introduce context taint are identical. Formally, when

$$\mathcal{I}\,(cp) = \texttt{Const(}r\texttt{, }c\texttt{)} \text{ or}$$
$$\mathcal{I}\,(cp) = \texttt{Move(}r_d\texttt{, }r_s\texttt{)} \text{ or}$$
$$\mathcal{I}\,(cp) = \texttt{Invoke(}mName\texttt{, }r_1\texttt{, }\ldots\texttt{, }r_n\texttt{)} \text{ or}$$
$$\mathcal{I}\,(cp) = \texttt{Add(}r_d\texttt{, }r_l\texttt{, }r_r\texttt{)} \text{ or}$$
$$\mathcal{I}\,(cp) = \texttt{NewInstance(}r\texttt{, }className\texttt{)},$$

$$\text{for every address } \hat{a} \text{ in } dom\,(writes)\,,$$
$$\widehat{ts}' = \widehat{ts}\left[\hat{a} \overset{\sqcup}{\mapsto} T_w \cup T_c\right]\,.$$

When an instruction can add context taint, an additional step is required. It occurs after propagation in the taint store. Whenever an instruction has only one successor (e.g., when a branch's condition is always true), no execution points are in its influence and the context taint is pruned. In the case of the `IfEqz` instruction, context taint can be introduced via $\widehat{sa}$. In the abstract interpretation semantics for `IfEqz`, $\widehat{ts}$ is never updated because $dom\,(writes) = \emptyset$ (assuming the widening schemes are sufficiently precise). Formally, when

$$\mathcal{I}\,(cp) = \texttt{IfEqz(}r\texttt{, }ln\texttt{)},$$

$$\text{for every address } \hat{a} \text{ in } dom\,(writes)\,,$$
$$\widehat{ts}' = \widehat{ts}\left[\hat{a} \overset{\sqcup}{\mapsto} T_w \cup T_c\right] \text{ and}$$

$$CTs' = CTs\left[\widehat{ep}'_1 \overset{\sqcup}{\mapsto} \left(\widehat{ep}_\varsigma, \widehat{ts}\,(\widehat{sa}_s)\right), \ldots, \widehat{ep}'_n \overset{\sqcup}{\mapsto} \left(\widehat{ep}_\varsigma, \widehat{ts}\,(\widehat{sa}_s)\right)\right], \text{ where}$$
$$\widehat{sa} = \left(\hat{\phi}, r\right)$$
$$\left\{\widehat{ep}'_1, \ldots, \widehat{ep}'_n\right\} = EPG\,(\widehat{ep}_\varsigma)\,.$$

A different rule can be used for `Invoke` instructions when virtual dispatch is possible. When

$$\mathcal{I}(cp) = \texttt{Invoke(}mName\texttt{,}\ r_1\texttt{,}\ \ldots\texttt{,}\ r_n\texttt{)},$$

$$\begin{aligned} &\text{for every address } \hat{a} \text{ in } dom\,(writes)\,,\\ &\widehat{ts}' = \widehat{ts}\left[\hat{a} \overset{\sqcup}{\mapsto} T_w \cup T_c\right] \text{ and}\end{aligned}$$

$$\begin{aligned} CTs' = CTs\left[\widehat{ep}'_1 \overset{\sqcup}{\mapsto} (\widehat{ep}_\varsigma, T), \ldots, \widehat{ep}'_n \overset{\sqcup}{\mapsto} (\widehat{ep}_\varsigma, T)\right], \text{ where}\\ \widehat{sa}_1 &= \left(\hat{\phi}, r_1\right)\\ T &= \bigcup_{\hat{a}\in A} \widehat{ts}\,(\hat{a})\\ A &= \left(\hat{\sigma}\,(\widehat{sa}_1) \cap \widehat{ObjectAddress}\right) \cup \{\widehat{sa}_1\}\\ \left\{\widehat{ep}'_1, \ldots, \widehat{ep}'_n\right\} &= EPG\,(\widehat{ep}_\varsigma)\ .\end{aligned}$$

The `Return` instruction cannot directly add taint to CTs but when control returns to a tainted context, CTs will still contain the context taint that existed. When

$$\mathcal{I}(cp) = \texttt{Return(}r\texttt{)} \text{ and } \hat{\kappa} = \widehat{\mathbf{retk}_p}(cp, \hat{\phi}', \widehat{ka}),$$

$$\begin{aligned} &\text{for every address } \hat{a} \text{ in } dom\,(writes)\,,\\ &\widehat{ts}' = \widehat{ts}\left[\hat{a} \overset{\sqcup}{\mapsto} T_w \cup T_c\right]\ .\end{aligned}$$

Another version of $\hat{\mathcal{T}}$ is useful for the remaining three instructions. $\hat{\mathcal{T}}_t : CodePoint \times \widehat{FP} \times \widehat{Store} \times \widehat{Kont} \times \mathcal{P}\left(\widehat{EP}\right) \to \emptyset$ is stateful; it updates CTs and then sometimes recurs. It returns nothing. It uses the shorthand $\widehat{ep}_{\mathcal{T}} = \widehat{ep}\left(cp, \widehat{SH}\,(\hat{\kappa}, \hat{\sigma}, \{\})\right)$ to refer to the execution point at the current level of stack inspection. The update to CTs is exactly as it appears; it merges context taint from $\widehat{ep}_t$ into context taint for $\widehat{ep}_{\mathcal{T}}$.

When

$$\mathcal{H}(cp) = cp_h$$

then

$$\begin{aligned} &\hat{\mathcal{T}}_t\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{EP}_t\right) =\\ \text{for every } \widehat{ep}_t \in \widehat{EP}_t,\quad &CTs' = CTs\left[\widehat{ep}_{\mathcal{T}} \overset{\sqcup}{\mapsto} CTs\,(\widehat{ep}_t) \cup T_{ex}\right];\\ &\emptyset\,.\end{aligned}$$

The second case of $\hat{\mathcal{T}}_t$ moves up one level in the stack:

If

$$cp \notin dom\left(\mathcal{H}\right), \text{ and}$$

$$\hat{\kappa} = \widehat{\mathbf{retk}_p}(cp_k, \hat{\phi}_k, \widehat{ka}),$$

then for every $\hat{\kappa}_k$ in the set $\hat{K}$ of succeeding continuations $\hat{\sigma}\left(\widehat{ka}\right) \cap \widehat{Kont}$,

$$\hat{\mathcal{T}}_t\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{EP}_t\right) = \hat{\mathcal{T}}_t\left(cp_k, \hat{\phi}_k, \hat{\kappa}_k, \widehat{EP}_t \cup \{\widehat{ep}_{\mathcal{T}}\}\right).$$

In all other cases, an error would be reached and no context taint propagation is necessary.

$$\hat{\mathcal{T}}_p\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) = \emptyset$$

