Noninterference with Precise Interprocedural Control Flow

Peter Aldous
and Eric Mercer
Department of Computer Science
Brigham Young University
Provo, Utah 84602–6576
Email: aldous,egm@cs.byu.edu

Matthew Might
Hugh Kaul Precision Medicine Institute
University of Alabama at Birmingham
Birmingham, Alabama 35233–1806
Email: might@uab.edu

***Abstract*—A malicious information flow occurs when a program accesses information and relays it without a user's knowledge or intent. Some users, such as journalists, diplomats, and military personnel, have a particular need for privacy because of the sensitivity of the data they handle. In their use case, there is a need for a high degree of privacy assurance that justifies heavyweight program analyses. Although many information flow analyses exist, few of them prove noninterference, or the absence of information flows. Classical information flow analyses use the control-flow graph to reason about control flow; as a result, they are imprecise in the presence of interprocedural control flows. One existing analysis, herein called the augmented-state analysis, is capable of proving noninterference precisely in the presence of interprocedural control flow. This paper shows that the augmented-state analysis has exponential asymptotic time complexity in the size of the program, making it intractable. It presents a new *a posteriori* analysis with polynomial complexity that tracks the same information flows as does the augmented-state analysis and, as a result, provides the same proof of noninterference. This paper further describes a reference implementation of both analyses that operates directly on Android applications. It presents an empirical study over a set of 119 small Android benchmarks and a case study of twelve larger Android applications. All 131 Android programs in the study were designed to test security analyses. The study confirms the augmented-state analysis to be intractable and shows the *a posteriori* analysis to be tractable and precise for Android applications of moderate size.**

I. Introduction

Although privacy is a concern for everyone, there are groups of people for whom it is especially important. Diplomats, journalists, and military personnel, for example, have a greater need for privacy and are consequently more likely to invest time and computational resources for stronger assurances of privacy. Many of these people routinely use smartphones with cameras, microphones, and GPS receivers, which could potentially be used to gain access to their sensitive data. Malicious information flows are especially difficult to identify in programs that have a legitimate need to access both sensitive information (colloquially, **sources**) and communications channels (**sinks**).

In an attempt to address this problem, a recent DARPA project [3] envisioned a curated, verified app store. One property they hoped to verify is the absence of malicious information flows. Recognizing that fully automated verification may not be possible, this project used tools to focus the efforts of analysts on a reduced set of potentially malicious flows to manually verify. For this use case, the inability to prove that all flows are safe is acceptable if the analysis significantly reduces the number of flows that must be manually inspected. The absence of information flows is demonstrated with **noninterference**, which is the property that one datum cannot interfere with (or influence) another.

Many works that prove noninterference operate on languages lacking crucial features, such as methods or exceptional flow [6], [8], [15], [22], [42], [51], [52]. Although they provide a necessary foundation, these analyses cannot be applied directly to programs in languages that contain these features. Some few works [2], [9], [25], [39], [44] do prove noninterference in languages with methods and exceptions. Still fewer works [25], [38] present an implementation of their analysis (further publications on these analyses include [12], [13], [24], [26], [27], [47]). These analyses depend on classical techniques, such as control flow graphs, to reason about information flows. As a result, they are unable to reason precisely about interprocedural control flows. This paper evaluates and improves upon the analysis of Aldous and Might [2], which is able to reason about interprocedural control flows and which is written for a summary of Dalvik bytecode. (Although the DVM has been replaced by ART, Android devices download programs in Dalvik bytecode and then compile them.) Because their analysis adds components to the states in an abstract interpreter, it is referred to throughout this paper as the **augmented-state** analysis.

This paper presents a complexity analysis of the augmented-state analysis, which shows that it is exponential in the size of the program because of its **implicit taint values**. Implicit taint values allow this analysis to track information flows and then retrospectively remove taints that are proven infeasible. This paper restructures the augmented-state analysis into two phases, abstract interpretation and taint propagation, obviating the need for implicit taint values. The result is a polynomial analysis. To distinguish it from the augmented-state analysis, the restructured analysis is called the ***a posteriori*** analysis because it propagates taints only after abstract interpretation

```
1  if (source) {
2    sink = true;
3  } else {
4    sink = false;
5  }
6  safe = "safe";
```

Fig. 1: An implicit information leak

has concluded.

This paper also presents an empirical evaluation of both analyses based on implementations that target full Dalvik bytecode. Both implementations require a specification of the sources and sinks, as do all information flow analyses, but require no other annotations. The evaluation shows that the new *a posteriori* analysis, unlike the augmented-state analysis, is tractable over a benchmark set of malicious apps from DARPA. It also measures the precision and speed of the *a posteriori* analysis on a set of Android information flow micro-benchmarks [4].

In summary, the contributions of this work are:

- a novel *a posteriori* analysis that tracks information flows in Dalvik bytecode;
- complexity analyses that show the asymptotic complexity of the augmented-state analysis to be exponential and the *a posteriori* analysis to be polynomial;
- a proof that the *a posteriori* analysis tracks the same information flows as the augmented-state analysis (which, in turn, proves noninterference);
- an implementation of both analyses for Android applications; and
- an empirical evaluation of both analyses, which show that the augmented-state analysis is intractable but that the *a posteriori* analysis is tractable for programs of moderate size.

Section II elaborates on the background for this work. Section III describes the *a posteriori* analysis and Section IV demonstrates a simple example of it. Section V is an empirical evaluation. Section VI summarizes relevant related work. Section VII concludes.

II. Background

The standard method for tracking information flows is to apply a security type or label to values that come from a source. Colloquially, these types or labels are called **taints**. In order to prove noninterference, it is necessary to identify situations where values flow from address to address (**explicit information flows**) as well as situations where values flow via control flow (**implicit information flows**).

Figure 1 illustrates a simple implicit information flow from `source` to `sink`. `sink` copies the constant true in the true branch and the constant false in the false branch. By exploiting control flow, `sink` duplicates `source` while evading detection by explicit-only data flow analyses.

```
void main(String []a) {
  source = a[0];
  intermediate();
  sink = true;
}
void intermediate() {
  try {
    parse();
  } catch (Exception e) {}
}
void parse() {
  Integer.parseInt(source);
}
```

Fig. 2: An escaping exception

Denning and Denning [19] showed that it is possible to track implicit information flows by applying taints to **context**, or program location, in addition to variables or addresses. By using these **context taints**, it is possible to track implicit flows together with explicit flows. To address unstructured control flow, such as goto statements, Denning and Denning further assert that the effect of a branch ends at the immediate postdominator of the branch's node in the control-flow graph. Volpano et al. [51] later validated this claim.

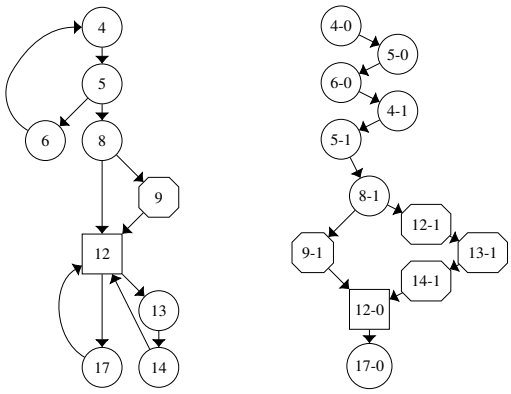
However, their claims only apply in an intraprocedural context; in other words, this technique is unable to reason about interprocedural control flows. This is more than a minor drawback, as interprocedural control flow is frequently relevant to questions of information flow. Consider, for example, the program in Figure 2. As do many parsers, this one throws an exception that can escape its calling function. An analysis that cannot reason about interprocedural control flows observes that control flow may reach differing exit nodes from `parse()` and so concludes that everything that occurs downstream in the program is unsafe. As a result, these analyses cannot prove that `sink` is unaffected by `source` (and, in fact, JOANA reports an information flow when analyzing this program).

The obvious solution to this problem is to use an interprocedural control-flow graph. However, Aldous and Might [2] show that an interprocedural control-flow graph does not suffice to prove noninterference. However, this leads to an analysis that can miss information flows. The program in Figure 3 uses the stack to leak information about `source` to the terminal when it is invoked as `printSource(0)`. The program recurs immediately as `printSource(1)`. Regardless of the value of `source`, execution reaches line 12. However, the value of `frame` reflects the value of `source`.

The interprocedural control-flow graph in Figure 4a illustrates this behavior. Since line 12 is the immediate postdominator of the branch at line 8, paths through the control-flow graph from line 8 merge at line 12. It is shown as a rectangle. Line 9 is shown as an octagon because it is in the **influence** of line 8; that is, between line 8 and line 12 in the CFG. Accordingly, a taint analysis reports that no taint propagates

```
 1  private boolean source;
 2  private boolean sink = true;
 3
 4  void printSource(int frame) {
 5    if (frame == 0) {
 6      printSource(1);
 7    } else {
 8      if (source) {
 9        return;
10      }
11    }
12    if (frame == 1) {
13      sink = false;
14      return;
15    }
16
17    System.out.println(sink);
18  }
```

Fig. 3: An information leak via the stack



(a) The interprocedural CFG (b) The execution point graph

Fig. 4: Representations of control flow in Figure 3

to the call to `println`.

Aldous and Might propose an **execution point graph** as a more precise definition of context. The execution point graph couples control flow information with stack depth to reason more precisely about program context. Nodes in the execution point graph are called **execution points** and contain a program location and a numeral indicating the height of the stack when execution reaches this location. Aldous and Might show that the effect of a branch ends at the immediate postdominator of its execution point.

Their augmented-state analysis uses the execution point graph to identify and remove information flows that are infeasible. The execution point graph for the program in Figure 3 is shown in Figure 4b. Although 17-0 (the execution point at line 17 and stack depth 0) occurs outside of the context of the branch at 8-1, it reads `sink`, which is written inside of the influence of 8-1. Since control flow at 8-1 depends on `source`, the taint on `source` is applied to context at 8-1 and from there to `sink` at 13-1.

Aldous and Might prove termination-insensitive noninterference [6] (often called progress-insensitive noninterference in more recent works [7], [42]) with their analysis. The proof of noninterference is based on a property they call **similarity**, which is a generalization of **low-equivalence** that includes context. Low-equivalence is the property that two stores, or models of memory, are equivalent on their low-security, or observable, values. Two stores are similar if they are equivalent excepting addresses marked with a taint. Similarity extends low-equivalence to contexts by requiring two states to have the same execution point unless there is a context taint. If all observable behaviors, such as network traffic, comes from low-equivalent sources and in low-equivalent contexts, progress-insensitive noninterference is preserved.

Their analysis is progress-insensitive because it relies on postdominance to identify and remove infeasible taints. Because postdominance only treats paths that reach an exit node, it ignores infinite paths that never reach a postdominator. In addition, their implementation ignores top-level exceptions in the postdominance calculation. Their claim can be strengthened to progress-sensitive noninterference if it can be proven that control flow reaches relevant postdominators. This can be accomplished using termination analyses and by proving the infeasibility of the top-level exceptions.

III. *A posteriori* TAINT TRACKING

The AAM [48] framework for abstract interpretation is convenient for producing the data structures upon which taint propagation depends. AAM abstract interpreters use a Galois connection over states based on the CESK framework [21], which directly executes formal semantics. Additional details of the abstract interpretation are included in the technical report [1].

Abstract interpretation is used to create four data structures: the execution point graph, a map Δ that records which addresses are read and written at each program location, a map β that records the addresses that affect control flow, and a map ψ that stores execution points passed as interpretation moves up the stack looking for an exception handler. The second phase uses these data structures to perform an information flow analysis.

A. Data structures from abstract interpretation

1) EPG: the execution point graph: The execution point graph is created from the abstract state space. Execution points are pairs: $\hat{e} \in \hat{E} = C \times (\mathbb{N} + ?)$. The second member of each pair represents the height of the stack, which may not be computable in an abstract interpreter. Whenever an exact stack height cannot be computed, the abstract stack height is ?. Execution points may only act as postdominators if they have an exact stack height. Stack height is computed with $\widehat{SH}$, which is formally defined in Appendix B.

2) Δ: the writes map: The writes map Δ records the addresses read and written at each code point. It is constructed in concert with abstract interpretation for convenience and for

generality in the presence of some of the more esoteric features of AAM abstract interpretation [40].

$\Delta : C \to \widehat{Addr} \to \mathcal{P}\left(\widehat{Addr}\right)$ contains a map from addresses written to addresses read at each code point. The intuitive structure for the writes map Δ would reverse addresses read and addresses written. However, the given structure allows for convenient propagation of both explicit and implicit information flows.

Consider a code point c_c whose instruction is `Const(`r_c`,` c`)`. r_c is written but no values influence the write. (For simplicity, this section uses Dalvik virtual registers as if they were addresses.) Accordingly, $\Delta(c_c) = [r_c \mapsto \emptyset]$. This structure indicates that r_c is written at c_c, which enables the propagation of context taints to r_c.

