Static analysis of non-interference in expressive low-level languages

Peter Aldous and Matthew Might

University of Utah, Salt Lake City, USA
{peteya,might}@cs.utah.edu

Abstract. Early work in implicit information flow detection applied only to flat, procedureless languages with structured control-flow (e.g., if statements, while loops). These techniques have yet to be adequately extended and generalized to expressive languages with interprocedural, exceptional and irregular control-flow behavior. We present an implicit information flow analysis suitable for languages with conditional jumps, dynamically dispatched methods, and exceptions. We implement this analysis for the Dalvik bytecode format, the substrate for Android. In order to capture information flows across interprocedural and exceptional boundaries, this analysis uses a projection of a small-step abstract interpreter's rich state graph instead of the control-flow graph typically used for such purposes in weaker linguistic settings. We present a proof of termination-insensitive non-interference. To our knowledge, it is the first analysis capable of proving non-trivial non-interference in a language with this combination of features.

1 Introduction

With increasing awareness of privacy concerns, there is a demand for verification that programs protect private information. Although many systems exist that purport to provide assurances about private data, most systems fail to address implicit information flows, especially in expressive low-level languages. Expressive low level languages, such as Dalvik bytecode, do not provide static guarantees about control-flow due to the presence of dynamic dispatch, exceptions and irregular local jumps.

To demonstrate that a program does not leak information, we prove *non-interference* [14]: any two executions of a *non-interfering* program that vary only in their "high-security" inputs (the values that should not leak) must exhibit the same observable behaviors; in other words, high-security values must never affect observable program behavior. Our analysis proves *termination-insensitive non-interference* [2,11,27].

For its foundation, our analysis uses a rich *static* taint analysis to track high-security values in a program: high-security values are marked as tainted and, as (small-step) abstract interpretation proceeds, any values affected by tainted values are also marked. (Taint analysis is a dynamic technique but an abstract

interpretation of a concrete semantics enriched with taint-tracking makes it a static analysis.)

We enrich the taint analysis to track implicit flows and prove non-interference by executing the advice of Denning and Denning [8]. Their advice suggested that languages *without* statically bounded control-flow structures could precisely track implicit information flows with a static analysis that calculated the immediate postdominator (or immediate forward dominator) of every branch in the program's control-flow graph.[1] We show (and prove) that this old and simple principle fully (and formally) generalizes from flat, procedureless languages to rich, modern languages by constructing projections of abstract state graphs.

1.1 Contributions

This paper presents an analysis for a summarized version of Dalvik bytecode that preserves all essential features: conditional jumps, methods, and exceptional flow. This analysis is similar in spirit to the suggestion of Denning and Denning but uses an *execution point graph* instead of the program's control-flow graph. The execution point graph is formed by projecting the state graph that results from an small-step abstract interpreter [28]. Nodes in the execution point graph contain code points *and* contextual information such as the height of the stack to prevent subtle information leaks. Consider, for example, the function in Figure 1.

There are two possible executions of `leak` when called when `top` is **true** (and when the captured value `printed` is initially set to **false**). In each case, `leak` immediately recurs, this time with `top` set to **false**. At this point, the topmost stack frame has `top` set to **false** and the other stack frame has `top` set to **true**. This leak exploits the difference between these stack frames by returning immediately (effectively setting `top` to **true**) in one case—and then proceeding to print the value of `top`.

```
static void leak(boolean top) {
  if (top) {
    leak(false);
  } else {
    if (sensitive) {
      return;
    }
  }
  if (!printed) {
    printed = true;
    System.out.println(top);
  }
}
```

Fig. 1: A leak after convergence

The execution point graph contains just enough information about the stack to prevent leakages of this variety. During small-step abstract interpretation, an implicit taint is added to any value that changes after control-flow has changed due to a high-security value. Convergence in the execution point graph indicates that control-flow has converged and, consequently, that no further information can be gleaned

[1] A node P *postdominates* another node A in a directed graph if every walk from A to the exit node includes P.

from it about high-security values. This calculation can be done lazily and is fully *a posteriori*; as such, it may be more efficient than performing one abstract interpretation to create the graph and another to perform taint analysis.