With $\hat{\mathcal{T}}_t$ defined, taint propagation rules for the remaining three instructions can be defined with a single rule. This rule uses elements of $\hat{\varsigma} = \left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right)$. When

$$\begin{aligned}
&\mathcal{I}\left(cp\right) = \texttt{Throw(}r\texttt{)} \text{ and } \widehat{sa} = \left(\hat{\phi}, r\right), \text{ or}\\
&\mathcal{I}\left(cp\right) = \texttt{IGet(}r_d\texttt{, } r_o\texttt{, } \mathit{field}\texttt{)} \text{ and } \widehat{sa} = \left(\hat{\phi}, r_o\right), \text{ or}\\
&\mathcal{I}\left(cp\right) = \texttt{IPut(}r_s\texttt{, } r_o\texttt{, } \mathit{field}\texttt{)} \text{ and } \widehat{sa} = \left(\hat{\phi}, r_o\right),
\end{aligned}$$

$$\text{for every address } \hat{a} \text{ in } dom\left(writes\right),$$
$$\widehat{ts}' = \widehat{ts}\left[\hat{a} \overset{\sqcup}{\mapsto} T_w \cup T_c\right]$$

$$\text{and for every } \widehat{ep}' \text{ that succeeds } \widehat{ep}_{\varsigma} \text{ in } EPG$$
$$\text{and every } \widehat{oa} \text{ in } \hat{\sigma}\left(\widehat{sa}\right) \cap \widehat{ObjectAddress},$$
$$CTs' = CTs\left[\widehat{ep}' \overset{\sqcup}{\mapsto} \left(\widehat{ep}_{\varsigma}, \widehat{ts}\left(\widehat{sa}\right) \cup \widehat{ts}\left(\widehat{oa}\right)\right)\right]$$

$$\hat{\mathcal{T}}_t\left(cp, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right).$$

Taint propagation can occur over the state space in any order. Of the obvious choices, the depth-first ordering appears to be fastest. Iterations over this sequence proceed until no changes are made either to the taint store or to the map of context taints, where each execution point has a unique context taint.

4.4 Equivalence to augmented-state analysis

The a posteriori analysis tracks the same information flows as does the augmented-state analysis. This equivalence is most easily demonstrated with an intermediate analysis that performs the augmented-state analysis as if the execution point graph were already available. This prescient augmented-state analysis is referred to as the **valid-only** analysis. The two equivalent analyses must have the same widening scheme; in particular, the context taint maps in the augmented-state analysis and in the valid-only analysis must be execution-pointwise widened to match the widening of the a posteriori analysis.

4.4.1 Transitivity of taint invalidity

4.4.1.1 Lemma

No taint value derived from an invalid taint, including context taints at locations outside of the influence of their respective execution points, is valid.

4.4.1.2 Proof

This proof is presented by cases.

4.4.1.2.1 Explicit taint values. This lemma is vacuously true in the case of explicit taint values, which are always valid.

4.4.1.2.2 Copying of implicit taint values. An invalid implicit taint value may be copied from one address to another. Validity is defined independently of the address at which the taint is stored, so every copy of an invalid taint is still invalid.

4.4.1.2.3 Derivation of implicit taint values. It is also possible to create a new implicit taint value from an existing implicit taint value with $\widehat{implicit}$, as follows: $\widehat{implicit}\left(\widehat{ep}_b, \widehat{ep}_a, \widehat{itv}\right) = \widehat{itv}'$. By the definition of *valid*, if $\widehat{itv}$ is not valid, a pair of abstract execution points $(\widehat{ep}_{bt}, \widehat{ep}_{at})$ must exist in $\widehat{itv}$ such that $\widehat{ep}_{at} \notin \mathit{influence}\,(\widehat{ep}_{bt})$. By the definition of $\widehat{implicit}$, $\widehat{itv}'$ contains $(\widehat{ep}_{bt}, \widehat{ep}_{at})$ and so is also invalid.

4.4.1.2.4 Implicit values from context taints. An abstract implicit taint value may be created from a context taint via $\widehat{implicit}$. When the abstract execution point $\widehat{ep}$ at which the abstract implicit taint value is created is outside of the influence of the abstract execution point $\widehat{ep}_b$ of the context taint, the resulting abstract implicit taint value $\widehat{itv}$ contains $(\widehat{ep}_b, \widehat{ep})$. Because $\widehat{ep} \notin \mathit{influence}\,(\widehat{ep}_b)$, $\widehat{itv}$ is not valid.

4.4.2 Equivalence of CESK state space

4.4.2.1 Lemma

The CESK state graph produced by projecting the augmented-state graph onto a CESK state space is identical to the state graph produced by the state graph produced by the first phase of the a posteriori analysis. This CESK state graph is also equivalent to the graph produced by projecting the state graph from the valid-only state graph onto a CESK state space.

4.4.2.2 Proof

By induction on the structure of the summarized Dalvik bytecode language semantics, the semantics of the augmented-state analysis are identical to those of the a posteriori analysis except for taint propagation and bookkeeping. Since no CESK value ever depends on a taint value in any form or on the external bookkeeping performed by the a posteriori analysis, the differences between these graphs are discarded in the projection onto the CESK state space. The augmented-state analysis and valid-only analysis differ only in components that are discarded during projection to the CESK state space.

4.4.3 Equivalence of augmented-state analysis to valid-only analysis

4.4.3.1 Theorem

The valid information flows identified by the augmented-state analysis are identical to the information flows identified by the valid-only analysis.

4.4.3.2 Proof

The difference between the augmented-state analysis and the valid-only analysis is that the former propagates and then removes taints once they are shown to be invalid. As such, the taints in the valid-only analysis are a subset of the taints in the augmented-state analysis before removal.

Lemma 4.4.1 shows that if a taint value or context taint is invalid, every taint value that results from it is also invalid. Since the only taints that can differ between the two analyses are taints in the augmented-state analysis that are invalid and since all invalid taints are subsequently removed, the two analyses identify exactly the same information flows.

This remains true if the augmented-state analysis sees more states than the valid-only analysis; the additional states can only differ by additional, invalid, taints. Crucially, there are no differences in the CESK components of these additional states; Lemma 4.4.2 holds.

4.4.4 Equivalence of taint propagation

4.4.4.1 Lemma

For each state $\hat{\varsigma}$ in the CESK state graph, the same taints propagate in the valid-only states that project to $\hat{\varsigma}$ as do in the a posteriori analysis when propagating from the aggregate of the valid-only states' taint stores and context taint if the two analyses allocate the same addresses.

Formally, the set $\hat{\Sigma}_V = \{\hat{\varsigma}_1, \ldots, \hat{\varsigma}_n\}$ of states in the valid-only analysis contains all states from the state space of this analysis that project to $\hat{\varsigma}$. Their respective abstract taint stores are TS and the aggregation of their respective taint stores is

$$\widehat{ts}_\sqcup = \bigsqcup_{\widehat{ts}_i \in TS} \widehat{ts}_i \, .$$

Their respective context taint maps are CT and the aggregation of CT is

$$\widehat{ct}_\sqcup = \bigsqcup_{\widehat{ct}_i \in CT} \widehat{ct}_i \, .$$

The set $\hat{\Sigma}'_V = \{\hat{\varsigma}'_1, \ldots, \hat{\varsigma}'_m\}$ is the set of successors to the states in $\hat{\Sigma}_V$. The aggregation $\widehat{ts}'_\sqcup$ is defined analogously.

Context taint maps are aggregated analogously at each successor's execution point. For each abstract execution point $\widehat{ep}_i$ that succeeds the execution point $\widehat{ep}_p$ at $\hat{\varsigma}$, an abstract context taint map $\widehat{ct}_i$ is the aggregate of the abstract context taint maps of states at this abstract execution point.