3) β: the branch map: The next data structure, the branch map or β, shows which context taints should be created and where. It is a map from execution points to sets of addresses: $\beta : \hat{E} \to \mathcal{P}\left(\widehat{Addr}\right)$. β is empty at an execution point that does not branch.

In the program in Figure 1, β is empty for each line except for 1, where a branch occurs. $\beta((1,h)) = \{\texttt{source}\}$ for some stack height h.

β is also used for subtler branches, such as the exception that may or may not be thrown at an `IGet` instruction because the receiver may or may not be null. Because values merge in the abstract store, multiple objects may coexist and each one may affect control flow. Consider an instruction `IGet(`r_d`,` r_o`,` *field*`)`. In this case, $\beta(\hat{e}_\varsigma)$ includes r_o and every object address (every member of $\widehat{OA}$) in the abstract store at r_o: $\beta(\hat{e}_\varsigma) = \{r_o\} \cup \left(\hat{\sigma}(r_o) \cap \widehat{OA}\right)$.

4) ψ: the unwinding map: ψ, the final data structure required for taint propagation, is used to track a subtle species of context taint. When an exception is thrown, the interpreter unwinds the stack until it finds a suitable exception handler. As it unwinds, it effectively visits the call sites in the stack and the analysis must accumulate context taints from them. The program snippet in Figure 5 demonstrates a situation where ψ is necessary.

This program begins at `main()`, which calls `f()`, branches, and then calls `g()` in both branches. Both executions of `g()` occur at the same stack depth. As such, there should be no context taint at 24-2. Returning from `g()` results in the context taint being reapplied. However, an exception that escapes `g()` also escapes the `true` branch of `f()` and arrives at the exception handler in `main()`, which is reachable in no other way. The result is that `sink` duplicates the value of `source` and prints it to the terminal. ψ enables the *a posteriori* analysis to detect such a flow.

In *a posteriori* taint tracking, the unwinding map $\psi : \hat{E} \to \hat{E} \to \mathcal{P}\left(\hat{E}\right)$ is used to aggregate context taints as the stack unwinds. $\psi(\hat{e}_\varsigma)(\hat{e}_d) = \hat{E}_\psi$ means that when an exception is thrown at $\hat{e}_\varsigma$ that reaches $\hat{e}_d$, the stack unwinds past the execution points in $\hat{E}_\psi$.

```
 1  private boolean source;
 2  void main() {
 3    boolean sink = false;
 4    try {
 5      f();
 6    } catch(Exception e) {
 7      sink = true;
 8    }
 9    System.out.println(sink);
10  }
11  void f() {
12    if (source) {
13      g();
14    } else {
15      try {
16        g();
17      } catch(Exception e) {}
18    }
19  }
20  void g() {
21    throw new Exception();
22  }
```

Fig. 5: Taint via stack unwinding

B. Definitions

Taint values may be defined according to the needs of the analysis; for present purposes, they identify the execution point at which a source is accessed: $\widehat{\Theta} \subseteq \hat{E}$. A taint store is a map from addresses to taint values: $\widehat{\mathcal{T}} = \widehat{Addr} \to \widehat{\Theta}$. An empty taint store maps every address to the empty set.

A single global taint store is shared by all states, although other configurations are equally correct. The taint store $\widehat{\tau}$ is updated weakly at an address $\hat{a}$ with taint values $\widehat{\Theta}_u$ as follows:

$$\widehat{\tau}\left[\hat{a} \overset{\sqcup}{\mapsto} \widehat{\Theta}_u\right] \equiv \widehat{\tau}\left[\hat{a} \mapsto \widehat{\tau}(\hat{a}) \cup \widehat{\Theta}_u\right] .$$

The updated taint store merges with an existing taint store using $\sqcup : (\widehat{\tau}_1 \sqcup \widehat{\tau}_2)(\hat{a}) = \widehat{\tau}_1(\hat{a}) \cup \widehat{\tau}_2(\hat{a})$.

Context taint values $\widehat{\omega} \in \widehat{\Omega} : \hat{E} \to \widehat{\Theta}$ are maps from branching execution points to the taint values on values that affect control flow at these locations. $\Xi : \hat{E} \to \widehat{\Omega}$ is a widened context taint map that contains the context taint values at each execution point. $\Xi(\hat{e}_c)(\hat{e}_b) = \widehat{\Theta}_e$ means that at $\hat{e}_c$ there is a context taint from a branch at $\hat{e}_b$, where the branch at $\hat{e}_b$ depended on values marked with $\widehat{\Theta}_e$.

Ξ and context taint values are implicitly empty whenever they are undefined. That is, when there is no context taint at $\hat{e}$, $\Xi(\hat{e}) = []$ and if $\widehat{\omega}$ is undefined at $\hat{e}_b$ (for example, if it is the empty map $[]$ in the previous case), $\widehat{\omega}(\hat{e}_b) = \emptyset$. With this implicit emptiness, $\sqcup$ is even simpler to define for context taint values:

$$(\widehat{\omega}_1 \sqcup \widehat{\omega}_2)(\hat{e}) \equiv \widehat{\omega}_1(\hat{e}) \cup \widehat{\omega}_2(\hat{e}) .$$

Every update to Ξ is weak and conditional. As is the case with taint stores, weak update merges with existing values instead of replacing them. Updates are conditional because context taints do not propagate beyond the influence of their branch. The influence of an execution point $\hat{e}_b$, denoted $\iota(\hat{e}_b)$, is the set of all execution points along some path from $\hat{e}_b$ to its immediate postdominator (where the postdominator appears only at the end of the path). When $\hat{e}_c$ is outside the influence of $\hat{e}_b$, control flow does not depend on the branch at $\hat{e}_b$. In this case, Ξ is not updated. This is analogous to the removal of invalid implicit taints in the augmented-state analysis. The set of execution points with non-empty context taint that contain an execution point $\hat{e}_c$ in their influence is $\hat{E}_b(\hat{e}_c) \equiv \{\hat{e}_b \mid \widehat{\omega}(\hat{e}_b) \neq \emptyset \wedge \hat{e}_c \in \iota(\hat{e}_b)\}$. Weak, conditional updates are presented as $\Xi(\hat{e}_c) \stackrel{?}{:=} \widehat{\omega}$:

$$\forall \hat{e}_u \in \hat{E}_b(\hat{e}_c) : \Xi(\hat{e}_c)(\hat{e}_u) := \Xi(\hat{e}_c)(\hat{e}_u) \cup \widehat{\omega}(\hat{e}_u) \ .$$

The following four functions look up the data necessary for taint propagation. $\widehat{\Theta}_\Delta$ and $\widehat{\Theta}_\Xi$ are used to update the taint store, respectively reading from Δ and Ξ. $\widehat{\Theta}_\beta$ and $\widehat{\Theta}_\psi$ are used to update the context taint map, respectively using β and ψ.

$\widehat{\Theta}_\Delta(\hat{a}, c)$ is used to track explicit flows. It indicates the aggregation of taints from $\Delta(c)(\hat{a})$.

$\widehat{\Theta}_\Xi(\hat{e})$ is used to track implicit flows. It contains the set of context taints at some execution point $\hat{e}$. It collects all taint values associated with any execution point $\hat{e}_b$. No context taint is defined for execution points that do not branch or whose influence does not include $\hat{e}$. In these cases, Ξ correctly reports an empty set of taints.

$\widehat{\Theta}_\beta(\hat{e})$ is the aggregation of taints at addresses that affect control flow at $\hat{e}$.

Recall from Section III-A that when an exception is thrown, the stack unwinds until a suitable exception handler is found. As this search unfolds, it traverses some set of execution points. For each successor $\hat{e}'$, the taints in Ξ at each of these execution points is $\widehat{\Theta}_\psi(\hat{e}_\varsigma, \hat{e}')$. $\hat{e}_\mathcal{T}$ denotes an execution point traversed during this process between $\hat{e}_\varsigma$ and $\hat{e}'$. $\widehat{\Theta}_\psi(\hat{e}_\varsigma, \hat{e}')$ is the aggregation of all context taints at each of these execution points.

$$\begin{aligned}
\widehat{\Theta}_\Delta(\hat{a}, c) &\equiv \bigcup_{\hat{a}_s \in \Delta(c)(\hat{a})} \widehat{\tau}(\hat{a}_s) \\
\widehat{\Theta}_\Xi(\hat{e}) &\equiv \bigcup_{\hat{e}_b \in \hat{E}} \Xi(\hat{e})(\hat{e}_b) \\
\widehat{\Theta}_\beta(\hat{e}) &\equiv \bigcup_{\hat{a} \in \beta(\hat{e})} \widehat{\tau}(\hat{a}) \\
\widehat{\Theta}_\psi(\hat{e}_\varsigma, \hat{e}') &\equiv \bigcup_{\hat{e}_\mathcal{T} \in \psi(\hat{e}_\varsigma)(\hat{e}')} \widehat{\Theta}_\Xi(\hat{e}_\mathcal{T}) \ .
\end{aligned}$$

C. Taint propagation

With $\widehat{\Theta}_\Delta$, $\widehat{\Theta}_\Xi$, $\widehat{\Theta}_\beta$, and $\widehat{\Theta}_\psi$, taint propagation is extremely simple. It iterates over the execution points in EPG until reaching a fixed point. At each execution point, it begins by updating the abstract taint store with taints propagated explicitly ($\widehat{\Theta}_\Delta$) and with taints propagated implicitly ($\widehat{\Theta}_\Xi(\hat{e}_\varsigma)$).

Then, new context taints are propagated to each successor of the execution point. First, the context taint value $\Xi(\hat{e}_\varsigma)$ is retrieved. Then, new context taints are added for any relevant taints in from branching ($\widehat{\Theta}_\beta$) and unwinding the stack ($\widehat{\Theta}_\psi$). Finally, $\stackrel{?}{:=}$ drops all of the context taints whose branches' influence does not contain $\hat{e}'$ before updating $\Xi(\hat{e}')$. d computes the domain of some partial function.

$$\begin{aligned}
&\forall \hat{e}_\varsigma = (c, h) \in d(EPG) : \\
&\quad \forall \hat{a} \in d(\Delta(c)) : \widehat{\tau} := \widehat{\tau}\left[\hat{a} \overset{\sqcup}{\mapsto} \widehat{\Theta}_\Delta(\hat{a}, c) \cup \widehat{\Theta}_\Xi(\hat{e}_\varsigma)\right] ; \\
&\quad \forall \hat{e}' \in EPG(\hat{e}_\varsigma) : \\
&\qquad \Xi(\hat{e}') \stackrel{?}{:=} \Xi(\hat{e}_\varsigma)\left[\hat{e}_\varsigma \overset{\sqcup}{\mapsto} \left(\widehat{\Theta}_\beta(\hat{e}_\varsigma) \cup \widehat{\Theta}_\psi(\hat{e}_\varsigma, \hat{e}')\right)\right] .
\end{aligned}$$

Note that if there is no branch or exception at $\hat{e}_\varsigma$, the update to $\Xi(\hat{e}_\varsigma)$ is empty. The whole process repeats until no changes are made to the taint store or to the context taint map. Taint propagation needs no data besides EPG, Δ, β, and ψ. None of these data structures requires semantics to be used. As a result, the target language is decoupled from the taint analysis.

D. Complexity

Although the *a posteriori* analysis occurs in two phases and appears substantially different from the augmented-state analysis, the two analyses actually perform the same work. As such, the calculation of the time complexity of each analysis is the same—except for their respective definitions of taint values. The augmented-state analysis requires implicit taint values, which are obviated by the availability of the execution point graph in the *a posteriori* analysis. In both cases, the bound on complexity is calculated as the height of a lattice, as each analysis ascends its respective lattice monotonically.

The bounds on taint value size ($|\widehat{\Theta}|$) and on the height of the system lattice ($\log|Sys|$) in the augmented-state analysis are exponential:

$$\begin{aligned}
|\widehat{\Theta}| &= |\hat{E}| + |\hat{E}| \cdot 2^{|\hat{E}|^2} = O\left(2^{|C|^6}\right) \text{ and} \\
\log|Sys| &= O\left(2^{|C|^6}\right) .
\end{aligned}$$

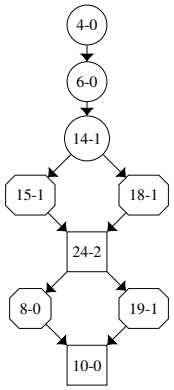
In contrast, abstract taint values and the height of the system lattice in the *a posteriori* analysis are polynomial. Abstract interpretation dominates the complexity of this analysis:

$$|\widehat{\Theta}| = |\hat{E}| = O\left(|C|^3\right) \text{ and } \log|Sys| = O\left(|C|^6\right) .$$

A complete demonstration of the two analyses' complexity is included in Appendix C.