An execution point graph whose nodes consist of a code point and the height of the stack is preferable to a more precise graph whose nodes contain full stacks; conflating more execution points means that, in some cases, convergence happens after fewer execution points. As a result, this less precise graph translates into more precise tracking of implicit flows.

Section 2 presents a language that summarizes the features of Dalvik bytecode, gives semantics for the language, and describes the abstraction of this dynamic analysis to a static analysis. Section 3 presents the proof of termination-insensitive non-interference. Further discussion and related work follow.

2 Language and semantics

$$
\begin{aligned}
\mathit{prgm} &\in \mathsf{Program} = \mathsf{ClassDef}^* \\
\mathit{classdef} &\in \mathsf{ClassDef} ::= \texttt{Class}\ \mathit{className}\ \{\mathit{field}_1, \ldots, \mathit{field}_n, m_1, \ldots, m_m\} \\
m &\in \mathsf{Method} ::= \texttt{Def}\ \mathit{mName}\ \{\mathit{handler}_1, \ldots, \mathit{handler}_n, \mathit{stmt}_1, \ldots, \mathit{stmt}_m\} \\
\mathit{handler} &\in \mathsf{Handler} ::= \texttt{Catch}(\mathit{ln}, \mathit{ln}, \mathit{ln}) \\
\mathit{stmt} &\in \mathsf{Stmt} ::= \mathit{ln}\ \texttt{Const}(r, c) \\
&\quad | \ \mathit{ln}\ \texttt{Move}(r, r) \\
&\quad | \ \mathit{ln}\ \texttt{Invoke}(\mathit{mName}, r_1, \ldots, r_n) \\
&\quad | \ \mathit{ln}\ \texttt{Return}(r) \\
&\quad | \ \mathit{ln}\ \texttt{IfEqz}(r, \mathit{ln}) \\
&\quad | \ \mathit{ln}\ \texttt{Add}(r, r, r) \\
&\quad | \ \mathit{ln}\ \texttt{NewInstance}(r, \mathit{className}) \\
&\quad | \ \mathit{ln}\ \texttt{Throw}(r) \\
&\quad | \ \mathit{ln}\ \texttt{IGet}(r, r, \mathit{field}) \\
&\quad | \ \mathit{ln}\ \texttt{IPut}(r, r, \mathit{field}) \\
r &\in \mathsf{Register} = \{\texttt{result}, \texttt{exception}, 0, 1, \ldots\} \\
\mathit{ln} &\in \mathsf{LineNumber} \text{ is a set of line numbers} \\
\mathit{mName} &\in \mathsf{MName} \text{ is a set of method names} \\
\mathit{field} &\in \mathsf{Field} \text{ is a set of field names} \\
\mathit{cp} &\in \mathit{CodePoint} ::= (\mathit{ln}, m)
\end{aligned}
$$

Fig. 2: Abstract syntax

2.1 Syntax

The abstract syntax for our summarized bytecode language is given in Figure 2. See Appendix A for the differences between this language and Dalvik bytecode.

In conjunction with this syntax, we need the following metafunctions:

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

2.2 State space

A state ς contains six members, which are formally defined in Figure 3:

1. A code point cp.
2. A frame pointer ϕ. All registers in the machine are frame-local. Frame addresses are represented as a pair consisting of a frame pointer and an index.
3. A store σ, which is a partial map from addresses to values.
4. A continuation κ.
5. A taint store ts, which is a map from addresses to taint values. It is updated in parallel with the store. Undefined addresses are mapped to the empty set.
6. A context taint set ct, which is a set of execution points where control-flow has diverged before reaching the current state.

2.3 Semantics

Our semantics require projection metafunctions. $\mathit{height} : \mathit{Kont} \to \mathbb{Z}$ calculates stack height and $p_\varsigma : \Sigma \to \mathit{ExecPoint}$ uses height to create execution points.