Assuming (without loss of generalization) that the two analyses allocate addresses identically, performing a posteriori taint propagation on $\hat{\varsigma}$ with taint store $\widehat{ts}_\sqcup$ and with the context taint map $\widehat{ct}_\sqcup$ at $\widehat{ep}_p$, results in a taint store $\widehat{ts}'_\sqcup$ and the results in a context taint map $\widehat{ct}_i$ at each abstract execution point.

4.4.4.2 Proof

This lemma claims equivalence when the allocations are identical. The analyses are executed in isolation; usually, only the a posteriori analysis executes. In essence, this agnosticism towards allocation strategies creates an entire class of analyses; this proof considers matched pairs of analyses from the valid-only class of analyses and the a posteriori family of analyses and proves each pair's equivalence. This assertion does not lessen generality.

In the augmented-state and valid-only semantics, every taint propagation within the taint store copies taints from one address onto another. In the a posteriori semantics, these same addresses are stored in the *writes* map and are then propagated via T_w. Similarly, taints added to context in the valid-only semantics are added to context in the taint propagation phase of the a posteriori analysis. Taints from context in the valid-only analysis propagate in the a posteriori analysis via T_c.

The abstract taint store that results from taint propagation in the a posteriori analysis is $\widehat{ts}_p$. Every taint in $\widehat{ts}'_{\sqcup}$ either exists in $\widehat{ts}_{\sqcup}$ or propagated via the semantics of the augmented-state analysis. If it exists in $\widehat{ts}_{\sqcup}$, it must exist in $\widehat{ts}_p$. If it does not, then it either propagates via *writes* and T_w (reading from $\widehat{ts}_{\sqcup}$) or is created from context (reading from $\widehat{ct}_{\sqcup}$). By induction on the respective semantics, these propagations are identical, so $\widehat{ts}_p$ must be equivalent to $\widehat{ts}_{\sqcup}$.

Similarly, taints added to context originate from $\widehat{ts}_{\sqcup}$ and identical propagation rules in the respective semantics, so the abstract context taint map at each execution point $\widehat{ep}_i$ must be $\widehat{ct}_i$.

4.4.5 Equivalence of a posteriori analysis to valid-only analysis

4.4.5.1 Theorem

The information flows identified by the a posteriori analysis are identical to the information flows identified by the valid-only analysis.

4.4.5.2 Proof

By Lemma 4.4.2, both analyses see the same sources, so every taint that originates in one analysis also originates in the other.

Also by Lemma 4.4.2, every state seen in the a posteriori taint tracking phase is identical to the projection onto the CESK state space of at least one state in the valid-only analysis. Similarly, every state in the valid-only analysis has an analogue in the a posteriori taint tracking phase. Because both proceed to a fixed point, both calculate a transitive closure, which is agnostic to the order of propagation. By induction on Lemma 4.4.4, the same taints must propagate at each of these states.

4.4.6 Equivalence of information flows

4.4.6.1 Theorem

The information flows identified by the a posteriori analysis are identical to the information flows identified by the augmented-state analysis.

4.4.6.2 Proof

By Theorem 4.4.3 and Theorem 4.4.5.

4.5 Complexity

The asymptotic complexity of a small-step abstract interpreter is measured by the complexity of its state space. Because the a posteriori analysis has no need of special values for implicit taints, its state space is smaller than that of the augmented-state analysis. Since implicit taint values contain a set of pairs of execution points, the removal of implicit taint values from the state space removes two exponential terms from the state space: one from the taint store and one from the context taint map.

In addition, the state space of the abstract interpreter in the a posteriori analysis is much smaller than that of the augmented-state analysis, so the calculation of the execution point graph (quadratic in the size of the state graph) is less complex. In addition, the removal of spurious context taints could have a similar effect on runtime as does abstract garbage collection [29]; by removing spurious flows, it may improve performance in a way not predicted by asymptotic complexity.

CHAPTER 5

IMPLEMENTATION

Correctly implementing an analysis for Dalvik bytecode requires attention to minute details. Making this analysis fast and precise requires other modifications. This section describes some of the salient details of the abstract interpreter and taint tracking mechanism.

5.1 Scaling up to full Dalvik bytecode

Although the abstract syntax and its accompanying semantics are for a summarized representation of Dalvik bytecode, the abstract interpreter operates on Dalvik bytecode. In order to adapt this analysis to full Dalvik bytecode, it is necessary to recognize the aspects of Dalvik that the abstract syntax does not represent.

5.1.1 Data types

The syntax and semantics represent only integers and objects. Dalvik bytecode has floats, doubles, longs, ints, booleans, bytes, shorts, objects (including `String` objects, which get some special treatment), and arrays. In the Dalvik semantics, only the first four are true primitives; the other integral types are stored in int registers. The interpreter takes advantage of the fact that there is no implicit type conversion in Dalvik bytecode and stores data of disparate types separately; updating the abstract store with an integer value has no effect on any float registers that may be stored there. It is possible that an attacker could manipulate bytecode after compilation to take advantage of this optimization. Arrays require an address system similar to that of objects.

5.1.2 Instructions

There are thirteen variants of the `Move` instruction. Some of them contain type information; for example, `MoveObject` moves a reference (either an object or an array) from one register to another. Others specify the bit width of the information being moved; `MoveWide` moves a pair of registers to another pair of registers and is suitable for moving longs and doubles.

Line numbers are not part of the Dalvik bytecode specification; the bytecode uses offsets into code. In conformance with the parser from the Tapas [17] project, which performs the Dalvik bytecode parsing, the abstract interpreter uses line numbers as opposed to labels. Code points in the interpreter for full Dalvik bytecode are represented as pairs of a line number and an encoded method. Additionally, there are special singleton objects for the end position, which is reached upon invocation of the halt continuation, and an error position.

Similar straightforward generalizations exist for other instructions.

5.1.3 Methods

Dalvik bytecode differentiates between methods and "encoded methods." Invocation instructions have a method ID, which is an index into an array of methods. Each method has a prototype, a name, and a class ID used to identify the class that defines the method. Each encoded method includes a method id, flags such as private or final, and an offset into the code table. From that offset, it is possible to ascertain the instructions, register count, and exception handling information for a method.

Arguments, including the receiver, if there is one, are placed at the end of the invocation frame in order and registers are zero-indexed. For example, a method with a register count of six that takes two (single-register) arguments and a receiver expects the receiver to be in virtual register 3 and the two arguments to be in virtual registers 4 and 5. Registers 0, 1, and 2 are general-purpose registers.

Virtual method dispatch requires the type of the receiver. If the class data item (indexed with an ID from the class definition item) has an encoded method whose ID matches the method ID in the invocation instruction, that encoded method is used. If not, the superclass (whose ID is stored in the class definition) is examined and the process repeated until a matching method is found.

Methods in Dalvik bytecode can take up to 256 arguments. There are five kinds of invocations: direct, static, interface, virtual, and super. Direct methods are methods such as private methods that cannot be overridden and require no type lookups. Additionally, each kind of invocation has two instructions: a standard invocation that uses between zero and five registers for its arguments and a range instruction that specifies a range of consecutive registers as its arguments.

5.1.4 Exceptions

The provided semantics represent exception handling in Dalvik bytecode faithfully except that they omit exception types. Checking handlers' types against thrown exceptions is straightforward.

5.1.5 Concurrency

The abstract interpreter's computational model is not concurrent. As such, it treats monitor instructions as no-ops.