E. Equivalence to augmented-state analysis

The proof that the *a posteriori* analysis tracks the same flows as does the augmented-state analysis uses an intermediate analysis, the **valid-only** analysis, that performs the augmented-state analysis as if the execution point graph were already available. More specifically, the proof demonstrates



(a) The execution point graph (EPG)

c	address written	address read
4	`sink`	$\emptyset$
8	`sink`	$\emptyset$

(b) Writes map (Δ)

$\hat{e}$	addresses
14-1	{`source`}
24-2	$\{\widehat{oa}\}$

(c) Branches map (β)

origin	destination	passed
24-2	8-0	{15-1}
24-2	19-1	$\emptyset$

(d) Unwinding map (ψ)

Fig. 6: Taint propagation data structures for Figure 5

that for every execution of the *a posteriori* analysis, there is an analogous augmented-state analysis that tracks exactly the same information flows (and vice versa). The proof of equivalence is included in Appendix E.

IV. An example

Taint propagation on the program in Figure 5 uses all four of the data structures defined in Section III-C. This section demonstrates taint propagation with this program as a concrete example. For simplicity, this section uses Java variable names as addresses. Empty data, such as a row where $c = 6$ in Δ, are elided. A single pass over the execution point graph is sufficient for this case.

Figure 6 contains the taint propagation data for the program in Figure 5. Figure 6a shows the execution point graph, Table 6b shows the writes map, Table 6c shows the branches map, and Table 6d shows the unwinding map.

The single pass over the execution point graph for taint propagation is shown in Table I. At each step, only new values are shown. As an example, the first row in the table shows the behavior at 4-0: the program begins and `source` has a taint value $\widehat{\theta}$ assigned to it in the taint store.

V. Empirical evaluation

The empirical evaluation comprises two experiments. The first is an evaluation of the analyses on DroidBench [4], a set of 119 information flow microbenchmarks for Android. DroidBench programs primarily exist to test the feature completeness and precision of Android analyses.

The second experiment is a case study on twelve programs from the Automated Program Analysis for Cybersecurity program [3] (APAC), a project designed to create a curated app store of secure programs. The applications from the APAC project were developed to confuse analysis tools.

Although DroidBench benchmarks clearly state what information flows are possible, APAC has no such definition. Furthermore, each of them uses several sources and sinks that are not part of any intended flows. As a result, their precision and correctness are best measured against manually determined ground truth. The DroidBench applications are small enough to have exhaustive definitions of ground truth, but all of the APAC programs except Filterize have an intractably large number of sources and sinks. Accordingly, the measured ground truth for the other APAC programs is based on a random sampling of sources and sinks. When an information flow could happen but relied on the Android library, it was marked out of bounds and omitted from the results.

All applications used the same configuration, including the same abstract domains. The analysis does not validate library code and does not perform termination analysis. In the spirit of progress-sensitive noninterference, postdominance ignores top-level exceptions. The implementation assumes that the bytecode it analyzes is well formed; specifically, that no implicit type coercions occur. Additional details about the configurations and experiments are included in Appendix D.

A. Results

26 of the DroidBench programs used features not yet added to the implementations. Most of these features require only a small engineering effort to add. The *a posteriori* analysis correctly identified all realizable information flows in the remaining 93 programs. 24 of the 119 programs exhibited false positives. Most false positives were due to a lack of object sensitivity, which can be changed simply by changing the analysis configuration.

The augmented-state analysis has no results, as it timed out or ran out of memory in each trial. Table II shows the state space and time metrics of the *a posteriori* analysis for the APAC case study. It lists the number of reachable states and instructions in the program, as well as the time spent in the abstract interpretation and information flow phases and the total analysis time. Four programs timed out during abstract interpretation and are omitted. Time measurements are an average of four trials.

Table III shows the measurements of precision. The first three columns show ground truth, as measured by hand: P (positive) shows the number of realizable flows, N (negative) shows the number of unrealizable flows, and Total shows the number of flows measured. The next four columns show

$\hat{e}_\varsigma$	$\widehat{\tau}$	Ξ	$\widehat{\Theta}_\Xi(\hat{e}_\varsigma)$
4-0	$\left[\texttt{source} \mapsto \left\{\widehat{\theta}\right\}\right]$	–	–
6-0	–	–	–
14-1	–	$\left[\text{15-1} \mapsto \left[\text{14-1} \mapsto \left\{\widehat{\theta}\right\}\right], \text{18-1} \mapsto \left[\text{14-1} \mapsto \left\{\widehat{\theta}\right\}\right]\right]$	–
15-1	–	–	$\left\{\widehat{\theta}\right\}$
24-2	–	$\left[\text{8-0} \mapsto \left[\text{24-2} \mapsto \left\{\widehat{\theta}\right\}\right]\right]$	–
8-0	$\left[\texttt{sink} \mapsto \left\{\widehat{\theta}\right\}\right]$	–	$\left\{\widehat{\theta}\right\}$
10-0	$\left[\texttt{STDOUT} \mapsto \left\{\widehat{\theta}\right\}\right]$	–	–
19-1	–	–	–
18-1	–	–	–

TABLE I: Taint propagation for Figure 5

Application	States	Instructions	AI time (s)	IF time (s)	Total time (s)
BattleStat	3951	2117	61.2	414.4	491.3
ConferenceMaster	8940	3159	1552.7	267.2	1835.4
Filterize	3405	1460	19.9	7.9	44.9
keymaster	13881	4594	49141.8	2414.4	51572.8
PassCheck	10865	4911	454.2	144.7	614.1
pocketsecretary	39069	5123	50499.3	2106.0	52626.0
rLurker	1105	915	13.9	3.5	33.5
Valet	1808	1462	28.3	11.9	59.9

TABLE II: State space and time measurements

the precision of the analysis. For example, a false positive (FP) is an unrealizable flow the analyzer failed to prove safe and a true negative (TN) is an unrealizable flow proven safe by the analysis. The table also shows false positives and true negatives as a percentage of the total number of flows measured.

B. Analysis

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

The asymptotic complexity of the abstract interpretation phase provides the dominating term in the overall complexity of the *a posteriori* analysis. This is corroborated by the data, which show that abstract interpretation dominates analysis time for all of the APAC apps except BattleStat. This includes the applications that timed out; every application that timed out did so during the abstract interpretation phase.

The precision of the analysis can be measured in terms of false positives. However, this tool's purpose is to disprove the possibility of information flows. Accordingly, true negatives are a more appropriate measure of the efficacy of this tool. True negatives range between 64% and 92% of the measured flows in the APAC applications. In the case of Filterize, which is measured exhaustively, 88% of the possible information flows can be automatically proved unrealizable. In every one of these cases, the analysis removes a majority of the proof burden on a human analysis.

VI. Related work

Sabelfeld and Myers [45] summarize the literature on information flows done as of 2003. Taint tracking has its origins as far back as the work of Denning [18], which introduces the idea of taint values as lattices instead of booleans. Denning and Denning [19] describe a static analysis that guarantees noninterference (although they do not use the term) on a simple imperative language. They do not, however, consider function calls or exceptional flow. Volpano et al. [51] validate the claims of Denning and Denning. Volpano and Smith [52] then extend it to handle termination leaks and some exceptional flow leaks. Their language lacks function calls and jumps, which greatly simplifies the complexity of exceptional flow.

Injection into Android programs is nontrivial because they are event-driven. Liang et al. [35] introduce entry-point saturation, which repeatedly injects into each entry point and accumulates values in a weak store.

Many related works address explicit information flows. Chang et al. [16] present a compiler-level tool that tracks explicit flows. Kim et al. [33] perform an abstract interpretation on Android programs and track explicit flows. Arzt et al. [5] present FlowDroid, a static analyzer for Android applications. They model application lifecycles but do not address implicit information flows. Huang et al. [30] present DFlow, a type-based system that tracks some information flows. Xu et al. [53], Kang et al. [32], and Liang and Might [36] each perform information flow analyses on various languages. None of these works proves noninterference.

Liu and Milanova [37] perform a static analysis on Java programs that tracks both explicit and implicit information flows but do not present a grammar, prove noninterference,

Application	P	N	Total	TP	FP	TN	FN	FP%	TN%
BattleStat	1	24	25	1	1	23	0	4%	92%
ConferenceMaster	0	25	25	0	3	22	0	12%	88%
Filterize	4	296	300	4	33	263	0	11%	88%
keymaster	1	24	25	1	8	16	0	32%	64%
PassCheck	0	25	25	0	2	23	0	8%	92%
pocketsecretary	0	25	25	0	4	21	0	16%	84%
rLurker	0	25	25	0	2	23	0	8%	92%
Valet	1	24	25	1	1	23	0	4%	92%

TABLE III: Precision metrics: true positives (TP), false positives (FP), etc.

or discuss exceptional control flow. Giacobazzi and Mastroeni [22], Askarov et al. [6], Cabon and Schmitt [15], Assaf et al. [8], and Kobayashi and Shirane [34] all prove noninterference in simple imperative languages. In each case, the language analyzed lacks language features necessary for direct application to Dalvik bytecode.

Pottier and Simonet [44] present a type system that guarantees noninterference in an ML-like language. The language analyzed in this work is powerful; it even contains exceptional flow. Lourenço and Caires [39] present a type system that proves noninterference in a language based on the lambda calculus. In both of these cases, it is possible to rewrite Dalvik bytecode in terms of their language, although it is not clear how such a rewrite would affect precision or program size. It is also unclear how much type annotation is required to type check a program. Neither paper presents empirical results.

Barthe et al. [10] prove noninterference in a type system in a summarized bytecode language. Their language includes functions, objects, and exceptional control flow. They present no implementation or empirical evaluation of their analysis. The JOANA information flow framework [12], [13], [24]–[27], [47] performs information flow analysis on Java programs, as discussed in Section II. While JOANA is mature, its Android variant JoDroid [41] is still under development. Lortz et al. [38] present Cassandra. All of these works rely on control-flow graphs to reason about control flow and so cannot reason precisely about interprocedural control flows.

Moore et al. [42] prove termination-sensitive noninterference with a type system. Similarly, several works prove termination-insensitive noninterference in JavaScript [11], [17], [28]. TaintDroid [20] is a dynamic extension to Android's runtime environment. It does not purport to identify all possible program behaviors; instead, it monitors behaviors as they happen. These systems rely on runtime enforcement and can only identify or prevent behaviors manifest in a particular execution.

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. [50] perform a static prepass that adds tracking instructions to inform a dynamic analysis. This analysis preserves termination-insensitive noninterference. Jia et al. [31] present a system that allows programmers to provide annotations that are enforced dynamically. Myers [43] presents 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. Heule et al. [29] 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 et al. [14] present a Haskell library that performs a hybrid analysis. Programmers that use this library can guarantee noninterference in Haskell code. Their work uses the structure of the language to avoid some of the difficulties inherent to bytecode analysis. Vassena et al. [49] have recently continued this work.

VII. Conclusion

Standard noninterference analyses are limited by their inability to reason about interprocedural control flows. The augmented-state analysis proves noninterference in the presence of interprocedural control flows but is intractable. However, the *a posteriori* analysis is capable of tracking the same information flows in polynomial time. An implementation of the *a posteriori* analysis demonstrates that it is possible to perform such an analysis on real-world Android applications to substantial effect.

Acknowledgements

The authors thank Sean Brown for his contributions to the empirical evaluation. This article reports on work supported by the Defense Advanced Research Projects Agency under agreements no. AFRL FA8750-15-2-0092 and FA8750-12-2-0106. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] Peter Aldous, Eric Mercer, and Matthew Might. Some title.

[2] Peter Aldous and Matthew Might. Static analysis of non-interference in expressive low-level languages. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis*, volume 9291 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin Heidelberg, 2015.

[3] APAC. Automated program analysis for cybersecurity (apac). https://www.darpa.mil/program/automated-program-analysis-for-cybersecurity, 2016. Accessed 2017-09-25.

[4] Steven Arzt. Droidbench 2.0. https://github.com/secure-software-engineering/DroidBench, 2017.

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 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*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[6] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 333–348, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Aslan Askarov and Andrei Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Computer Security Foundations Symposium, 2009. CSF'09. 22nd IEEE*, pages 43–59. IEEE, 2009.

[8] Mounir Assaf, David A. Naumann, Julien Signoles, Éric Totel, and Frédéric Tronel. Hypercollecting semantics and its application to static analysis of information flow. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 874–887, New York, NY, USA, 2017. ACM.

[9] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In *ESOP*, volume 4421, pages 125–140. Springer, 2007.

[10] Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. *Mathematical Structures in Computer Science*, 23(5):10321081, 2013.

[11] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information flow control in webkit's javascript bytecode. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust*, pages 159–178, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[12] Simon Bischof, Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Low-deterministic security for low-deterministic programs. *Journal of Computer Security*, 26:335–336, 2018.

[13] Joachim Breitner, Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. On improvements of low-deterministic security. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9635 of *Lecture Notes in Computer Science*, pages 68–88. Springer Berlin Heidelberg, 2016.

[14] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: mixing static and dynamic typing for information-flow control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 289–301, New York, NY, USA, August 2015. ACM.

[15] Gurvan Cabon and Alan Schmitt. Non-interference through annotated multisemantics. In *28ièmes Journées Francophones des Langages Applicatifs*, 2017.

[16] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM.