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

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

The concrete semantics for our language are defined by the relation $(\to) \subseteq \Sigma \times \Sigma$. In its transition rules, we use the following shorthand: ep_ς is a state's

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

Fig. 3: Concrete state space

execution point and itv_ς is the set of implicit taint values generated at a state. For a state ς with context taint set $\{ep_1, \ldots, ep_n\}$,

$$ep_\varsigma = p_\varsigma(\varsigma) \quad \text{and} \quad itv_\varsigma = \left\{\left(ep_1, ep_\varsigma\right), \ldots, \left(ep_n, ep_\varsigma\right)\right\}$$

The `Const` instruction writes a constant value to a register. Observe that implicit taints can be applied to a constant assignment.

$$\frac{\mathcal{I}(cp) = \texttt{Const(}r\texttt{, }c\texttt{)}}{(cp, \phi, \sigma, \kappa, ts, ct) \to (next(cp), \phi, \sigma', \kappa, ts', ct)}, \text{ where}$$
$$sa = (\phi, r)$$
$$\sigma' = \sigma[sa \mapsto c]$$
$$ts' = ts[sa \mapsto itv_\varsigma]$$

The `Move` instruction simulates all of Dalvik bytecode's move instructions.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Move(}r_d\texttt{, }r_s\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \left(next\left(cp\right), \phi, \sigma', \kappa, ts', ct\right)}, \text{ where}$$
$$\begin{aligned} sa_d &= (\phi, r_d) \\ sa_s &= (\phi, r_s) \\ \sigma' &= \sigma[sa_d \mapsto \sigma\left(sa_s\right)] \\ ts' &= ts[sa_d \mapsto ts\left(sa_s\right) \cup itv_\varsigma] \end{aligned}$$

The `Invoke` instruction simulates Dalvik's invoke instructions. Virtual method resolution is deferred to Appendix A. The method's receiver is in register 1.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Invoke(}mName\texttt{, }r_1\texttt{, }\ldots\texttt{, }r_n\texttt{)}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \left(cp', \phi', \sigma', \kappa', ts', ct'\right)}, \text{ where}$$
$$\begin{aligned} cp' &= init\left(\mathcal{M}\left(mName\right)\right) \\ \kappa' &= \mathbf{retk}(cp, \phi, ct, \kappa) \\ \phi' &= \text{ is a fresh frame pointer address} \\ &\quad \text{for each } i \text{ from 1 to } n, \\ sa_{di} &= (\phi', i-1) \text{ and } sa_{si} = (\phi, r_i) \\ \sigma' &= \sigma[sa_{d1} \mapsto \sigma\left(sa_{s1}\right), \ldots, sa_{dn} \mapsto \sigma\left(sa_{sn}\right)] \\ ts' &= ts[sa_{d1} \mapsto ts\left(sa_{s1}\right) \cup itv_\varsigma, \ldots, sa_{dn} \mapsto ts\left(sa_{sn}\right) \cup itv_\varsigma] \\ ct' &= \begin{cases} ct & \text{if } ts\left(sa_{s0}\right) = \emptyset \\ ct \cup \{ep_\varsigma\} & \text{if } ts\left(sa_{s0}\right) \neq \emptyset \end{cases} \end{aligned}$$

The `Return` instruction summarizes Dalvik's return instructions. The `Return` instruction introduces context taint if invocation occurred in a tainted context.

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Return(}r\texttt{)} \quad \kappa = \mathbf{retk}(cp, \phi', ct_k, \kappa')}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \left(next\left(cp'\right), \phi', \sigma', \kappa', ts', ct'\right)}, \text{ where}$$
$$\begin{aligned} sa_d &= (\phi', \texttt{result}) \\ sa_s &= (\phi, r) \\ \sigma' &= \sigma[sa_d \mapsto \sigma\left(sa_s\right)] \\ ts' &= ts[sa_d \mapsto ts\left(sa_s\right) \cup itv_\varsigma] \\ ct' &= \begin{cases} ct & \text{if } ct_k = \emptyset \\ ct \cup \{ep_\varsigma\} & \text{if } ct_k \neq \emptyset \end{cases} \end{aligned}$$
$$\frac{\mathcal{I}\left(cp\right) = \texttt{Return(}r\texttt{)} \quad \kappa = \mathbf{halt}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \rightarrow \mathbf{endstate}}$$

The `IfEqz` instruction jumps to the given target if its argument is 0:

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

$$sa_s = \left(\phi, r\right)$$

$$cp' = \begin{cases} jump\left(cp, ln\right) & \text{if } \sigma\left(sa_s\right) = 0 \\ next\left(cp\right) & \text{if } \sigma\left(sa_s\right) \neq 0 \end{cases}$$

$$ct' = \begin{cases} ct & \text{if } ts\left(sa_s\right) = \emptyset \\ ct \cup \left\{ep_\varsigma\right\} & \text{if } ts\left(sa_s\right) \neq \emptyset \end{cases}$$

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

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

$$sa_d = \left(\phi, r_d\right)$$

$$sa_l = \left(\phi, r_l\right)$$

$$sa_r = \left(\phi, r_r\right)$$

$$\sigma' = \sigma[sa_d \mapsto \sigma\left(sa_l\right) + \sigma\left(sa_r\right)]$$

$$ts' = ts[sa_d \mapsto ts\left(sa_l\right) \cup ts\left(sa_r\right) \cup itv_\varsigma]$$

Object instantiation is done with the `NewInstance` instruction:

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

$$oa \text{ is a fresh object address}$$

$$sa = \left(\phi, r\right)$$

$$\sigma' = \sigma[sa \mapsto oa]$$

$$ts' = ts[sa \mapsto itv_\varsigma]$$

The remaining instructions use an additional metafunction:

$$\mathcal{T} : CodePoint \times FP \times ContextTaint \times Kont \rightharpoonup CodePoint \times FP \times ContextTaint \times Kont$$

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

$$\mathcal{T}(cp, \phi, ct, \kappa) = \begin{cases} (cp_h, \phi, ct, \kappa) & \text{if } \mathcal{H}(cp) = cp_h \\ \mathcal{T}(cp_k, \phi_k, ct, \kappa_k) & \text{if } cp \notin dom(\mathcal{H}) \text{ and } ct_k = \emptyset \\ & \text{and } \kappa = \mathbf{retk}(cp_k, \phi_k, ct_k, \kappa_k) \\ \mathcal{T}(cp_k, \phi_k, ct \cup itv_\varsigma, \kappa_k) & \text{if } cp \notin dom(\mathcal{H}) \text{ and } ct_k \neq \emptyset \\ & \text{and } \kappa = \mathbf{retk}(cp_k, \phi_k, ct_k, \kappa_k) \end{cases}$$

The `Throw` instruction requires two cases. One is for continued execution at the specified handler and one is for an error state when no handler can be found.

$$\frac{\mathcal{I}(cp) = \texttt{Throw(}r\texttt{)} \quad (cp, \phi, ct, \kappa) \in dom(\mathcal{T})}{(cp, \phi, \sigma, \kappa, ts, ct) \rightarrow (cp', \phi', \sigma', \kappa', ts', ct')}, \text{ where}$$
$$\begin{aligned} sa_s &= (\phi, r) \\ sa_d &= (\phi', \texttt{exception}) \\ (cp', \phi', ct', \kappa') &= \mathcal{T}(cp, \phi, ct, \kappa) \\ \sigma' &= \sigma[sa_d \mapsto \sigma(sa_s)] \\ ts' &= ts[sa_d \mapsto ts(sa_s) \cup itv_\varsigma] \end{aligned}$$

$$\frac{\mathcal{I}(cp) = \texttt{Throw(}r\texttt{)} \quad (cp, \phi, ct, \kappa) \notin dom(\mathcal{T})}{(cp, \phi, \sigma, \kappa, ts, ct) \rightarrow \textbf{errorstate}}$$

The `IGet` instruction represents the family of instance accessor instructions in Dalvik bytecode. It requires three transition rules. In the first, the object address is non-null. In the others, the object address is null; the second case shows a handled exception and the third shows a top-level exception.

$$\frac{\mathcal{I}(cp) = \texttt{IGet(}r_d\texttt{, }r_o\texttt{, }field\texttt{)} \quad oa \neq \texttt{null}}{(cp, \phi, \sigma, \kappa, ts, ct) \rightarrow (next(cp), \phi, \sigma', \kappa, ts', ct)}, \text{ where}$$
$$\begin{aligned} sa_d &= (\phi, r_d) \\ sa_o &= (\phi, r_o) \\ oa &= \sigma(sa_o) \\ fa &= (oa, field) \\ \sigma' &= \sigma[sa_d \mapsto \sigma(fa)] \\ ts' &= ts[sa_d \mapsto ts(oa) \cup ts(fa) \cup itv_\varsigma] \end{aligned}$$

$$\frac{\mathcal{I}(cp) = \texttt{IGet}(r_d,\ r_o,\ field) \quad oa = \texttt{null} \quad (cp, \phi, ct, \kappa) \in dom(\mathcal{T})}{(cp, \phi, \sigma, \kappa, ts, ct) \rightarrow (cp', \phi', \sigma', \kappa', ts', ct')}, \text{ where}$$