5.2 Modularity

The abstract interpreter is flexible by design. Many of the design choices available to the designers of small-step abstract interpreters have been implemented and may be selected with command-line parameters. For example, it has built-in support for global store widening, in which all states share a common store. It also supports pointwise widening, in which all states at any particular code point share a common store. Naturally, it also supports no store widening, in which each state has a store. Other widening schemes can be implemented with a subclass of `Widener` and used by adding a command line pattern to the appropriate argument in the `Settings` class.

There is also considerable flexibility in the allocators for frame pointers, arrays, objects, and continuation addresses and in the abstractions for primitive values. Since `String` objects are treated specially (e.g., there are literal `String` objects but no other literal objects), a separate allocator can be specified for `String` objects than for other objects. Allocation strategies include $alloc_{P4F}$, $alloc_{0CFA}$, $alloc_{cp}$, and $alloc_{\top}$. Additional allocation strategies may easily be added by creating subclasses of `KontAllocator`, `ObjectAllocator`, and so forth.

The abstract interpreter currently supports four styles of abstraction for primitives and more can be added by creating a subclass of the appropriate class; for example, a new abstraction for integers could be created with a new subclass of `IntRegister`. The implemented abstractions are $\alpha_{\top}$, α_{INT32}, α_I, and α_c. Each of these abstractions is described in Section 2.2.1.1. Analogous abstractions exist for long integers and single- and double-precision floating point numbers.

If the store were adapted to allow for strong updates instead of weak updates and if concrete allocators were implemented, it could change from an abstract interpreter to a concrete interpreter by changing its runtime configuration.

5.3 Register deallocation

The Android SDK coalesces virtual registers during compilation, which leads to an imprecise analysis. This can be ameliorated by reversing the process; performing a liveness analysis demonstrates which virtual registers can be separated. This is similar to a single static assignment transformation without phi nodes; instead of inserting a phi node, the register in question is simply not separated.

Use-def chains can be used in place of bare registers in frame addresses to attain the same benefits to precision without requiring transformation of the source code. This change has the effect of reversing much of the imprecision caused by the register coalescing done at compile time. Because it distinguishes registers by where they are accessed in the program, it also reverses much of the imprecision in frame addresses caused by global store widening.

5.4 Preparation

In Java, addresses in memory must be written before they are read. This invariant is ensured by a process called **preparation** by the Java virtual machine specification [32]. Preparation precedes initialization and writes default values to instance and class members. Default values are either `null` or `0`, depending on the type of the member. In Dalvik bytecode, per the Dalvik virtual machine specification [16], `null` is represented with the number 0, so preparation writes zeros to all members. In Dalvik bytecode, which is a register-based machine instead of a stack-based machine, default values are written to local variables in a similar way to preparation for class and instance members.

Preparation leads to imprecision in the abstract interpreter because its semantics use weak updates; the value at every address that is prepared must include 0. This imprecision can be avoided by skipping preparation in cases where the default value must be overwritten before it is ever read. For example, when a new object is created, its constructor is usually called in the next instruction. In this case, the object can be prepared by retroactively writing zero to every member not written upon returning from the constructor. As long as every execution does so, all values that may be unwritten in any execution of the constructor are retroactively prepared.

In order to ensure correctness against a pathological corner case, it is necessary to ensure that reads during a constructor or class initializer always return zero in addition to any values in the store.

CHAPTER 6

EMPIRICAL RESULTS

6.1 Methodology

The test suite of applications for measurement comes from the Automated Program Analysis for Cybersecurity (APAC) program. APAC's purpose was to develop tools that could be used to vet software for a curated app store, allowing only those programs it deemed to be secure. The teams developing these tools tested them at periodic engagements, where the program provided applications for them to analyze. The twelve applications in the test suite are the entire set of applications from one engagement. All applications from the test suite were built for Android 4.4.2.

In order to ensure that comparisons between the a posteriori analysis and the augmented-state analysis, the augmented-state analysis was implemented as a branch of the a posteriori analysis. All changes made to one branch were applied to the other, except where those changes were not applicable.

In order to ascertain the analyses' correctness and precision, they were compared against manually measured ground truth. In the case of Filterize, there was a small enough number of sources and sinks that it was possible to measure the ground truth exhaustively. In the case of the other applications, samples of the sources and of the sinks were taken at random. Flows that depended on knowledge of the semantics of the Android standard library were marked as out of bounds and were not counted as either positives (possible flows) or as negatives (impossible flows). There were a total of seven such flows identified: four in the exhaustive count of flows in Filterize and three in the random sampling for chatterbocs.

Analyses were executed on a server with 12 cores and 64 GB of RAM. It runs MacOS 10.8.5, Scala 2.11.7, and Java 7. Only four analyses ran at once in an attempt to ensure that there would be no contention for hardware resources.

All analyses used the same configuration, which seems to be reasonably well optimized for speed and precision. The configuration includes global store widening and no abstract garbage collection. It uses $alloc_{P4F}$ for continuation addresses and $alloc_{cp}$ for objects (including `String` objects), arrays, and frame pointers. Integers are abstracted using α_I, where I includes the numbers from -1 to 10 inclusively and all numbers used as IDs in the program's layout XML; that is, if XML is the set of IDs in layout XML, $I = \{-1, 0, \ldots, 10\} \cup XML$. Other primitive types are abstracted with $\alpha_\top$. Analyses timed out after 24 hours and each had 3 GB of RAM available to it. Taints are sets of labels that identify the location in the program where they originated.

In one case, a program timed out after finishing abstract interpretation. As this program had exhibited variability in its execution time, it was run again. As timing out yields no runtime measurement, no attempt was made to average different measurements; instead, its measurement in Table 6.1 is marked with an asterisk (*). Section 7.2 discusses the cause of the variability in execution time.

6.2 Results

The augmented-state analysis timed out or ran out of memory on each application in the test suite.

Table 6.1 shows the space and time metrics of the a posteriori analysis on the test suite. It lists the number of states found and instructions covered by abstract interpretation, the time spent performing abstract interpretation (excluding initialization, parsing, and information flow), and the time spent on the entire analysis.

Table 6.2 shows the precision metrics of the a posteriori analysis on the test suite. The column labeled TP (true positives) indicates the number of information flows that can occur in each app, as identified by manual analysis, that the analyzer correctly identified. The column labeled FP (false positives) indicates the number of information flows that cannot occur in each app that the analyzer failed to prove impossible. The column labeled TN (true negatives) indicates the number of information flows that cannot occur that the analyzer correctly proved to be impossible. The column labeled FN (false negatives) indicates the number of information flows that can occur that

Application	States	Instructions	AI time (s)	Total time (s)
BattleStat	3951	2117	45.9	361.2
chatterbocs	–	–	–	–
ConferenceMaster	8926	3150	884.7	1149.9
Filterize	3405	1460	14.4	27.2
ICD9	–	–	–	–
keymaster	13708	4574	28678.5	30175.1
Noiz2	–	–	–	–
PassCheck	10865	4911	341.9	460.6
pocketsecretary	48962	5421	53879.9*	56550.4*
rLurker	1105	915	9.6	22.7
splunge	–	–	–	–
Valet	1791	1445	30.9	41.0

Table 6.1: Space and time measurements

the analyzer failed to identify as potentially possible. There were no false negatives, as is expected as a result of the proof of noninterference.

Additionally, Table 6.2 contains a total number of information flows from its first four columns and the percentage of the total flows that were false positives (FP %) or true negatives (TN %).