[17] Andrey Chudnov and David A. Naumann. Inlined information flow monitoring for javascript. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 629–643, New York, NY, USA, 2015. ACM.

[18] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[19] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[20] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. pages 1–6, 2010.

[21] Matthias Felleisen and Daniel P. Friedman. A reduction semantics for imperative higher-order languages. In *Proceedings of the Parallel Architectures and Languages Europe, Volume I*, pages 206–223, London, UK, UK, 1987. Springer-Verlag.

[22] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 186–197, New York, NY, USA, 2004. ACM.

[23] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 691–704, New York, NY, USA, January 2016. ACM.

[24] Jürgen Graf. *Information Flow Control with System Dependence Graphs – Improving Modularity, Scalability and Precision for Object Oriented Languages*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, November 2016.

[25] Jürgen Graf, Martin Hecker, and Martin Mohr. Using joana for information flow control in java programs-a practical guide. In *Software Engineering (Workshops)*, volume 215, pages 123–138, 2013.

[26] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Checking applications using security apis with joana. July 2015. 8th International Workshop on Analysis of Security APIs.

[27] Jürgen Graf, Martin Hecker, Martin Mohr, and Gregor Snelting. Tool demonstration: Joana. In Frank Piessens and Luca Viganò, editors, *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9635 of *Lecture Notes in Computer Science*, pages 89–93. Springer Berlin Heidelberg, 2016.

[28] Daniel Hedin, Luciano Bello, and Andrei Sabelfeld. Information-flow security for javascript and its apis. *Journal of Computer Security*, 24(2):181–234, 2016.

[29] Stefan Heule, Deian Stefan, Edward Z Yang, John C Mitchell, and Alejandro Russo. Ifc inside: Retrofitting languages with dynamic information flow control. In *International Conference on Principles of Security and Trust*, pages 11–31. Springer, 2015.

[30] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 106–117, New York, NY, USA, 2015. ACM.

[31] Limin Jia, Jassim Aljuraidan, Elli Fragkaki, Lujo Bauer, Michael Stroucken, Kazuhide Fukushima, Shinsaku Kiyomoto, and Yutaka Miyake. Run-time enforcement of information-flow properties on android. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *Computer Security – ESORICS 2013*, volume 8134 of *Lecture Notes in Computer Science*, pages 775–792. Springer Berlin Heidelberg, 2013.

[32] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011*. The Internet Society, February 2011.

[33] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, and Junbum Shin. Scandal: Static analyzer for detecting privacy leaks in android applications. Mobile Security Technologies, 2012.

[34] Naoki Kobayashi and Keita Shirane. Type-based information flow analysis for low-level languages. In *In Proceedings of the 3rd Asian Workshop on Programming Languages and Systems (APLAS02*, pages 2–21, 2002.

[35] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. 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*, SPSM '13, pages 21–32, New York, NY, USA, 2013. ACM.

[36] Shuying Liang and Matthew Might. Hash-flow taint analysis of higher-order programs. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS '12, pages 8:1–8:12, New York, NY, USA, 2012. ACM.

[37] Yin Liu and Ana Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 146–155, March 2010.

[38] Steffen Lortz, Heiko Mantel, Artem Starostin, Timo Bähr, David Schneider, and Alexandra Weber. Cassandra: Towards a certifying app store for android. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, SPSM '14, pages 93–104, New York, NY, USA, 2014. ACM.

[39] Luísa Lourenço and Luís Caires. Dependent information flow types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL '15, pages 317–328, New York, NY, USA, January 2015. ACM.

[40] Matthew Might and Panagiotis Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 260–274, Berlin, Heidelberg, 2009. Springer-Verlag.

[41] Martin Mohr, Jürgen Graf, and Martin Hecker. Jodroid: Adding android support to a static information flow control tool. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015, Dresden, Germany, 17.-18. MÄrz 2015.*, volume 1337 of *CEUR Workshop Proceedings*, pages 140–145. CEUR-WS.org, 2015.

[42] Scott Moore, Aslan Askarov, and Stephen Chong. Precise enforcement of progress-sensitive security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 881–893, New York, NY, USA, 2012. ACM.

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

[44] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):117–158, January 2003.

[45] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, September 2006.

[46] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[47] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. Checking probabilistic noninterference using joana. *it - Information Technology*, 56:280–287, November 2014.

[48] David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62, New York, NY, USA, 2010. ACM.

[49] Marco Vassena, Pablo Buiras, Lucas Waye, and Alejandro Russo. Flexible manipulation of labeled values for information-flow control libraries. In *European Symposium on Research in Computer Security*, pages 538–557. Springer, 2016.

[50] V. N. Venkatakrishnan, Wei Xu, Daniel C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In Peng Ning, Sihan Qing, and Ninghui Li, editors, *Information and Communications Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 332–351. Springer Berlin Heidelberg, 2006.

[51] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2–3):167–187, January 1996.

[52] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 156–168, June 1997.

[53] Wei Xu, Sandeep Bhatkar, and R. Sekar. 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, Berkeley, CA, USA, 2006. USENIX Association.

$$
\begin{aligned}
prgm &\in \mathsf{Program} = \mathsf{ClassDef}^* \\
classdef &\in \mathsf{ClassDef} ::= \texttt{Class}\ className\ \{field_1, \ldots, field_n, m_1, \ldots, m_m\} \\
m &\in \mathsf{Method} ::= \texttt{Def}\ mName\ \{handler_1, \ldots, handler_n, stmt_1, \ldots, stm \\
handler &\in \mathsf{Handler} ::= \texttt{Catch}(ln, ln, ln) \\
stmt &\in \mathsf{Stmt} ::= ln\ \texttt{Const}(r, c) \\
&\quad \mid ln\ \texttt{Move}(r, r) \\
&\quad \mid ln\ \texttt{Invoke}(mName, r_1, \ldots, r_n) \\
&\quad \mid ln\ \texttt{Return}(r) \\
&\quad \mid ln\ \texttt{IfEqz}(r, ln) \\
&\quad \mid ln\ \texttt{Add}(r, r, r) \\
&\quad \mid ln\ \texttt{NewInstance}(r, className) \\
&\quad \mid ln\ \texttt{Throw}(r) \\
&\quad \mid ln\ \texttt{IGet}(r, r, field) \\
&\quad \mid ln\ \texttt{IPut}(r, r, field) \\
r &\in \mathsf{Register} = \{\texttt{r}, \texttt{e}, \texttt{0}, \texttt{1}, \ldots\} \\
ln &\in \mathsf{LineNumber} \text{ is a set of line numbers} \\
mName &\in \mathsf{MName} \text{ is a set of method names} \\
field &\in \mathsf{Field} \text{ is a set of field names} \\
c \in C &::= (ln, m)
\end{aligned}
$$

Fig. 7: Abstract syntax

APPENDIX

A. AAM abstract interpretation with instrumentation

This section presents semantics for a CESK abstract interpreter with bookkeeping for the data structures needed by the *a posteriori* taint analysis. For comparison, it includes augmented-state semantics. The two sets of semantics are placed side by side.

Taint propagation iterates over states until it reaches a fixed point. Because taint tracking is a may-analysis, the absence of a reported flow is a proof that no flow can exist. Figure 8 shows the CESK state space used in the analysis. For simplicity, it is presented without widening.

In this presentation, the general elements of Dalvik bytecode are summarized in a small set of instructions that cover the essential language features: data with movement, call/return, control, arithmetic, and exceptions. The `Const` and `Move` instructions write to a register; the former writes a constant value and the latter copies the contents of another register. The `Invoke` instruction calls a method and the `Return` instruction returns. The `NewInstance` instruction instantiates an object. `IGet` and `IPut` are specialized `Move` instructions. The former copies a value from an object to a register and the latter from a register to an object. Both `IGet` and `IPut` can throw exceptions when the object address given is `null`. The `IfEqz` instruction is a conditional branch. The `Add` instruction performs arithmetic and the `Throw` throws an exception. The abstract syntax is presented formally in Figure 7.

1) Restatement of definitions: This section restates definitions given in the main body of the paper with some

elaborations. Appendix A2 contains the semantics for the two analyses.

Because an abstract interpreter can use nondeterministic allocators [40], the *a posteriori* analysis also relies on a record of which addresses are written and read at each state during abstract interpretation. This record keeping can occur in a writes map named Δ that is external to the state space (and can use any widening scheme). Δ maps addresses written to addresses read so that information is recorded about every write, even if the value written is a constant. This structure simplifies the propagation of taints that identify implicit information flows. Although it is not incorporated into these simplified semantics, it is also useful to record writes that result from a call to a source.

This section contains semantics for the augmented-state abstract interpretation and for the abstract interpretation phase of the *a posteriori* analysis. Analogous rules are presented adjacent to each other for easy comparison. Stores, taint stores, and context taint maps may be widened; however, only the context taint map is widened in this presentation; widened context taint simplifies the presentation of context taint aggregation as an exception unwinds the stack. The construction of β and ψ is most efficiently done as a postprocessing step after abstract interpretation and may be performed lazily. However, it is presented with the abstract interpretation semantics to more clearly show the relationship between the augmented-state semantics and the *a posteriori* semantics.

The following definitions simplify the presentation of both sets of semantics: The successor of a code point c is $N(c)$. A state's execution point is abbreviated $\hat{e}_\varsigma$. In the augmented-state analysis, the set of implicit taint values created from context taint at a state is notated $\widehat{\Theta}_\Omega$. As before, $\overset{\sqcup}{\mapsto}$ indicates weak update; e.g., $\hat{\sigma}\left[\hat{a} \overset{\sqcup}{\mapsto} \widehat{v}\right] \equiv \hat{\sigma}\left[\hat{a} \mapsto \widehat{v} \sqcup \hat{\sigma}(\hat{a})\right]$.

In the *a posteriori* analysis, the notation $\Delta(c)(\hat{a}_d) \overset{\sqcup}{\coloneqq} \{\hat{a}_s\}$ for some destination address $\hat{a}_d$ and some source address $\hat{a}_s$ indicates that $\Delta(c)$, the writes map at c, is updated weakly in place. In other words, $\Delta(c)\left[\hat{a}_d \overset{\sqcup}{\mapsto} \{\hat{a}_s\}\right]$ replaces $\Delta(c)$.

The function α computes abstract values; $\alpha(0)$ is the abstraction of 0 to the appropriate abstract domain.

In both analyses, context taints are widened to a context taint map Ξ, which contains a (possibly empty) map $\widehat{\omega} \in \widehat{\Omega} = \hat{E} \to \widehat{\Theta}$ at each execution point $\hat{e}$. Each binding in a context taint map $\hat{e}_b \mapsto \widehat{\Theta}$ indicates a branching execution point and the taints on the branch's condition. For example, if a program branches at $\hat{e}_b$ on a condition marked with a single taint value $\widehat{\theta}$ and if the program reaches $\hat{e}$ before reaching $EPG(\hat{e}_b)$, then Ξ would contain the appropriate mapping: $\Xi(\hat{e}) = \left[\hat{e}_b \mapsto \left\{\widehat{\theta}\right\}\right]$.

The context taint map is updated weakly in the augmented-state semantics:

$$\Xi(\hat{e}_c) \overset{\sqcup}{\coloneqq} \widehat{\omega} \equiv \forall\ \hat{e}_b \in d(\widehat{\omega}) : \Xi(\hat{e}_c)(\hat{e}_b) \coloneqq \Xi(\hat{e}_c)(\hat{e}_b) \cup \widehat{\omega}(\hat{e}_b)\ .$$

2) Abstract interpretation semantics: Semantics for the `Const` instruction demonstrates augmented-state implicit taint propagation and an update to Δ for later use in *a posteriori* taint tracking.

$$\frac{\widehat{sa} = \left(\hat{\phi}, r\right)\quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \alpha(c)\right] \quad \Delta(c)(\widehat{sa}) \overset{\sqcup}{\coloneqq} \emptyset \quad \mathcal{I}(c) = \texttt{Const}(r,\ c)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(N(c), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}$$

$$\frac{\widehat{sa} = \left(\hat{\phi}, r\right)\quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \alpha(c)\right] \quad \widehat{\tau}' = \widehat{\tau}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \widehat{\Theta}_\Omega\right] \quad \hat{e}' \in EPG(\hat{e}_\varsigma)\quad \Xi(\hat{e}') \overset{\sqcup}{\coloneqq} \Xi(\hat{e}_\varsigma) \quad \mathcal{I}(c) = \texttt{Const}(r,\ c)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(N(c), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{\tau}'\right)}$$