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

$$\frac{\mathcal{I}(cp) = \texttt{IGet}(r_d,\ r_o,\ field) \quad oa = \texttt{null} \quad (cp, \phi, ct, \kappa) \notin dom(\mathcal{T})}{(cp, \phi, \sigma, \kappa, ts, ct) \rightarrow \textbf{errorstate}}, \text{ where}$$

$$\begin{aligned}
sa_o &= (\phi, r_o) \\
oa_o &= \sigma(sa_o)
\end{aligned}$$

The `IPut` instruction also represents a family of instructions; `IPut` stores values in objects. Like `IGet`, `IPut` requires three transition rules.

$$\frac{\mathcal{I}(cp) = \texttt{IPut}(r_s,\ r_o,\ field) \quad oa \neq \texttt{null}}{(cp, \phi, \sigma, \kappa, ts, ct) \rightarrow (next(cp), \phi, \sigma', \kappa, ts', ct)}, \text{ where}$$

$$\begin{aligned}
sa_s &= (\phi, r_s) \\
sa_o &= (\phi, r_o) \\
oa &= \sigma(sa_o) \\
fa &= (oa, field) \\
\sigma' &= \sigma[fa \mapsto \sigma(sa_s)] \\
ts' &= ts[fa \mapsto ts(sa_s) \cup itv_\varsigma]
\end{aligned}$$

$$\frac{\mathcal{I}(cp) = \texttt{IPut}(r_d,\ r_o,\ field) \quad oa = \texttt{null} \quad (cp, \phi, ct, \kappa) \in dom(\mathcal{T})}{(cp, \phi, \sigma, \kappa, ts, ct) \rightarrow (cp', \phi', \sigma', \kappa', ts', ct')}, \text{ where}$$

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

$$\frac{\mathcal{I}(cp) = \texttt{IPut}(r_d, r_o, field) \quad oa = \texttt{null} \quad (cp, \phi, ct, \kappa) \notin dom(\mathcal{T})}{(cp, \phi, \sigma, \kappa, ts, ct) \rightarrow \textbf{errorstate}}, \text{ where}$$
$$sa_o = (\phi, r_o)$$
$$oa_o = \sigma(sa_o)$$

2.4 Abstraction

A small-step analyzer as described by Van Horn and Might [28] overapproximates program behavior and suits our needs. Abstraction of taint stores and context taint sets is straightforward: they store execution points, which are code points and stack heights. Code points need no abstraction and the height of abstract stacks is suitable. Any abstraction of continuations (even that of PDCFA [9]) admits indeterminate stack heights; an abstract execution point with an indeterminate stack height cannot be a postdominator.

3 Non-interference

3.1 Influence

The influence of an execution point ep_0 is the set of execution points that lie along some path from ep_0 to its immediate postdominator ep_n (where ep_0 appears only at the end) in the execution point graph.

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

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

With P defined, we can define the influence of ep_0 as:

$$\{ep \in V \mid \exists p = \langle ep_0, \ldots, ep_n\rangle \in P \ : \ ep \in p\} - \{ep_0, ep_n\}$$

3.2 Program traces

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

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

3.3 Observable behaviors

Which program behaviors are observable depends on the attack model and is a decision to be made by the user of this analysis. In this proof, we consider the general case: every program behavior is observable. A more realistic model would be that invocations of certain functions are observable, as well as top-level exceptions. Accordingly, we define $obs \subseteq \Sigma$ so that $obs = \Sigma$.