6.3 Analysis

These results demonstrate that it is possible to gain automated assurance about information flows in an expressive low-level language. They also demonstrate what work remains to be done to this end.

6.3.1 Interpretation

Since the analysis is exponential, it is expected that some programs will fail to terminate in a practically useful period of time. The applications in the test bench are mostly of moderate size, when measured either by total instructions or by covered instructions (Table 6.3). Since abstract interpretation covers all reachable instructions, instructions not covered are dead code. Each of the applications that times out has a large number of total instructions, which is unsurprising. They could be either larger or smaller than typical applications in some other corpus. It would, of course, be desirable to finish analysis on more applications. However, it is already clear that many applications can be analyzed in this way.

Since the analysis produces automated assurance about the absence of information flows, precision in its results is best measured in the total number of true negatives. Each true negative is an information flow that a human analyst need not check. In this light, even the least precise analysis (keymaster, with only 64% true negatives) saves a human analyst a great deal of effort. The four most precise analyses identify a stunning 92% true negatives, which means that the vast majority of an analyst's effort has been saved.

Viewed through the eyes of false positives, which is a more natural fit for analyses that assist in the identification of bugs, the analysis is still promising. The worst of the twelve applications reports 32% false positives, which is moderate but acceptable. Two of them report as few as one false positive.

Application	TP	FP	TN	FN	Total	FP%	TN%
BattleStat	1	1	23	0	25	4%	92%
chatterbocs	–	–	0	–	–	–	0%
ConferenceMaster	0	3	22	0	25	12%	88%
Filterize	4	33	263	0	300	11%	88%
ICD9	–	–	0	–	–	–	0%
keymaster	1	8	16	0	25	32%	64%
Noiz2	–	–	0	–	–	–	0%
PassCheck	0	2	23	0	25	8%	92%
pocketsecretary	0	4	21	0	25	16%	84%
rLurker	0	2	23	0	25	8%	92%
splunge	–	–	0	–	–	–	0%
Valet	1	1	23	0	25	4%	92%

Table 6.2: Precision metrics: true positives (TP), false positives (FP), etc.

Application	Total instructions	Instructions covered	% covered
BattleStat	3460	2117	61.2%
chatterbocs	22146	–	–
ConferenceMaster	34543	3150	9.1%
Filterize	2913	1460	50.1%
ICD9	44820	–	–
keymaster	8985	4574	50.9%
Noiz2	17452	–	–
PassCheck	17588	4911	27.9%
pocketsecretary	8648	5421	62.7%
rLurker	1580	915	57.9%
splunge	136848	–	–
Valet	2864	1445	50.5%

Table 6.3: Total instructions in each application

6.3.2 Comparing against random chance

Another way these results could be measured is against random chance, in the spirit of Zitser, Lippman, and Leek [40]. This would require the construction of a feasible null analysis that performs consistently with intuitions about what random chance means. It would be beneficial to design an analysis that can actually be constructed.

One such analysis is an analysis that, for each pair of a source and a sink in a given program, flips an unweighted coin. This analysis has the advantage of being just as likely to succeed as to fail for any potential information flow. As such, it is expected that it will be accurate 50% of the time, regardless of how many information flows are actually possible or impossible. A confidence interval can easily be created, given an arbitrary p value, against which the results of the information flow analysis can be compared.

This analysis has several shortcomings. First, it treats all inaccuracies equally. While this is appropriate for many analyses, it is not true in the case of an analysis designed for assurance, where a single false negative invalidates the analysis and where false positives are mere inconveniences. Second, it fails to account for the fact that information flows are not entirely independent of each other; for example, an information flow from an unreachable source cannot be realized, regardless of its sink. Third, the choices of coin weight and p value are entirely arbitrary.

This analysis predicts that our model of random chance would be correct 12.5 times for an analysis with 25 information flows. The probability that 7 or fewer unweighted coin flips would be incorrect is 0.022; the probability that 8 or fewer unweighted coin flips would be incorrect is 0.054. Accordingly, any analysis with 25 potential information flows that is incorrect at most 7 times is better than random chance plus its random interval (for a p value of 0.05). This is true of all but keymaster, which is just shy of the mark at 8 false positives. For Filterize, the probability that 135 or fewer flows would be incorrect is 0.047; the probability that 136 or fewer flows would be incorrect is 0.059. Filterize is incorrect 33 times, which accuracy would be obtained with an unweighted coin with probability 7.06×10^{-48}. Table 6.4 has complete data.

An analysis with a weighted coin that matches the frequency of potential information flows that can be realized seems more suitable at first blush, but quickly

Application	Inaccuracies	Flows measured	Probability
BattleStat	1	25	7.75×10^{-7}
chatterbocs	–	–	–
ConferenceMaster	3	25	7.82×10^{-5}
Filterize	33	300	7.06×10^{-48}
ICD9	–	–	–
keymaster	8	25	0.054
Noiz2	–	–	–
PassCheck	2	25	9.72×10^{-6}
pocketsecretary	4	25	4.55×10^{-4}
rLurker	2	25	9.72×10^{-6}
splunge	–	–	–
Valet	1	25	7.75×10^{-7}

Table 6.4: A comparison of accuracy against an unweighted coin

degenerates in extreme cases. A weighted coin for an application, such as rLurker, pocketsecretary, PassCheck, or ConferenceMaster, for which no measured information flow could be realized, would always assert that an information flow cannot be realized. As such, its expectation would be perfect accuracy and its error bound would be of size zero. A single false positive in the analysis would not only be below the measurement of random chance; it would be below the error bound. Since it is expected that there will sometimes be false positives, this comparison is of little use.

6.3.3 Future work

One way many analyses improve their speed is by compromising precision. While this approach is often useful, initial results indicate that it may not be helpful for this analysis. It is possible to skip the taint tracking entirely and use only abstract interpretation to report reachable sources and sinks and then report that all of them reach each other, but this would be very imprecise. Also, abstract interpretation dominates analysis time for most applications, so it would also save little time. Similarly, in initial tests, removing precision from the abstract interpreter did not seem to be very useful. This is consistent with the results of abstract garbage collection [29], which improves precision at the cost of asymptotic complexity but generally improves empirically measured runtime.

Another common solution is to return immediately with little or no information. While this is possible with many abstract interpreters, the taint tracking phase requires a complete mapping from states to addresses written and read. It might be possible to improve speed by removing all precision from the abstract interpreter, effectively reducing it to the generation of an interprocedural control flow graph. This could, however, introduce more spurious states and result in a yet slower analysis.

The analysis itself may admit optimization. P4F [14] serves as an example of such an optimization to abstract interpretation; it provides perfect precision with regards to function returns at no extra asymptotic computational cost (when the continuations in the store are widened globally). As such, it removes spurious states without affecting worst-case complexity. In addition, it may be possible to rewrite the taint tracking ordering to improve its speed. Simplifications to the memory model could also improve its efficiency.

6.3.4 Conclusion

There is still a great deal of work that could be done to improve both speed and complexity in this analysis. However, its performance as it stands is more than sufficient to demonstrate that it is fast enough and precise enough to be useful to analysts who desire a high degree of assurance. The work that remains to be done indicates that there is even more reason for optimism, as both speed and precision could improve even further, making the analysis more broadly applicable.