The `Move` instruction adds explicit taint propagation to the augmented-state semantics and instrumentation of Δ to the *a posteriori* analysis:

$$\frac{\widehat{sa}_d = \left(\hat{\phi}, r_d\right)\quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}(\widehat{sa}_s)\right] \quad \widehat{sa}_s = \left(\hat{\phi}, r_s\right)\quad \Delta(c)(\widehat{sa}_d) \overset{\sqcup}{\coloneqq} \{\widehat{sa}_s\} \quad \mathcal{I}(c) = \texttt{Move}(r_d,\ r_s)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(N(c), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}$$

$$\frac{\widehat{sa}_d = \left(\hat{\phi}, r_d\right)\quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}(\widehat{sa}_s)\right] \quad \widehat{sa}_s = \left(\hat{\phi}, r_s\right)\quad \widehat{\tau}' = \widehat{\tau}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{\tau}(\widehat{sa}_s) \cup \widehat{\Theta}_\Omega\right] \quad \hat{e}' \in EPG(\hat{e}_\varsigma)\quad \Xi(\hat{e}') \overset{\sqcup}{\coloneqq} \Xi(\hat{e}_\varsigma) \quad \mathcal{I}(c) = \texttt{Move}(r_d,\ r_s)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(N(c), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{\tau}'\right)}$$

The `Invoke` instruction performs several writes. In this presentation, the object is assumed to be non-null. However, later instructions demonstrate explicit and implicit exceptions and the same mechanism can be added to `Invoke`.

$$\frac{c' = init(\mathcal{M}(mName))\quad \hat{\kappa}' = \left(c, \hat{\phi}, \hat{\kappa}\right) \quad \hat{\phi}' \text{ is a fresh frame pointer} \quad \text{for each } i \text{ from 1 to } n, \quad \widehat{sa}_{si} = \left(\hat{\phi}, r_i\right)\quad \widehat{sa}_{di} = \left(\hat{\phi}', i\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_{d1} \overset{\sqcup}{\mapsto} \hat{\sigma}(\widehat{sa}_{s1}), \ldots, \widehat{sa}_{dn} \overset{\sqcup}{\mapsto} \hat{\sigma}(\widehat{sa}_{sn})\right] \quad \Delta(c)(\widehat{sa}_{d1}) \overset{\sqcup}{\coloneqq} \{\widehat{sa}_{s1}\}, \ldots, \widehat{sa}_{dn} \overset{\sqcup}{\mapsto} \{\widehat{sa}_{sn}\} \quad \beta(\hat{e}_\varsigma) = \{\widehat{sa}_s\} \quad \mathcal{I}(c) = \texttt{Invoke}(mName,\ r_1,\ \ldots,\ r_n)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(c', \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}$$

$$\frac{c' = init\left(\mathcal{M}\left(mName\right)\right) \quad \hat{\kappa}' = \left(c, \hat{\phi}, \hat{\kappa}\right) \quad \hat{\phi}' \text{ is a fresh frame pointer} \quad \text{for each } i \text{ from 1 to } n, \quad \widehat{sa}_{si} = \left(\hat{\phi}, r_i\right) \quad \widehat{sa}_{di} = \left(\hat{\phi}', i\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_{d1} \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_{s1}\right), \ldots, \widehat{sa}_{dn} \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_{sn}\right)\right] \quad \widehat{\tau}' = \widehat{\tau}\left[\widehat{sa}_{d1} \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}_{s1}\right), \ldots, \widehat{sa}_{dn} \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}_{sn}\right)\right] \quad \hat{e}' \in EPG\left(\hat{e}_\varsigma\right) \quad \Xi\left(\hat{e}'\right) \overset{\sqcup}{\coloneqq} \Xi\left(\hat{e}_\varsigma\right)\left[\hat{e}_\varsigma \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}_{s1}\right)\right] \quad \mathcal{I}\left(c\right) = \texttt{Invoke}\left(mName, r_1, \ldots, r_n\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(c', \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{\tau}'\right)}$$

The `Return` instruction takes two cases. In one case, the continuation is not **halt** and it writes to the special result register `r`:

$$\frac{\hat{\kappa} = \left(c, \hat{\phi}', \widehat{ka}\right) \quad \hat{\kappa}' \in \hat{\sigma}\left(\widehat{ka}\right) \quad \widehat{sa}_s = \left(\hat{\phi}, r\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_s\right)\right] \quad \widehat{sa}_d = \left(\hat{\phi}', \texttt{r}\right) \quad \Delta\left(c\right)\left(\widehat{sa}_d\right) \overset{\sqcup}{\coloneqq} \{\widehat{sa}_s\} \quad \mathcal{I}\left(c\right) = \texttt{Return}\left(r\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(N\left(c\right), \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}$$

$$\frac{\hat{\kappa} = \left(c, \hat{\phi}', \widehat{ka}\right) \quad \hat{\kappa}' \in \hat{\sigma}\left(\widehat{ka}\right) \quad \widehat{sa}_s = \left(\hat{\phi}, r\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_s\right)\right] \quad \widehat{sa}_d = \left(\hat{\phi}', \texttt{r}\right) \quad \widehat{\tau}' = \widehat{\tau}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}_s\right)\right] \quad \hat{e}' \in EPG\left(\hat{e}_\varsigma\right) \quad \Xi\left(\hat{e}'\right) \overset{\sqcup}{\coloneqq} \Xi\left(\hat{e}_\varsigma\right) \quad \mathcal{I}\left(c\right) = \texttt{Return}\left(r\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(N\left(c\right), \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{\tau}'\right)}$$

In the other, the continuation is **halt** and nothing is written:

$$\frac{\hat{\kappa} = \textbf{halt} \quad \mathcal{I}\left(c\right) = \texttt{Return}\left(r\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \textbf{endstate}}$$

$$\frac{\hat{\kappa} = \textbf{halt} \quad \mathcal{I}\left(c\right) = \texttt{Return}\left(r\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \textbf{endstate}}$$

The `IfEqz` instruction does not write to the store. The two cases are not mutually exclusive. First is the case when the condition may be zero:

$$\frac{c' = jump\left(c, ln\right) \quad \alpha\left(0\right) \sqsubseteq \sigma\left(\widehat{sa}\right) \quad \widehat{sa} = \left(\hat{\phi}, r\right) \quad \beta\left(\hat{e}_\varsigma\right) = \{\widehat{sa}\} \quad \mathcal{I}\left(c\right) = \texttt{IfEqz}\left(r, ln\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(c', \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right)}$$

$$\frac{c' = jump\left(c, ln\right) \quad \alpha\left(0\right) \sqsubseteq \sigma\left(\widehat{sa}\right) \quad \widehat{sa} = \left(\hat{\phi}, r\right) \quad \hat{e}' \in EPG\left(\hat{e}_\varsigma\right) \quad \Xi\left(\hat{e}'\right) \overset{\sqcup}{\coloneqq} \Xi\left(\hat{e}_\varsigma\right)\left[\hat{e}_\varsigma \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}\right)\right] \quad \mathcal{I}\left(c\right) = \texttt{IfEqz}\left(r, ln\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(c', \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right)}$$

In the second case, the condition may be nonzero:

$$\frac{\widehat{sa} = \left(\hat{\phi}, r\right) \quad c' = N\left(c\right) \quad \exists z \in \mathbb{Z} : z \neq 0 \wedge \alpha\left(z\right) \sqsubseteq \sigma\left(\widehat{sa}\right) \quad \beta\left(\hat{e}_\varsigma\right) = \{\widehat{sa}\} \quad \mathcal{I}\left(c\right) = \texttt{IfEqz}\left(r, ln\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(c', \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right)}$$

$$\frac{\widehat{sa} = \left(\hat{\phi}, r\right) \quad c' = N\left(c\right) \quad \exists z \in \mathbb{Z} : z \neq 0 \wedge \alpha\left(z\right) \sqsubseteq \sigma\left(\widehat{sa}\right) \quad \hat{e}' \in EPG\left(\hat{e}_\varsigma\right) \quad \Xi\left(\hat{e}'\right) \overset{\sqcup}{\coloneqq} \Xi\left(\hat{e}_\varsigma\right)\left[\hat{e}_\varsigma \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}\right)\right] \quad \mathcal{I}\left(c\right) = \texttt{IfEqz}\left(r, ln\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(c', \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right)}$$

The `Add` instruction is straightforward:

$$\frac{\hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_l\right) \hat{+} \hat{\sigma}\left(\widehat{sa}_r\right)\right] \quad \widehat{sa}_r = \left(\hat{\phi}, r_r\right) \quad \widehat{sa}_d = \left(\hat{\phi}, r_d\right) \quad \widehat{sa}_l = \left(\hat{\phi}, r_l\right) \quad \Delta\left(c\right)\left(\widehat{sa}_d\right) \overset{\sqcup}{\coloneqq} \{\widehat{sa}_l, \widehat{sa}_r\} \quad \mathcal{I}\left(c\right) = \texttt{Add}\left(r_d, r_l, r_r\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(N\left(c\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}$$

$$\frac{\hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_l\right) \hat{+} \hat{\sigma}\left(\widehat{sa}_r\right)\right] \quad \widehat{sa}_r = \left(\hat{\phi}, r_r\right) \quad \widehat{sa}_d = \left(\hat{\phi}, r_d\right) \quad \widehat{sa}_l = \left(\hat{\phi}, r_l\right) \quad \widehat{\tau}' = \widehat{\tau}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}_l\right) \cup \widehat{\tau}\left(\widehat{sa}_r\right)\right] \quad \hat{e}' \in EPG\left(\hat{e}_\varsigma\right) \quad \Xi\left(\hat{e}'\right) \overset{\sqcup}{\coloneqq} \Xi\left(\hat{e}_\varsigma\right) \quad \mathcal{I}\left(c\right) = \texttt{Add}\left(r_d, r_l, r_r\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(N\left(c\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{\tau}'\right)}$$

The same is true of the `NewInstance` instruction:

$$\frac{\widehat{sa} = \left(\hat{\phi}, r\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \widehat{oa}\right] \quad \widehat{oa} \text{ is a fresh object address} \quad \Delta(c)(\widehat{sa}_d) \overset{\sqcup}{\coloneqq} \{\widehat{oa}\} \quad \mathcal{I}(c) = \texttt{NewInstance}(r,\ className)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(N(c), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}$$

$$\frac{\widehat{sa} = \left(\hat{\phi}, r\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa} \overset{\sqcup}{\mapsto} \widehat{oa}\right] \quad \widehat{oa} \text{ is a fresh object address} \quad \widehat{\tau}' = \widehat{\tau}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{\tau}(\widehat{oa})\right] \quad \hat{e}' \in EPG(\hat{e}_\varsigma) \quad \Xi(\hat{e}') \overset{\sqcup}{\coloneqq} \Xi(\hat{e}_\varsigma) \quad \mathcal{I}(c) = \texttt{NewInstance}(r,\ className)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(N(c), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{\tau}'\right)}$$

The relations $\hat{\mathcal{T}} \subseteq \left(C \times \widehat{\Phi} \times \widehat{S} \times \widehat{K} \times \widehat{\Omega}\right) \times \left(\left(C \times \widehat{\Phi} \times \widehat{K} \times \widehat{\Omega}\right) \cup \{error\}\right)$ (for the augmented-state analysis) and $\hat{\mathcal{T}}_p \subseteq \left(C \times \widehat{\Phi} \times \widehat{S} \times \widehat{K} \times \mathcal{P}\left(\hat{E}\right)\right) \times \left(\left(C \times \widehat{\Phi} \times \widehat{K} \times \mathcal{P}\left(\hat{E}\right)\right) \cup \{error\}\right)$ (for the *a posteriori* analysis) simplify the presentation of instructions that throw exceptions. Both make use of $\mathcal{H} : C \rightharpoonup C$, which finds a method-local exception handler for a code point, if it exists.

These relations are selectively transitive; when a suitable exception handler does not exist in the current method, the interpreter searches for a suitable handler in the program stack. $\hat{\mathcal{T}}$ accumulates taint values from Ξ, while $\hat{\mathcal{T}}_p$ accumulates execution points to build ψ.