3.4 Valid taints

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

$$valid = ExplicitTV \cup \{(ep_b, ep_a) \in ImplicitTV \mid ep_a \in influence\,(ep_b)\}$$

3.5 Labeled behaviors

A state ς has a *labeled behavior* if and only if it reads values at one or more addresses with valid taint or if it occurs at a state with valid context taint. We define the function $inputs : \Sigma \to Addr^*$ and $labeled : \Sigma \to \mathcal{P}\,(TaintValue)$ so that *inputs* identifies the addresses read by ς's instruction and *labeled* identifies the valid taints at those addresses. The use of itv_ς reflects context taint. The formal definitions of *inputs* and *labeled* are given in Figure 4.

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

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

Fig. 4: A definition of addresses read by instruction. $\varsigma = (cp, \phi, \sigma, \kappa, ts, ct)$

3.6 Similar stores

Two stores are *similar* with respect to two frame pointers, two continuations, and two taint stores if and only if they differ only at reachable addresses that

are tainted in their respective taint stores. This definition requires some related definitions, which follow.

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

1. Both stores are undefined at their respective address, or
2. Either store is tainted at its respective address, or
3. The stores map their respective addresses to the same value, or
4. The stores map respective addresses to structurally identical objects.

Formally,

$$
\begin{gathered}
(\sigma_1, a_1, ts_1) \approx_a (\sigma_2, a_2, ts_2) \\
\Leftrightarrow \\
\left(a_1 \notin dom\,(\sigma_1) \wedge a_2 \notin dom\,(\sigma_2)\right) \vee \qquad (1) \\
\left(\exists\ tv \in ts_1\,(a_1)\ :\ tv \in valid\ \vee\ \exists\ tv \in ts_2\,(a_2)\ :\ tv \in valid\right) \vee \qquad (2) \\
\sigma_1\,(a_1) = \sigma_2\,(a_2)\ \vee \qquad (3) \\
\left(\sigma_1\,(a_1) = oa_1\ \wedge\ \sigma_2\,(a_2) = oa_2\ \wedge \qquad (4)\right. \\
\left.\forall\ field \in \mathsf{Field}, (\sigma_1, (oa_1, field)\,, ts_1) \approx_a (\sigma_2, (oa_2, field)\,, ts_2)\right)
\end{gathered}
$$

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

Formally,

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

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

1. The stores are similar with respect to the given pair of frame pointers, and
2. They are recursively similar with respect to the given continuations.

Formally,

$$
\begin{gathered}
(\sigma_1, \phi_1, \kappa_1, ts_1) \approx_\sigma (\sigma_2, \phi_2, \kappa_2, ts_2) \\
\Leftrightarrow \\
(\sigma_1, \phi_1, ts_1) \approx_\phi (\sigma_2, \phi_2, ts_2) \wedge \qquad (1) \\
\left(\kappa_1 = \kappa_2 = \mathbf{halt}\ \vee \qquad (2)\right. \\
\kappa_1 = \mathbf{retk}(cp_1, \phi_1', ct_1', \kappa_1')\ \wedge \kappa_2 = \mathbf{retk}(cp_2, \phi_2', ct_2', \kappa_2')\ \wedge \\
\left.(\sigma_1, \phi_1', \kappa_1', ts_1) \approx_\sigma (\sigma_2, \phi_2', \kappa_2', ts_2)\right)
\end{gathered}
$$

3.7 Similar states

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

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

3.8 Similar traces

Two traces $\pi = \langle \varsigma_1, \varsigma_2, \ldots, \varsigma_n \rangle$ and $\pi' = \langle \varsigma'_1, \varsigma'_2, \ldots, \varsigma'_m \rangle$ are *similar* if and only if their observable behaviors are identical except for those marked as tainted. We formulate the similarity of traces using a partial function $dual : \pi \rightharpoonup \pi'$. Similarity of traces is equivalent to the existence of a function *dual* such that:

1. *dual* is injective, and
2. *dual* maps each state in π to a similar state in π' if such a state exists, and
3. All states in π not paired by *dual* occur in a tainted context, and
4. All states in π' not paired by *dual* occur in a tainted context, and
5. The pairs of similar states occur in the same order in their respective traces.