CHAPTER 7

DISCUSSION

7.1 Guarantees

The results of this empirical evaluation demonstrate the utility of the analysis. A human analyst charged with proving the security of information in an application could use this analysis to save a great deal of time on a small- or moderate-sized application. It is trivial to scan a program for relevant sources and sinks; without this tool, the analyst would have to evaluate all pairings of sources and sinks to prove that information from the sources never reaches the sinks.

On all of the small- and moderate-sized applications in the test suite, the majority of these pairings (as measured in the sample) was demonstrated to be safe with the aforementioned caveats: it cannot detect termination leaks or behaviors that depend on library code or on bytecode manipulation to circumvent Java's type system. It is possible to warn the analyst in these cases. This analysis does not save time on all applications and does not eliminate all of the work a human analyst would have to do, but the results suggest that it does eliminate the majority of the work a human analyst would be required to do without it.

7.2 Nondeterminism

Both analyses are sound, although their precision can vary from invocation to invocation. Because this nondeterminism concerns only overapproximated behaviors, it is an issue of precision and not of soundness. This nondeterminism is not manifest in the programs in the test suite with the current configuration but is theoretically possible. All of these nondeterministic behaviors occur in part because Scala's set operations happen in nondeterministic order. Entry points, class initializers, and the state exploration queue are all stored in sets and, as such, the order of the operations based on these data cannot be predicted at compile time. Nondeterminism can be avoided by forcing these operations to happen in order.

One situation in which nondeterminism is manifest is the use of stateful primitive abstractions. Choosing not to abstract the first n integers encountered and then to abstract the remainder to $\top$ is sound. However, different sets of integers might be abstracted to $\top$ in different executions. This could cause the analysis to explore a branch that cannot be reached by any concrete execution in some executions of the abstract interpreter but not in others.

One result of nondeterministic order in exploration of the state graph is variability in the amount of time that state exploration takes. Lyde and Might [27] demonstrated a large degree of variance in state exploration time in the presence of store widening with several different orderings. They also showed that there is no one clear choice for state graph ordering; different programs finish fastest with different exploration orders.

7.3 Generalized state graph postprocessing

The augmented-state approach to analysis is intuitive but proved to be inefficient. When designing future analyses based on small-step abstract interpreters, it is likely that it will be similarly practical to perform the analysis after abstract interpretation rather than extending the state space. For example, it is likely that this same technique could be applied to abstract counting [29].

7.4 Analysis–agnostic noninterference

The separation of taint tracking from abstract interpretation suggests that it may be possible to further distinguish the two analyses from each other. It may be possible to perform any abstract interpretation on any language and then to perform a taint tracking analysis on the results of that interpretation. This separation would make it possible to prove noninterference in a program with any sound abstract interpreter without additional theoretical work.

CHAPTER 8

RELATED WORK

8.1 Seminal work

Sabelfeld and Myers [34] summarize the literature on information flows done by 2003. This literature review summarizes the problem of information flows and presents common solutions to it. In particular, it presents explicit flow tracking as a type system and adds program counter taint to this type system. Sabelfeld and Myers also describe noninterference.

Denning [7] introduces the idea of taint values as lattices instead of booleans. Using a lattice of taint values means that adding taint to an already tainted value means simply finding the least upper bound of the two taints.

Denning and Denning [8] describe a static analysis that guarantees noninterference (although they do not use the term) on a simple imperative language. They point out that control flow analysis is necessary to apply their technique to languages with jump statements and describe the necessary analysis briefly. In addition, they describe how arrays and data structures can be added to their analysis framework. They do not, however, consider function calls or exceptional flow. This paper provides a theoretical framework and a strong example for the many information flow analyses that succeed it. This work is an extension of these ideas to richer languages.

Volpano et al. [37] validate the claims of Denning and Denning. Volpano and Smith [38] then extend it to handle loop termination leaks and some exceptional flow leaks. Their language lacks function calls and use expressions instead of jumps, each of which adds a significant layer of complexity to exceptional flow. They treat only arithmetic exceptions resulting from division in their simple language. As with prior work, they use a type system and ignore jump statements.

Cavallaro et al. [5] dismiss the effectiveness of static techniques in their introduction. They then discuss the shortcomings of dynamic analyses, particularly against intentionally malicious code.

8.2 Entry-point saturation

Android programs use an implicit main function provided by the Android framework. They define event handlers and the Android system invokes them as is appropriate. As such, injection is nontrivial. Liang et al. [23] introduce **entry-point saturation**. Entry-point saturation involves injecting into each entry point of the program and analyzing to completion. The analyzer continues to inject into each entry point in turn until it analyzes all entry points without a change to the store; in other words, it reaches a fixed point.

Additionally, new entry points may be encountered during analysis, as Android programs can register event handlers dynamically. These additional entry points are added to the queue and invoked repeatedly until a fixed point is reached.

Entry-point saturation relies on stores with weak updates to guarantee soundness. Reaching a fixed point means that all possible program behaviors have been encountered—for all possible interleavings of the entry points. It does not address concurrent execution of event handlers.

8.3 Partial solutions

Many related works address information flows but do not address all implicit flows. Several of them track only explicit flows while others track some implicit flows but make no attempt on others.

8.3.1 Explicits only

Chang et al. [6] present a compiler-level tool that transforms untrusted C programs into C programs that enforce specified policies. They make no attempt to address implicit information flows.

Kim et al. [21] perform an abstract interpretation on Android programs. They desugar Android programs to an intermediate language they call Dalvik Core to simplify their analysis. They then demonstrate their work empirically on 90 applications and discuss their precision and performance. They claim soundness but do not claim or prove noninterference. They do not discuss or address implicit information flows.

Arzt et al. [2] present FlowDroid, a static analyzer for Android applications. They model application lifecycles by parsing the manifest and monitoring asynchronously executing components and callback registration. They explicitly state that they do not address implicit information flows (although a blog post for the project says that they have added support for implicits). The implementation of the analyses in this work originally parsed Android's XML files, which are stored in a binary format, using FlowDroid's library. The analyzer now uses its own code.

8.3.2 Some implicits

Xu et al. [39] perform a source-to-source transformation on C programs to instrument them for taint tracking. The resulting programs identify explicit information flows. They also discuss implicit information flows and their rationale for mostly ignoring them. They do track implicit flows in conditional expressions and array dereferences where the index is tainted.

The authors discuss optimizations specific to their implementation and the situations in which they can be safely used. Because C allows for pointer arithmetic and because their taint labels occur in program space, their instrumentation also must guarantee that the program cannot write taint labels.

C programs instrumented using this system can execute function calls and exceptional control flow in the same way that all C programs do. However, implicit information flows from if statements, function calls to pointers, exceptional control flow, and virtually any other control flow mechanism are ignored.

Kang et al. [20] perform a dynamic analysis called DTA++ that operates on Windows x86 binaries and tracks information flows. Their work focuses on unintentional implicit information flows in benign programs and emphasizes the need to avoid false positives—so much so that they allow false negatives. As such, this work presents a closer approximation to soundness than does a strictly explicit taint tracking mechanism but does not provide guarantees of noninterference.

The implicit flows targeted in DTA++ are those where few inputs cause the program to execute a branch and the remainder cause it to execute another. This is a logical choice, as the program behaviors along the first branch can give attackers relatively precise information.

DTA++ operates in two phases. In the first, traces from test cases generated either manually or by some other tool are used to create DTA++ rules, or specifications for additional taints to be made when certain branch conditions are tainted. The second phase performs ordinary explicit taint propagation and implicit taint propagation based on DTA++ rules.