Both relations are defined in three cases. In the first, a suitable exception is found:

$$\frac{\mathcal{H}(c) = c_h}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \hat{E}_\psi\right) \hat{\mathcal{T}}_p \left(c_h, \hat{\phi}, \hat{\kappa}, \hat{E}_\psi\right)}$$

$$\frac{\mathcal{H}(c) = c_h}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\Theta}\right) \hat{\mathcal{T}} \left(c_h, \hat{\phi}, \hat{\kappa}, \widehat{\Theta}\right)}$$

In the second case, no suitable handler is found and the current continuation is **halt**:

$$\frac{c \notin d(\mathcal{H}) \wedge \hat{\kappa} = \mathbf{halt}}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \hat{E}_\psi\right) \hat{\mathcal{T}}_p \ error}$$

$$\frac{c \notin d(\mathcal{H}) \wedge \hat{\kappa} = \mathbf{halt}}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\Theta}\right) \hat{\mathcal{T}} error}$$

In the last case, no suitable handler is found in the current method, so the stack unwinds.

$$\frac{c \notin d(\mathcal{H}) \quad \hat{\kappa} = \left(c_\kappa, \hat{\phi}_\kappa, \widehat{ka}\right) \quad \hat{\kappa}_\kappa \in \hat{\sigma}\left(\widehat{ka}\right) \quad \hat{e}_\mathcal{T} = \left(c_\kappa, \widehat{SH}(\hat{\sigma}, \hat{\kappa}_\kappa)\right) \quad \left(c_\kappa, \hat{\phi}_k, \hat{\sigma}, \hat{\kappa}_\kappa, \hat{E}_\psi \cup \{\hat{e}_\mathcal{T}\}\right) \hat{\mathcal{T}}_p \ n}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \hat{E}_\psi\right) \hat{\mathcal{T}}_p \ n}$$

$$\frac{c \notin d(\mathcal{H}) \quad \hat{\kappa} = \left(c_\kappa, \hat{\phi}_\kappa, \widehat{ka}\right) \quad \hat{\kappa}_\kappa \in \hat{\sigma}\left(\widehat{ka}\right) \quad \hat{e}_\mathcal{T} = \left(c_\kappa, \widehat{SH}(\hat{\sigma}, \hat{\kappa}_\kappa)\right) \quad \widehat{\Theta}' = \widehat{\Theta} \cup \left\{\widehat{\theta} \mid \exists \hat{e}_\psi : \widehat{\theta} \in \Xi(\hat{e}_\mathcal{T})(\hat{e}_\psi)\right\} \quad \left(c_\kappa, \hat{\phi}_k, \hat{\sigma}, \hat{\kappa}_\kappa, \widehat{\Theta}'\right) \hat{\mathcal{T}} \ n}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\Theta}\right) \hat{\mathcal{T}} \ n}$$

There are two nonexclusive cases for the `Throw` instruction. The first case demonstrates a caught exception. When an exception is caught, it writes to the special exception register `e`:

$$\frac{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}}_p \left(c', \hat{\phi}', \hat{\kappa}', \hat{E}_\psi\right) \quad \widehat{sa}_d = \left(\hat{\phi}', \mathtt{e}\right) \quad \widehat{oa} \in \hat{\sigma}(\widehat{sa}_s) \cap \widehat{OA} \quad \widehat{sa}_s = \left(\hat{\phi}, r\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{oa}\right] \quad \Delta(c)(\widehat{sa}_d) \overset{\sqcup}{\coloneqq} \{\widehat{sa}_s, \widehat{oa}\} \quad \beta(\hat{e}_\varsigma) = \{\widehat{sa}_s, \widehat{oa}\} \quad \hat{e}' = \left(c_\kappa, \widehat{SH}(\hat{\sigma}, \hat{\kappa}_\kappa)\right) \quad \psi(\hat{e}_\varsigma)(\hat{e}') \overset{\sqcup}{\coloneqq} \hat{E}_\psi \quad \mathcal{I}(c) = \texttt{Throw}(r)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(c', \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}$$

$$\frac{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}} \left(c', \hat{\phi}', \hat{\kappa}', \widehat{\Theta}_\psi\right) \quad \widehat{sa}_d = \left(\hat{\phi}', \mathtt{e}\right) \quad \widehat{oa} \in \hat{\sigma}(\widehat{sa}_s) \cap \widehat{OA} \quad \widehat{sa}_s = \left(\hat{\phi}, r\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{oa}\right] \quad \widehat{\tau}' = \widehat{\tau}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{\tau}(\widehat{sa}_s) \cup \widehat{\tau}(\widehat{oa})\right] \quad \hat{e}' \in EPG(\hat{e}_\varsigma) \quad \Xi(\hat{e}') \overset{\sqcup}{\coloneqq} \Xi(\hat{e}_\varsigma)\left[\hat{e}_\varsigma \overset{\sqcup}{\mapsto} \widehat{\Theta}_\psi\right] \quad \mathcal{I}(c) = \texttt{Throw}(r)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(c', \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{\tau}'\right)}$$

The second case demonstrates an uncaught exception:

$$\frac{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}}_p \ error \quad \mathcal{I}(c) = \texttt{Throw}(r)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \mathbf{errorstate}}$$

$$\frac{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}} \mathit{error} \quad \mathcal{I}(c) = \texttt{Throw}\,(r)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \textbf{errorstate}}$$

The IGet requires three nonexclusive transition rules. The first demonstrates normal execution:

$$\frac{\begin{array}{c}\exists oa \neq \texttt{null} : \alpha(oa) \sqsubseteq \widehat{oa} \\ \widehat{sa}_d = \left(\hat{\phi}, r_d\right) \quad \widehat{sa}_o = \left(\hat{\phi}, r_o\right) \\ \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \quad fa = (\widehat{oa}, \mathit{field}) \\ \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{fa}\right)\right] \\ \Delta(c)\left(\widehat{sa}_d\right) \overset{\sqcup}{:=} \left\{\widehat{sa}_o, \widehat{oa}, \widehat{fa}\right\} \\ \beta\left(\hat{e}_\varsigma\right) = \{\widehat{sa}_s, \widehat{oa}\} \\ \mathcal{I}(c) = \texttt{IGet}\,(r_d, \ r_o, \ \mathit{field})\end{array}}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(N(c), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}$$

$$\frac{\begin{array}{c}\exists oa \neq \texttt{null} : \alpha(oa) \sqsubseteq \widehat{oa} \\ \widehat{sa}_d = \left(\hat{\phi}, r_d\right) \quad \widehat{sa}_o = \left(\hat{\phi}, r_o\right) \\ \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \quad fa = (\widehat{oa}, \mathit{field}) \\ \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{fa}\right)\right] \quad \hat{e}' \in \mathit{EPG}\left(\hat{e}_\varsigma\right) \\ \widehat{\tau}' = \widehat{\tau}\left[\widehat{sa}_d \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}_o\right) \cup \widehat{\tau}\left(\widehat{oa}\right) \cup \widehat{\tau}\left(\widehat{fa}\right)\right] \\ \Xi\left(\hat{e}'\right) \overset{\sqcup}{:=} \Xi\left(\hat{e}_\varsigma\right)\left[\hat{e}_\varsigma \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}_o\right) \cup \widehat{\tau}\left(\widehat{oa}\right)\right] \\ \mathcal{I}(c) = \texttt{IGet}\,(r_d, \ r_o, \ \mathit{field})\end{array}}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(N(c), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{\tau}'\right)}$$

The second demonstrates an exception being thrown and caught:

$$\frac{\begin{array}{c}\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}}_p \left(c', \hat{\phi}', \hat{\kappa}', \hat{E}_\psi\right) \\ \widehat{sa}_o = \left(\hat{\phi}, r_o\right) \quad \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \\ \texttt{null} \sqsubseteq \widehat{oa} \quad \widehat{sa}_{ex} = \left(\hat{\phi}', \texttt{e}\right) \\ \widehat{oa}_{ex} \text{ is a fresh object address} \\ \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{oa}_{ex}\right] \\ A = \{\widehat{sa}_o, \widehat{oa}\} \\ \Delta(c)\left(\widehat{sa}_{ex}\right) \overset{\sqcup}{:=} A \quad \Delta(c)\left(\widehat{oa}_{ex}\right) \overset{\sqcup}{:=} A \\ \beta\left(\hat{e}_\varsigma\right) = \{\widehat{sa}_o, \widehat{oa}\} \\ \hat{e}' = \left(c_\kappa, \widehat{SH}\left(\hat{\sigma}, \hat{\kappa}_\kappa\right)\right) \quad \psi\left(\hat{e}_\varsigma\right)\left(\hat{e}'\right) \overset{\sqcup}{:=} \hat{E}_\psi \\ \mathcal{I}(c) = \texttt{IGet}\,(r_d, \ r_o, \ \mathit{field})\end{array}}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(c', \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}$$

$$\frac{\begin{array}{c}\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}} \left(c', \hat{\phi}', \hat{\kappa}', \widehat{\Theta}_\psi\right) \\ \widehat{sa}_o = \left(\hat{\phi}, r_o\right) \quad \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \\ \texttt{null} \sqsubseteq \widehat{oa} \quad \widehat{sa}_{ex} = \left(\hat{\phi}', \texttt{e}\right) \\ \widehat{oa}_{ex} \text{ is a fresh object address} \\ \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{oa}_{ex}\right] \\ \widehat{\Theta}_o = \widehat{\tau}\left(\widehat{sa}_o\right) \cup \widehat{\tau}\left(\widehat{oa}\right) \\ \widehat{\tau}' = \widehat{\tau}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{\Theta}_o, \widehat{oa}_{ex} \overset{\sqcup}{\mapsto} \widehat{\Theta}_o\right] \\ \Xi\left(\hat{e}'\right) \overset{\sqcup}{:=} \Xi\left(\hat{e}_\varsigma\right)\left[\hat{e}_\varsigma \overset{\sqcup}{\mapsto} \widehat{\Theta}_o \cup \widehat{\Theta}_\psi\right] \\ \mathcal{I}(c) = \texttt{IGet}\,(r_d, \ r_o, \ \mathit{field})\end{array}}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(c', \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{\tau}'\right)}$$

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

$$\frac{\texttt{null} \sqsubseteq \widehat{oa} \quad \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \quad \widehat{sa}_o = \left(\hat{\phi}, r_o\right) \quad \left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}}_p \ \mathit{error} \quad \mathcal{I}(c) = \texttt{IGet}\,(r_d, \ r_o, \ \mathit{field})}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \textbf{errorstate}}$$

$$\frac{\texttt{null} \sqsubseteq \widehat{oa} \quad \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \quad \widehat{sa}_o = \left(\hat{\phi}, r_o\right) \quad \left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}} \ \mathit{error} \quad \mathcal{I}(c) = \texttt{IGet}\,(r_d, \ r_o, \ \mathit{field})}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \textbf{errorstate}}$$

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

$$\frac{\begin{array}{c}\widehat{sa}_s = \left(\hat{\phi}, r_s\right) \quad \widehat{sa}_o = \left(\hat{\phi}, r_o\right) \quad \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \\ \exists oa \neq \texttt{null} : \alpha(oa) \sqsubseteq \widehat{oa} \quad \widehat{fa} = (\widehat{oa}, \mathit{field}) \\ \hat{\sigma}' = \hat{\sigma}\left[\widehat{fa} \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_s\right)\right] \\ \Delta(c)\left(\widehat{fa}\right) \overset{\sqcup}{:=} \{\widehat{sa}_o, \widehat{oa}, \widehat{sa}_s\} \\ \beta\left(\hat{e}_\varsigma\right) = \{\widehat{sa}_o, \widehat{oa}\} \\ \mathcal{I}(c) = \texttt{IPut}\,(r_s, \ r_o, \ \mathit{field})\end{array}}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(N(c), \hat{\phi}, \hat{\sigma}', \hat{\kappa}\right)}$$

$$\widehat{sa}_s = \left(\hat{\phi}, r_s\right) \quad \widehat{sa}_o = \left(\hat{\phi}, r_o\right) \quad \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right)$$
$$\exists oa \neq \texttt{null} : \alpha\left(oa\right) \sqsubseteq \widehat{oa} \quad \widehat{fa} = \left(\widehat{oa}, \mathit{field}\right)$$
$$\hat{\sigma}' = \hat{\sigma}\left[\widehat{fa} \overset{\sqcup}{\mapsto} \hat{\sigma}\left(\widehat{sa}_s\right)\right]$$
$$\widehat{\tau}' = \widehat{\tau}\left[\widehat{fa} \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}_o\right) \cup \widehat{\tau}\left(\widehat{oa}\right) \cup \widehat{\tau}\left(\widehat{sa}_s\right)\right]$$
$$\Xi\left(\hat{e}'\right) \overset{\sqcup}{:=} \Xi\left(\hat{e}_\varsigma\right)\left[\hat{e}_\varsigma \overset{\sqcup}{\mapsto} \widehat{\tau}\left(\widehat{sa}_o\right) \cup \widehat{\tau}\left(\widehat{oa}\right)\right]$$
$$\frac{\mathcal{I}\left(c\right) = \texttt{IPut}\left(r_s, \ r_o, \ \mathit{field}\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(N\left(c\right), \hat{\phi}, \hat{\sigma}', \hat{\kappa}, \widehat{\tau}'\right)}$$