Formally,

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

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

$$\forall\ \varsigma_i \in dom(dual),\ \varsigma_i \approx_\varsigma dual(\varsigma_i) \wedge \tag{2}$$

$$\forall\ \varsigma_i \notin dom(dual),\ itv_\varsigma \in valid\ \wedge \tag{3}$$

$$\forall\ \varsigma'_j \notin range(dual),\ itv_\varsigma \in valid\ \wedge \tag{4}$$

$$\forall\ i, j \in 1 \ldots n\ :\ dual(\varsigma_i) = \varsigma'_k \wedge dual(\varsigma_j) = \varsigma'_l, \tag{5}$$

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

3.9 Transitivity of similarity

Lemma If two states ς and ς' are similar and if their execution point's immediate postdominator in the execution point graph is ep_{pd}, the first successor of each state whose execution point is ep_{pd} is similar to the other successor.

Formally, if $\varsigma \approx_\varsigma \varsigma'$ and
$\varsigma \to \varsigma_1 \to \ldots \to \varsigma_n$ and $\varsigma' \to \varsigma'_1 \to \ldots \to \varsigma'_m$ and
$p_\varsigma(\varsigma) = p_\varsigma(\varsigma') = ep_0$ and $p_\varsigma(\varsigma_n) = p_\varsigma(\varsigma'_m) = ep_{pd}$ and
ep_{pd} is the immediate postdominator of ep_0, and
$\forall\ \varsigma_i \in \{\varsigma_1, \ldots, \varsigma_{n-1}\} \cup \{\varsigma'_1, \ldots, \varsigma'_{m-1}\},\ p_\varsigma(\varsigma_i) \neq ep_{pd}$, then $\varsigma_n \approx_\varsigma \varsigma'_m$

Proof Without loss of generality,

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

We refer to $\varsigma_1, \ldots, \varsigma_{n-1}$ and $\varsigma'_1, \ldots, \varsigma'_{m-1}$ as *intermediate states*.
It is given that $p_\varsigma(\varsigma_n) = p_\varsigma(\varsigma'_m)$. All that remains is to prove that

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

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

1. Addresses added to σ in some intermediate state,
2. Addresses added to σ' in some intermediate state,
3. Addresses changed in σ in some intermediate state,
4. Addresses changed in σ' in some intermediate state.

$$\forall\, a \in Addr,$$
$$a \notin dom(\sigma) \wedge a \in dom(\sigma_n) \Rightarrow ts_n(a) \cap valid \neq \emptyset \quad (1)$$
$$a \notin dom(\sigma') \wedge a \in dom(\sigma_m) \Rightarrow ts_m(a) \cap valid \neq \emptyset \quad (2)$$
$$\sigma(a) \neq \sigma_n(a) \Rightarrow ts_n(a) \cap valid \neq \emptyset \quad (3)$$
$$\sigma'(a) \neq \sigma_m(a) \Rightarrow ts_m(a) \cap valid \neq \emptyset \quad (4)$$

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

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

$$\forall\, r \in \mathsf{Register}, \phi_f \in FP_f,\ (\phi_f, r) \notin dom(\sigma) \cup dom(\sigma')$$

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

$$\phi_n \in FP_f \vee \phi_m \in FP_f \Rightarrow (\sigma_n, \phi_n, ts_n) \approx_\phi (\sigma_m, \phi_m, ts_m)$$

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

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

3.10 Global transitivity of similarity

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

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

Proof By induction on transitivity of similarity.

3.11 Labeled interference in similar states

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

$$\begin{gathered}
\varsigma_1 \approx_\varsigma \varsigma_2 \Rightarrow \mathit{labeled}(\varsigma_1) \neq \emptyset \vee \mathit{labeled}(\varsigma_2) \neq \emptyset \vee \\
\forall i \in \langle 1, \ldots n \rangle, (\sigma_1, a_i, ts_1) \approx_a (\sigma_2, a'_i, ts_2), \text{ where} \\
\mathit{inputs}(\varsigma_1) = \langle a_1, \ldots, a_n \rangle \text{ and } \mathit{inputs}(\varsigma_2) = \langle a'_1, \ldots, a'_n \rangle \text{ and} \\
\varsigma_1 = (cp_1, \phi_1, \sigma_1, \kappa_1, ts_1, ct_1) \text{ and } \varsigma_2 = (cp_2, \phi_2, \sigma_2, \kappa_2, ts_2, ct_2)
\end{gathered}$$

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

3.12 Concrete termination-insensitive labeled interference

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

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

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

Observe that because the choice of traces is arbitrary, π' is also examined.