Liang and Might [24] present a Scheme-like core calculus for scripting languages like Python. Their core language is expressive enough to contain not only function calls but also call/cc as a primitive. They present both concrete and abstract semantics for an A-normalized form of their core calculus and demonstrate bisimulation but do not claim or prove noninterference. Any treatment of implicit information flows is handled strictly by syntax and is not applicable to a language whose branches are jumps.

8.4 Work that tracks implicit flows soundly

Few works in the literature present systems that guarantee noninterference. Those that do guarantee noninterference often do so tangentially. As a result, the analyses tend to operate on simple languages that are not intended for general use.

8.4.1 Giacobazzi and Mastroeni

Giacobazzi and Mastroeni [12] demonstrate an abstract interpreter on programs in IMP. Their interpreter exists only as a platform for the rest of their paper. They demonstrate the formalization of noninterference properties, discuss declassification in such a framework, and describe in detail how to compute a maximally powerful attacker for which a particular program is secure.

The purpose of this work is not to present a practical analysis for a real-world language but to discuss what can be done when such an analysis is performed. Accordingly, the analysis itself is unremarkable. The analyzer operates on a language that lacks functions and exceptional control flow—including the inability to break from a loop. It relies on the syntax to delineate the effects of branching instructions. As a result, this analysis is not suitable for Android applications.

8.4.2 Askarov et al.

Askarov et al. [3] also discuss noninterference. As is the case with Giacobazzi's work, this paper is not primarily about an analyzer. Rather, this paper is about different modifications of noninterference, most especially termination-insensitive noninterference. The paper demonstrates the sorts of information leakages that are possible when attackers can observe divergence. The analyzer and its rules serve as an illustration, as do the example programs.

The language analyzed in this paper serves to illustrate the interference properties presented but is not designed to operate on practical languages. In particular, the analyzer assumes syntactic boundaries on conditional statements and loops and lacks functions and exceptional control flow.

8.4.3 Liu and Milanova

Liu and Milanova [25] perform a static analysis on Java programs that tracks both explicit and implicit information flows. Their analysis performs the static analysis suggested by Denning and Denning [8]; it calculates the postdominators of basic blocks to determine the extent of a conditional statement's effect on control flow. Although they do not address jump statements explicitly, this technique is applicable to languages with jump statements. They go on to discuss and demonstrate applications of this analysis to concurrent execution models. Lastly, they present results of their analysis on a small number of test applications.

Although Liu and Milanova present an analysis technique, they do not present a grammar or prove noninterference. Furthermore, they do not discuss exceptional control flow. They mention interprocedural postdominance but offer no discussion of how postdominance can be calculated when faced with return statements.

8.4.4 Pottier and Simonet

Pottier and Simonet [33] present a type system that guarantees noninterference in an ML-like language. As noninterference relies on multiple program traces, their system allows for certain expressions to bifurcate; they may have one value for one execution and another value for another execution. These variances are stored in brackets. Noninterference relies on a proof that there are no variances in the output;

that is, that there are no brackets in the result of the reduction of a program.

The language analyzed in this work is powerful; it even contains exceptional flow. But its branches are syntactically structured. As such, the work that would need to be done to apply this system to Dalvik bytecode (to provide provable bounds on the effects of control flow from branches) is precisely what this work contributes.

8.4.5 Lourenço and Caires

Lourenço and Caires [26] present a type system that proves noninterference in a language based on the lambda calculus. As such, it is applicable to any language. They do not present any empirical evidence, so it is unclear how long type checking takes or how many safe programs do not type check. An empirical evaluation of this system would be a valuable addition to the literature.

8.5 Dynamic analyses

8.5.1 Moore

Moore et al. [30] present a type system with guarantees about progress-sensitive security, which is called termination-sensitive noninterference by Askarov et al. [3]. Their type system, together with runtime enforcement, guarantees progress-sensitive noninterference. The authors then discuss a modification of their system that allows for a limited amount of information to leak before the runtime enforcement terminates the program. Notably, their type system relies on a termination oracle built on an SMT solver.

The authors extend their work to use lattices to represent security values and then apply the whole system to Jif. They then verify the intraprocedural flows in Civitas (a remote voting system already proven to preserve progress-insensitive noninterference) using their system. They follow with a discussion of the modifications that were required to prove progress-sensitive noninterference. So few modifications were required that the authors conclude that other nonmalicious programs are likely to be straightforward to verify in this manner and suggest that requiring progress-sensitive guarantees is reasonable for security-critical applications.

This analysis sets an important precedent for security research. It does not, however, treat all information flows; interprocedural flows are simply ignored. It also does not guarantee that the program is safe to run except in their runtime environment. As such, the application of this work is limited to systems willing to adopt a runtime that does additional checks.

8.5.2 TaintDroid

TaintDroid [10] is a dynamic extension to Android's runtime environment. Being a dynamic analysis, it does not purport to identify all possible program behaviors. Instead, it monitors behaviors as they happen. Also, it tracks only explicit information flows. The authors mention implicit information flows but do not track them; instead, they suggest that static analyzers could identify them.

The contribution of TaintDroid, as claimed in the paper, is not innovation in taint tracking techniques but in their integration into the Android system. The bulk of the text explains how taints are stored and propagated in TaintDroid. Notably, TaintDroid tracks taints in storage as well as in memory.

8.6 Programmer-assisted analyses

Some of the work in the literature requires programmer cooperation. These systems can be useful for safe development but are typically not useful when the people who want to verify a program are not its developers.

Venkatakrishnan et al. [36] perform a static prepass that adds tracking instructions to inform a dynamic analysis. This analysis preserves termination-insensitive noninterference. The analysis is performed on an imperative language with branches, loops, and function calls. Notably, it lacks exceptional control flow. It relies on syntactic boundaries for the detection of implicit flows.

The static analysis determines which statements can execute along either branch and, by extension, which variables can be updated. Aliasing is irrelevant to this analysis because no pointers exist. Implicit flows due to function calls are impossible as functions are all defined statically. The program transformation is a simple encoding of updates to variables, including a program counter variable. Before some branches and loops, the variables in the condition expression are added to the program counter set. Its old value is preserved so that it can revert after the branch or loop has ended.

Jia et al. [19] present a system that allows programmers to provide annotations that are enforced dynamically. These annotations include security labels and declassification. Enforcement happens as Intents are intercepted and analyzed; individual processes are left completely alone. If the program trace differs from a program trace for a program with no access to secret information, the security property has been violated.

This work is designed to enforce Android application permissions. It makes no attempt to identify information flows, malicious or not, that occur within the scope of permissions allowed. It also makes no attempt to track flows to or from library calls.

Myers [31] created JFlow, an extension to Java that allows programmers to annotate values. Its type system performs some static proving and requires some runtime enforcement of rules. It relies on syntactic bounds on its branches. Furthermore, it permits several species of covert information flows, such as flows using the `hashCode` method of an object and flows using static values.

Heule et al. [18] perform a dynamic analysis on a rich language that includes lambda. Users must install and use a library to make use of this work, however, reducing the communication between components to an MPI-like interface.

Buiras, Vytiniotis, and Russo [4] present a Haskell library that performs a hybrid analysis. Programmers that use this library can guarantee noninterference in Haskell code.