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

$$\texttt{null} \sqsubseteq \widehat{oa} \quad \left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}}_p \left(c', \hat{\phi}', \hat{\kappa}', \hat{E}_\psi\right)$$
$$\widehat{sa}_o = \left(\hat{\phi}, r_o\right) \quad \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right)$$
$\widehat{oa}_{ex}$ is a fresh object address
$$\widehat{sa}_{ex} = \left(\hat{\phi}', \mathtt{e}\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{oa}_{ex}\right]$$
$$\Delta\left(c\right)\left(\widehat{sa}_{ex}\right) \overset{\sqcup}{:=} \{\widehat{sa}_o, \widehat{oa}\}$$
$$\Delta\left(c\right)\left(\widehat{oa}_{ex}\right) \overset{\sqcup}{:=} \{\widehat{sa}_o, \widehat{oa}\}$$
$$\beta\left(\hat{e}_\varsigma\right) = \{\widehat{sa}_o, \widehat{oa}\}$$
$$\hat{e}' = \left(c_\kappa, \widehat{SH}\left(\hat{\sigma}, \hat{\kappa}_\kappa\right)\right) \quad \psi\left(\hat{e}_\varsigma\right)\left(\hat{e}'\right) \overset{\sqcup}{:=} \hat{E}_\psi$$
$$\frac{\mathcal{I}\left(c\right) = \texttt{IPut}\left(r_s, \ r_o, \ \mathit{field}\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \left(c', \hat{\phi}', \hat{\sigma}', \hat{\kappa}'\right)}$$

$$\texttt{null} \sqsubseteq \widehat{oa} \quad \left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}}_p \left(c', \hat{\phi}', \hat{\kappa}', \widehat{\Theta}_\psi\right)$$
$$\widehat{sa}_o = \left(\hat{\phi}, r_o\right) \quad \widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right)$$
$\widehat{oa}_{ex}$ is a fresh object address
$$\widehat{sa}_{ex} = \left(\hat{\phi}', \mathtt{e}\right) \quad \hat{\sigma}' = \hat{\sigma}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{oa}_{ex}\right]$$
$$\widehat{\Theta}_o = \widehat{\tau}\left(\widehat{sa}_o\right) \cup \widehat{\tau}\left(\widehat{oa}\right)$$
$$\widehat{\tau}' = \widehat{\tau}\left[\widehat{sa}_{ex} \overset{\sqcup}{\mapsto} \widehat{\Theta}_o, \widehat{oa}_{ex} \overset{\sqcup}{\mapsto} \widehat{\Theta}_o\right]$$
$$\Xi\left(\hat{e}'\right) \overset{\sqcup}{:=} \Xi\left(\hat{e}_\varsigma\right)\left[\hat{e}_\varsigma \overset{\sqcup}{\mapsto} \widehat{\Theta}_o \cup \widehat{\Theta}_\psi\right]$$
$$\frac{\mathcal{I}\left(c\right) = \texttt{IPut}\left(r_s, \ r_o, \ \mathit{field}\right)}{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \left(c', \hat{\phi}', \hat{\sigma}', \hat{\kappa}', \widehat{\tau}'\right)}$$

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

$$\texttt{null} \sqsubseteq \widehat{oa} \quad \left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}}_p \ \mathit{error} \quad \widehat{sa}_o = \left(\hat{\phi}, r_o\right)$$
$$\widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \quad \mathcal{I}\left(c\right) = \texttt{IPut}\left(r_s, \ r_o, \ \mathit{field}\right)$$
$$\overline{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \rightsquigarrow \textbf{errorstate}}$$

$$\texttt{null} \sqsubseteq \widehat{oa} \quad \left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right) \hat{\mathcal{T}} \ \mathit{error} \quad \widehat{sa}_o = \left(\hat{\phi}, r_o\right)$$
$$\widehat{oa} \in \hat{\sigma}\left(\widehat{sa}_o\right) \quad \mathcal{I}\left(c\right) = \texttt{IPut}\left(r_s, \ r_o, \ \mathit{field}\right)$$
$$\overline{\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \widehat{\tau}\right) \rightsquigarrow \textbf{errorstate}}$$

B. Postprocessing

This section presents the semantics for postprocessing to create EPG, β, and ψ after abstract interpretation has completed and the execution point graph is fully formed. Construction is straightforward: one iteration over the state graph in any order suffices.

An execution point is created from each state by pairing its code point c with $\widehat{SH}\left(\hat{\sigma}, \hat{\kappa}, \emptyset\right)$, where $\hat{\sigma}$ is the state's abstract store and $\hat{\kappa}$ is the state's continuation. When one state succeeds another in the abstract state graph, its execution point succeeds the other state's execution point in the execution point graph. In this way, the execution point graph resembles a projection of the abstract state graph. The store $\hat{\sigma}$ is used to look up the successors of the continuation $\hat{\kappa}$. The set $\widehat{K}_s$ of continuations ensures that $\widehat{SH}$ avoids exploring loops infinitely.

$$\widehat{SH}\left(\hat{\sigma}, \hat{\kappa}, \widehat{K}_s\right) = \begin{cases} 0 & \hat{\kappa} = \textbf{halt} \\ n+1 & \hat{\kappa} \notin \widehat{K}_s \wedge \forall \hat{\kappa}' \in \hat{\sigma}\left(\hat{\kappa}.\widehat{ka}\right) : \\ & \quad \widehat{SH}\left(\hat{\sigma}, \hat{\kappa}', \widehat{K}_s \cup \{\hat{\kappa}\}\right) = n \\ ? & \text{otherwise}. \end{cases}$$

β is constructed at each state $\hat{\varsigma} = \left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right)$ whose execution point is $\hat{e}_\varsigma$, information flows according to the semantics for the state's instruction. In the case of many instructions, control flow is unaffected and the resulting set of addresses is empty:

$$\mathcal{I}\left(c\right) = \texttt{Move}\left(r_d, \ r_s\right) \vee \mathcal{I}\left(c\right) = \texttt{Const}\left(r, \ c\right) \vee \mathcal{I}\left(c\right) = \texttt{Return}\left(r\right) \vee \mathcal{I}$$
$$\mathcal{I}\left(c\right) = \texttt{NewInstance}\left(r, \ \mathit{className}\right)$$
$$\overline{\beta\left(\hat{e}_\varsigma\right) = \emptyset}$$

The remaining instructions do branch based on values, so β is non-empty.

$$\frac{\mathcal{I}\left(c\right) = \texttt{IfEqz}\left(r, \ ln\right)}{\beta\left(\hat{e}_\varsigma\right) = \left\{\left(\hat{\phi}, r\right)\right\}}$$

In each remaining case, the instruction branches based on a single object stored in a single register. In each of these cases, store lookup defaults to an empty set if the store has no mapping for $\widehat{sa}$:

$$\left(\mathcal{I}\left(c\right) = \texttt{Invoke}\left(r_1, \ \ldots, \ r_n\right) \wedge \widehat{sa} = \left(\hat{\phi}, r_1\right)\right) \vee \left(\mathcal{I}\left(c\right) = \texttt{Throw}\left(r\right)\right.$$
$$\left(\mathcal{I}\left(c\right) = \texttt{IGet}\left(r_d, \ r_o, \ \mathit{field}\right) \wedge \widehat{sa} = \left(\hat{\phi}, r_o\right)\right) \vee \left(\mathcal{I}\left(c\right) = \texttt{IPut}\left(r_s, \ r_o\right.\right.$$
$$\overline{\beta\left(\hat{e}_\varsigma\right) = \{\widehat{sa}\} \cup \left(\hat{\sigma}\left(\widehat{sa}\right) \cap \widehat{OA}\right)}$$

Like β, ψ references an abstract state $\hat{\varsigma} = \left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right)$ with abstract execution point $\hat{e}_\varsigma$. Most instructions do not throw exceptions, so ψ is an empty map in these cases:

$$\mathcal{I}(c) = \texttt{Move}(r_d, r_s) \vee \mathcal{I}(c) = \texttt{Const}(r, c) \vee \mathcal{I}(c) = \texttt{Return}(r) \vee \mathcal{I}(c) = \texttt{Add}(r_d, r_s) \vee$$
$$\mathcal{I}(c) = \texttt{NewInstance}(r, \mathit{className}) \vee \mathcal{I}(c) = \texttt{Invoke}(r_1, \ldots, r_n)$$
$$\overline{d\left(\psi\left(\hat{e}_\varsigma\right)\right) = \emptyset}$$

The remaining three instructions use a metafunction $\hat{\mathcal{T}}_\psi : C \times \widehat{\Phi} \times \widehat{S} \times \widehat{K} \times \mathcal{P}\left(\hat{E}\right) \to \emptyset$, which mimics $\hat{\mathcal{T}}_p$. Like $\hat{\mathcal{T}}_p$, $\hat{\mathcal{T}}_\psi$ uses the shorthand $\hat{e}_\mathcal{T} = \hat{e}\left(c, \widehat{SH}\left(\hat{\kappa}, \hat{\sigma}, \{\}\right)\right)$ for the execution point being inspected and $\hat{e}_d = \hat{e}\left(c_h, \widehat{SH}\left(\hat{\kappa}, \hat{\sigma}, \{\}\right)\right)$ for the destination execution point. These cases are exclusive, so $c \notin d\left(\mathcal{H}\right)$ in the last two cases.

$$\hat{\mathcal{T}}_\psi\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \hat{E}_\psi\right) = \begin{cases} \psi\left(\hat{e}_\varsigma\right)\left(\hat{e}_d\right) \stackrel{\sqcup}{:=} \hat{E}_\psi \\ \emptyset \\ \forall \hat{\kappa}_k \in \hat{\sigma}\left(\widehat{ka}\right) : \hat{\mathcal{T}}_\psi\left(c_k, \hat{\phi}_k, \hat{\sigma}, \hat{\kappa}_k, \hat{E}_\psi \cup \hat{e}_\mathcal{T}\right) \end{cases} \begin{array}{l} \mathcal{H}(c) = c_h \\ \hat{\kappa} = \mathbf{halt} \\ \hat{\kappa} = \left(c_\kappa, \hat{\phi}_\kappa, \widehat{ka}\right) \end{array}$$

In the case of the remaining three instructions (`Throw`, `IGet`, and `IPut`), ψ is computed by invoking $\hat{\mathcal{T}}_\psi$ on the state's contents and an empty set: $\hat{\mathcal{T}}_\psi\left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}, \emptyset\right)$.

C. Complexity

The complexity of each analysis is computed with an understanding of the state space, which forms a lattice. Each step in either analysis moves up its lattice. As a result, the asymptotic complexity of the analysis is logarithmic in the size of the state space. These analyses are defined largely without respect to choices of abstract domain. The complexity analysis is first presented generally and then specified using the abstract domains chosen for the empirical study. Both analyses use the state space shown in Figure 8. The specifics of the abstract domain are included in Section V.

Queue size and the complexity of each reduction are both ommitted because both are negligible in practice and because they have a similar effect on both analyses.

In both analyses, some components are shared across states. This sharing is a species of widening [46]. Stores and taint stores are shared globally and the context taint map is shared by states with a common execution point. Because of this widening, the complexity is calculated with respect to the height of the system space lattice, which includes both the state space and the shared components.

The size of the resulting CEK state space is abbreviated: $|\hat{\Sigma}_W| = |C| \cdot |\widehat{\Phi}| \cdot |\widehat{K}|$. The size of the resulting system space is:

$$|Sys| = \left(|\hat{\Sigma}_W| \cdot 2^{|\hat{\Sigma}_W|}\right) \cdot |\widehat{S}| \cdot |\widehat{\mathcal{T}}| \cdot \left(|\hat{E}| \cdot |\widehat{\Omega}|\right).$$

The height of the lattice is logarithmic in the size of the system space:

$$\log|Sys| = \log|\hat{\Sigma}_W| + |\hat{\Sigma}_W| + \log|\widehat{S}| + \log|\widehat{\mathcal{T}}| + \log|\hat{E}| + \log|\widehat{\Omega}|.$$

$$\hat{\varsigma} \in \hat{\Sigma} ::= \left(c, \hat{\phi}, \hat{\sigma}, \hat{\kappa}\right) \mid \mathbf{errorstate} \mid \mathbf{endstate}$$
$c \in C$ is the finite set of code points in the program
$\hat{\phi} \in \widehat{\Phi} = \widehat{Addr} \to \widehat{V}$ is a finite set of frame pointers
$\hat{\sigma} \in \widehat{S} = \widehat{Addr} \to \widehat{V}$
$\hat{v} \in \widehat{V} = \hat{\mathbb{Z}} + \mathcal{P}\left(\widehat{OA}\right) + \mathcal{P}\left(\widehat{K}\right)$
$\hat{z} \in \hat{\mathbb{Z}}$ is a finite set of abstract integers
$\hat{\kappa} \in \widehat{K} ::= \left(c, \hat{\phi}, \widehat{ka}\right) \mid \mathbf{halt}$
$\hat{a} \in \widehat{Addr} ::= \widehat{sa} \mid \widehat{fa} \mid \widehat{ka} \mid \texttt{null}$
$\widehat{sa} \in \widehat{SA} = \widehat{\Phi} \times \mathsf{Register}$
$\widehat{fa} \in \widehat{FA} = \widehat{OA} \times \mathsf{Field}$
$\widehat{oa} \in \widehat{OA}$ is a finite set of object addresses
$\widehat{ka} \in \widehat{KA}$ is a finite set of continuation addresses

Fig. 8: Abstract state space

$$|\widehat{\Phi}| = |C|$$
$$|\widehat{K}| = |C| \cdot |\widehat{\Phi}| \cdot |\widehat{KA}| = O\left(|C|^4\right)$$
$$|\widehat{SA}| = |\widehat{\Phi}| \cdot |\mathsf{Register}| \leq |C|^2$$
$$|\mathsf{Field}| \leq |C|$$
$$|\widehat{FA}| = |\widehat{OA}| \cdot |\mathsf{Field}|$$
$$|\widehat{Addr}| = |\widehat{SA}| + |\mathsf{Field}| + |\widehat{FA}| + |\widehat{KA}| \leq |C|^2 + |C| + |C|^2 + |C|^2 =$$
$$|\widehat{V}| = c + 2^{|C|^2} + 2^{|C|^2} = O\left(2^{|C|^2}\right)$$
$$\log|\widehat{S}| = \log\left(|\widehat{Addr}| \cdot |\widehat{V}|\right) = \log|\widehat{Addr}| + \log|\widehat{V}| = O\left(|C|^2\right)$$
$$\log|\widehat{\mathcal{T}}| = \log\left(|\widehat{Addr}| \cdot 2^{|\widehat{\Theta}|}\right) = \log|\widehat{Addr}| + |\widehat{\Theta}| = O\left(\log|C|^2 + |\widehat{\Theta}|\right)$$
$$|\hat{E}| = |C| \cdot |C|^2 = O\left(|C|^3\right)$$
$$\log|\widehat{\Omega}| = \log|\hat{E}| + |\widehat{\Theta}|.$$

Fig. 9: Calculation of asymptotic complexity

The cardinality of sets of components is determined by the abstractions chosen for them. For example, the bound on the size of execution points relies on maximum stack depth, which is defined by continuation addresses. For the abstraction used in the analysis, $|\widehat{KA}| = |C|^2$. The other upper bounds also follow from the abstractions chosen, as shown in Figure 9.

Using this calculation, the height of the system space lattice is:

$$O\left(\log|C|^6 + |C|^6 + |C|^2 + \log|C|^2 + |\widehat{\Theta}| + \log|C|^3 + \log|C|^3 + |\widehat{\Theta}|\right.$$

The bounds on taint values and on the height of the taint

Application	Total instructions	Live instructions	% live
BattleStat	3460	2117	61.2%
chatterbocs	22146	–	–
ConferenceMaster	34543	3159	9.1%
Filterize	2913	1460	50.1%
ICD9	44820	–	–
keymaster	8985	4594	51.1%
Noiz2	17452	–	–
PassCheck	17588	4911	27.9%
pocketsecretary	8648	5123	59.2%
rLurker	1580	915	57.9%
splunge	136848	–	–
Valet	2864	1462	51.0%

TABLE IV: Total instructions in each application

lattice in the augmented-state analysis are exponential:

$$|\widehat{\Theta}| = |\hat{E}| + |\hat{E}| \cdot 2^{|\hat{E}|^2} = O\left(2^{|C|^6}\right) \text{ and } \log|Sys| = O\left(2^{|C|^6}\right)$$

In contrast, abstract taint values and the height of the system lattice in the *a posteriori* analysis are polynomial. Abstract interpretation dominates the complexity of this analysis:

$$|\widehat{\Theta}| = |\hat{E}| = O\left(|C|^3\right) \text{ and } \log|Sys| = O\left(|C|^6\right).$$

D. Empirical evaluation

Many of the APAC applications attempted to confound analyses by including code that is suspicious but unreachable. As a result, the total number of instructions in these programs is largely meaningless. In this context, the size of each program is more appropriately measured by the number of live instructions it contains. The total instruction count of each application is included in Table IV.

There were a total of seven such flows identified in the 579 in the APAC that were measured: four in the exhaustive count of flows in Filterize and three in the random sampling for chatterbocs.

All analyses used the same configuration, including abstract domains, for abstract interpretation, which seems to be reasonably well optimized for speed and precision. The configuration includes global store widening and no abstract garbage collection. It uses P4F [23] continuation addresses. It uses pointwise allocators for objects (including `String` objects), arrays, and frame pointers; each of these addresses is simply the code point at which it originated. Integers are abstracted to either a number between -1 and 10 inclusively or an ID in the program's layout XML or to $\top$. Other primitive types are abstracted to $\top$. Taint values are sets of execution points that identify the location in the program where they originated.

Analyses were executed in the Fulton Supercomputing Lab at Brigham Young University and timed out after 24 hours. Both β and ψ are generated lazily and memoized. As a result, taint propagation in the implementation iterates over the state space and not over the set of execution points. Iterations over the state space may occur in any arbitrary order; the implementation uses a depth-first order because of empirically improved performance over breadth-first.

E. Equivalence to augmented-state analysis

Because of this existential structure, the proof assumes without loss of generality that nondeterministic behaviors such as search order occur in parallel between the compared analyses.

Appendix E1 presents formalisms that are especially relevant to the proof. Appendix A2 presents a side-by-side comparison of updated augmented-state semantics with those of the *a posteriori* analysis. Appendix E2 contains the proof of equivalence.

1) Selected augmented-state formalisms: In order to reason about the fidelity of the *a posteriori* analysis to the augmented-state analysis, it is necessary to repeat the relevant formalisms from Aldous and Might [2].

a) Implicit taint values: Aldous and Might present implicit taint values as sets of branch-assignment pairs, which suffice only to determine if a context should have a taint: $\widehat{\Theta_I} = \hat{E} \times \hat{E}$. In the more general case, implicit taint values contain a set of branch-assignment pair and an explicit taint value: $\widehat{\Theta_I} = \mathcal{P}\left(\hat{E} \times \hat{E}\right) \times \widehat{\Theta_E}$. The universe of taint values is the union of the (disjoint) universes of explicit and implicit taint values: $\widehat{\Theta} \equiv \widehat{\Theta_E} \cup \widehat{\Theta_I}$. This additional precision allows the analysis to determine the origin of the taint, which is necessary to identify which flows can occur.

The set of branch-assignment pairs is the result of chaining, when an implicit taint value is created because of a branch on a value marked with another implicit taint value. More accurately, the presentation of Aldous and Might avoids the need to group branch-assignment pairs together because the taint store maps to sets of taint values.

This simpler data structure, however, creates an analysis that lacks precision in a crucial way. Consider, for example, the program snippet in Figure 10, where `source` is marked with some explicit taint value $\widehat{\theta_E}$. With the branch on line 2 and the assignment on line 4, the analysis creates an (invalid) implicit taint value. This invalid implicit taint value is resident in the taint store on `x` when the program branches at line 6, so another implicit taint value is created at line 7, when an assignment is performed. In the formulation of Aldous and Might, the taint store maps `y` to $\{(2,4),(2,7),(6,7)\}$. (This section uses line numbers in place of execution points as shorthand.) The first two pairs are invalid, but the last is valid. In effect, the validity of these pairs is combined by disjunction; any of these assignments that occurs in the influence of its branch results in a valid taint.

In contrast, the general data structure with a set of branch-assignment pairs and an explicit taint value groups these sets of taints together, resulting in two implicit taint values: $\left(\{(2,4),(6,7)\},\widehat{\theta_E}\right)$ and $\left(\{(2,7)\},\widehat{\theta_E}\right)$. Because these sets represent chains of branches and assignments, they are more appropriately conjoined. The first implicit taint value is invalid because 4 is outside of the influence of 2. The second is invalid because 7 is outside of the influence of 2. As such, this new formulation correctly avoids a spurious taint on `y`.

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

Fig. 10: A broken chain of branches

Implicit taint values are created whenever an assignment occurs and the context taint map is non-empty (equivalently, implicit taint values are created at every assignment, although the set of values produced may be empty). Whenever an assignment occurs (at some execution point $\hat{e}_a$), a set of implicit taint values $\widehat{\Theta}_\Omega(\hat{e}_a)$ is created. $\widehat{\Theta}_\Omega(\hat{e}_a)$ relies on the function $\widehat{\theta_I} : \hat{E} \times \hat{E} \times \mathcal{P}\left(\widehat{\Theta}\right) \to \widehat{\Theta_I}$:

$$\widehat{\theta_I}\left(\hat{e}_a, \hat{e}_b, \widehat{\theta}\right) \equiv \begin{cases} \left(\{(\hat{e}_b, \hat{e}_a)\}, \widehat{\theta}\right) & \widehat{\theta} \in \widehat{\Theta_E} \\ \left(\{(\hat{e}_b, \hat{e}_a)\} \cup P, \widehat{\theta_E}\right) & \widehat{\theta} = \left(P, \widehat{\theta_E}\right) \end{cases}.$$

P is the set of execution point pairs in an implicit taint value. With $\widehat{\theta_I}$, $\widehat{\Theta}_\Omega(\hat{e}_a)$ is easily defined as $\widehat{\Theta}_\Omega(\hat{e}_a) \equiv \left\{\widehat{\theta_I}\left(\hat{e}_a, \hat{e}_b, \widehat{\theta}\right) \mid \hat{e}_b \in \hat{E}, \widehat{\theta} \in \Xi(\hat{e}_a)(\hat{e}_b)\right\}$.

Validity determines retrospectively whether or not a taint should exist. Recall that the augmented-state analysis keeps all implicit taint values until abstract interpretation has completed. Once abstract interpretation has completed and the execution point graph is formed, invalid taints are removed. Implicit taint validity is defined as a predicate on taint values:

$$\widehat{\nu}\left(\widehat{\theta}\right) \equiv \begin{cases} \mathsf{true} & \widehat{\theta} \in \widehat{\Theta_E} \\ \forall(\hat{e}_b, \hat{e}_a) \in P : \hat{e}_a \in \iota(\hat{e}_b) & \widehat{\theta} = \left(P, \widehat{\theta_E}\right) \end{cases}.$$

b) Context taints: Context taints are created whenever a branch occurs based on a tainted value. Without loss of generality, the branch occurs at $\hat{e}_b$ and $\widehat{\Theta}$ is the set of taints on the branch's condition. Ξ is weakly updated at each successor $\hat{e}'$:

$$\Xi(\hat{e}') \coloneqq \Xi(\hat{e}')\left[\hat{e}_b \overset{\sqcup}{\mapsto} \widehat{\Theta}\right] \equiv \Xi(\hat{e}') \coloneqq \Xi(\hat{e}')\left[\hat{e}_b \mapsto \Xi(\hat{e}')(\hat{e}_b) \sqcup \widehat{\Theta}\right]$$

As is the case with implicit taint values, context taints may be invalid. Invalid taints are removed after abstract interpretation is complete and the execution point graph has been created. A context taint is invalid if either it exists outside the influence of its branch or if its taint value is invalid. The validity $\widehat{\nu_\Omega}$ of a context taint at $\hat{e}$ from a branch that occurred at $\hat{e}_b$ that contains $\widehat{\theta}$ is:

$$\widehat{\nu_\Omega}\left(\hat{e}, \hat{e}_b, \widehat{\theta}\right) \equiv \hat{e} \in \iota(\hat{e}_b) \wedge \widehat{\nu}\left(\widehat{\theta}\right).$$

2) Proof of equivalence:

Theorem 1 (Equivalence of augmented-state analysis to valid-only analysis): The valid information flows identified by the augmented-state analysis are identical to the information flows identified by the valid-only analysis.

Proof 1 (Equivalence of augmented-state analysis to valid-only analysis):

By definition, the valid-only analysis differs only in that it never stores invalid taints. The proof considers only the taint store and context taint map because the CESK components are independent of the taint components. More specifically, explicit taint values are always valid, so only implicit taint values and context taints need consideration.

Implicit taint values are created from context taint. By the definitions of $\widehat{\theta_I}$, $\widehat{\nu}$, and $\widehat{\nu_\Omega}$, any implicit taint value created from invalid context taint is also invalid.

Context taints, in turn, are created from a pair of execution points and a taint value. By the update rule for Ξ and the definition of $\widehat{\nu_\Omega}$, any context taint created from an invalid implicit taint value is invalid.

Per the abstract interpretation semantics, context taints are propagated and modified as they go from one execution point to a successor. The remaining case to be considered is propagation of context taint from one execution point to a successor, which both copies the value and modifies it by changing $\hat{e}_a$ (crucially, $\hat{e}_b$ and $\widehat{\theta}$ are unchanged). In this case, it is possible for a context taint at an execution point $\hat{e}$ outside of the influence of its branch $\hat{e}_b$ ($\hat{e} \notin \iota(\hat{e}_b)$) to propagate to an execution point $\hat{e}'$ inside the influence of the branch ($\hat{e}' \in \iota(\hat{e}_b)$). However, an execution point exists in the influence of the branch only if it is reached along some actual path discovered by the abstract interpreter that does not leave the influence. As such, this valid context taint derived from an invalid context taint is redundant.

Theorem 2: Equivalence of *a posteriori* analysis to valid-only analysis The information flows identified by the *a posteriori* analysis are identical to those identified by the valid-only analysis.

Proof 2: By induction on the semantics given in Appendix A2 and the taint propagation phase given in Section III-C, the taints propagated by the valid-only analysis are identical to the taints that would be propagated if the *a posteriori* taint propagation occurred immediately. Because both state graph exploration and taint propagation happen until a fixed point is reached, order is irrelevant and every information flow that one analysis detects is detected by the other.

Theorem 3: Equivalence of *a posteriori* analysis to augmented-state analysis The information flows identified by the *a posteriori* analysis are identical to those identified by the augmented-state analysis.

Proof 3: By Theorem 1 and Theorem 2.