8.7 Relevant specifications

The official specifications for the bytecode language [15] and the dex file format [16] provide detailed information about the syntax and semantics of Dalvik bytecode. The Java Virtual Machine Specification [32] describes the semantics of the JVM.

REFERENCES

[1] Anderson, J.P.: Computer security technology planning study. volume 2. Technical report, DTIC Document (1972)

[2] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 259–269. PLDI '14, ACM, New York, NY, USA (2014)

[3] Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive non-interference leaks more than just a bit. In: Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security, pp. 333–348. ESORICS '08, Springer-Verlag, Berlin, Heidelberg (2008)

[4] Buiras, P., Vytiniotis, D., Russo, A.: HLIO: mixing static and dynamic typing for information-flow control in Haskell. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, pp. 289–301. ICFP 2015, ACM, New York, NY, USA (2015)

[5] Cavallaro, L., Saxena, P., Sekar, R.: On the limits of information flow techniques for malware analysis and containment. In: Zamboni, D. (ed.) Detection of Intrusions and Malware, and Vulnerability Assessment. LNCS, vol. 5137, pp. 143–163. Springer, Heidelberg (2008)

[6] Chang, W., Streiff, B., Lin, C.: Efficient and extensible security enforcement using dynamic data flow analysis. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 39–50. CCS '08, ACM, New York, NY, USA (2008)

[7] Denning, D.E.: A lattice model of secure information flow. Communications of the ACM **19**, 236–243 (1976)

[8] Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM **20**, 504–513 (1977)

[9] Earl, C., Sergey, I., Might, M., Van Horn, D.: Introspective pushdown analysis of higher-order programs. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, pp. 177–188. ICFP '12, ACM, New York, NY, USA (2012)

[10] Enck, W., Gilbert, P., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones pp. 1–6 (2010), http://dl.acm.org/citation.cfm?id=1924943.1924971

[11] Felleisen, M., Friedman, D.P.: A reduction semantics for imperative higher-order languages. In: Proceedings of the Parallel Architectures and Languages Europe, Volume I, pp. 206–223. Springer-Verlag, London, UK, UK (1987), http://dl.acm.org/citation.cfm?id=646425.692409

[12] Giacobazzi, R., Mastroeni, I.: Abstract non-interference: parameterizing non-interference by abstract interpretation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 186–197. POPL '04, ACM, New York, NY, USA (2004)

[13] Gilray, T., Adams, M.D., Might, M.: Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, pp. 407–420. ICFP 2016, ACM, New York, NY, USA (2016)

[14] Gilray, T., Lyde, S., Adams, M.D., Might, M., Van Horn, D.: Pushdown control-flow analysis for free. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 691–704. POPL '16, ACM, New York, NY, USA (2016)

[15] Google: Bytecode for the Dalvik VM. http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html (2014)

[16] Google: Dalvik executable format. http://source.android.com/devices/tech/dalvik/dex-format.html (2014)

[17] Combinator research group, U.: Tapas: Dalvik bytecode analysis in Scala. https://github.com/Ucombinator/Tapas (2014)

[18] Heule, S., Stefan, D., Yang, E.Z., Mitchell, J.C., Russo, A.: Ifc inside: Retrofitting languages with dynamic information flow control. In: International Conference on Principles of Security and Trust, pp. 11–31. Springer, London (2015)

[19] Jia, L., Aljuraidan, J., Fragkaki, E., Bauer, L., Stroucken, M., Fukushima, K., Kiyomoto, S., Miyake, Y.: Run-time enforcement of information-flow properties on android. In: Crampton, J., Jajodia, S., Mayes, K. (eds.) Computer Security – ESORICS 2013. LNCS, vol. 8134, pp. 775–792. Springer, Heidelberg (2013)

[20] Kang, M.G., McCamant, S., Poosankam, P., Song, D.: DTA++: Dynamic taint analysis with targeted control-flow propagation. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2011. The Internet Society (2011)

[21] Kim, J., Yoon, Y., Yi, K., Shin, J.: Scandal: Static analyzer for detecting privacy leaks in android applications. Mobile Security Technologies (2012)

[22] Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems (TOPLAS) **1**, 121–141 (1979)

[23] Liang, S., Keep, A.W., Might, M., Lyde, S., Gilray, T., Aldous, P., Van Horn, D.: Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In: Proceedings of the Third ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, pp. 21–32. SPSM '13, ACM, New York, NY, USA (2013)

[24] Liang, S., Might, M.: Hash-flow taint analysis of higher-order programs. In: Proceedings of the 7th Workshop on Programming Languages and Analysis for Security, pp. 8:1–8:12. PLAS '12, ACM, New York, NY, USA (2012)

[25] Liu, Y., Milanova, A.: Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In: 14th European Conference on Software Maintenance and Reengineering (CSMR), pp. 146–155. Washington, DC, USA (2010)

[26] Lourenço, L., Caires, L.: Dependent information flow types. In: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 317–328. POPL '15, ACM, New York, NY, USA (2015)

[27] Lyde, S., Might, M.: State exploration choices in a small-step abstract interpreter. In: Scheme and Functional Programming Workshop 2015. SFP 2015, Vancouver, Canada (2015)

[28] Might, M., Manolios, P.: A posteriori soundness for non-deterministic abstract interpretations. In: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation, pp. 260–274. VMCAI '09, Springer-Verlag, Heidelberg (2009)

[29] Might, M., Shivers, O.: Improving flow analyses via ΓCFA: abstract garbage collection and counting. In: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming, pp. 13–25. ICFP '06, ACM, New York, NY, USA (2006)

[30] Moore, S., Askarov, A., Chong, S.: Precise enforcement of progress-sensitive security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, pp. 881–893. CCS '12, ACM, New York, NY, USA (2012)

[31] Myers, A.C.: JFlow: practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 228–241. POPL '99, ACM, New York, NY, USA (1999)

[32] Oracle: Java virtual machine specification. https://docs.oracle.com/javase/specs/jvms/se7/html/index.html (2013)

[33] Pottier, F., Simonet, V.: Information flow inference for ML. ACM Transactions on Programming Languages and Systems (TOPLAS) **25**, 117–158 (2003)

[34] Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Selected Areas in Communications **21**, 5–19 (2006)

[35] Van Horn, D., Might, M.: Abstracting abstract machines. In: Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, pp. 51–62. ICFP '10, ACM, New York, NY, USA (2010)

[36] Venkatakrishnan, V.N., Xu, W., DuVarney, D.C., Sekar, R.: Provably correct runtime enforcement of non-interference properties. In: Ning, P., Qing, S., Li, N. (eds.) Information and Communications Security. LNCS, vol. 4307, pp. 332–351. Springer, Heidelberg (2006)

[37] Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Computer Security **4**, 167–187 (1996), http://dl.acm.org/citation.cfm?id=353629.353648

[38] Volpano, D., Smith, G.: Eliminating covert flows with minimum typings. In: Computer Security Foundations Workshop, 1997. Proceedings., 10th, pp. 156–168. Rockport, Massachusetts, USA (1997)

[39] Xu, W., Bhatkar, S., Sekar, R.: Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In: Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15. USENIX-SS '06, USENIX Association, Berkeley, CA, USA (2006), http://dl.acm.org/citation.cfm?id=1267336.1267345

[40] Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. In: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering, pp. 97–106. SIGSOFT '04/FSE-12, ACM, New York, NY, USA (2004)