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

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

3.13 Abstract non-interference

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

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

4 Discussion

Our analysis proves termination-insensitive non-interference, which allows divergence leaks; postdominance only considers paths that reach the exit node, so it excludes infinite paths. With termination analysis (a well understood technique), this analysis could prove non-interference without qualification. Side channel attacks, such as timing attacks, are beyond the scope of this paper.

It is possible that precision could be improved with less precise execution points. This would require a weaker definition of state similarity, such as similarity with respect to frame pointers but not to those in the continuation stacks.

Both the precision and complexity of this analysis depend on those of the abstract interpreter chosen. Imprecisions inherent to the choice of abstractions create false positives. For example, an abstract interpreter might simulate multiple branches executing when only one is possible—and, accordingly, would add unnecessary taint to values assigned during execution of those branches. Additionally, convergence admits some overtainting; different branches could assign identical values to a register. In this case, taint would be assigned unnecessarily. Accordingly, other improvements to precision may be possible.

5 Related work

Sabelfeld and Myers [26] summarize the early work on information flows.

Denning [7] introduces the idea of taint values as lattices instead of booleans. Denning and Denning [8] describe a static analysis on a simple imperative language and discuss how it could be applied to a language with conditional jumps. They do not discuss how it could be applied to a language with procedures and exceptional flow. Volpano, et al. [30] validate the claims of Denning and Denning for languages with structured control-flow. Volpano and Smith [31] then extend it to handle loop termination leaks and some exceptional flow leaks.

Chang and Streiff [5] present a compiler-level tool that transforms untrusted C programs into C programs that enforce specified policies. Kim et al. [19] perform an abstract interpretation on Android programs. Arzt et al. [1] present FlowDroid, a static analyzer for Android applications. All of these papers limit their analyses to explicit information flows although the FlowDroid project does claim in a blog post to have support for implicit information flows.

Xu et al. [32] perform a source-to-source transformation on C programs to instrument them for taint tracking and track one class of implicit information flows. Kang et al. [18] perform a dynamic analysis called DTA++ that operates on Windows x86 binaries and tracks information flows. DTA++ explicitly allows for false negatives in order to minimize false positives. Liang and Might [21] present a Scheme-like core calculus for scripting languages like Python. Their core language is expressive enough to contain not only function calls but also call/cc as a primitive but do not detect implicit information flows.

Giacobazzi and Mastroeni [13] demonstrate an abstract interpreter on programs in a simple imperative language that lacks functions and exceptional control-flow—the kind of language that the technique suggested by Denning and Denning [8] addresses. Askarov, et al. [2] also noninterference in a Jif-like language with syntactic boundaries on its control-flow constructs and that lacks functions and exceptional control-flow. Liu and Milanova [22] perform a static analysis on Java programs that tracks implicit information flows. Their analysis does as Denning and Denning [8] suggested; it calculates postdominance to determine the extent of a conditional statement's effect on control-flow. However, they do not present a grammar, prove non-interference, or discuss exceptional control-flow. Pottier and Simonet [25] present a type system that guarantees noninterference in an ML-like language. Their technique relies on syntactic structures. Barthe and Rezk [3] perform an analysis on Java bytecode but assume a postdominance calculation, even for exceptional control-flow. They also assume type annotations for functions. Cousot and Radia [12] discuss non-interference but do not discuss interprocedural flows.

Cavallaro, et al. [4] dismiss the effectiveness of static techniques. They then discuss the shortcomings of dynamic analyses, particularly against intentionally malicious code. Moore, et al. [23] present a type system that, with runtime enforcement and a termination oracle, guarantees progress-sensitive noninterference (also called termination-sensitive noninterference). TaintDroid [10] is a dynamic extension to Android's runtime environment. Being a dynamic analysis, it does not purport to identify all possible program behaviors.

Venkatakrishnan, et al. [29] perform a static pre-pass that adds tracking instructions to inform a dynamic analysis. This analysis preserves termination-insensitive noninterference but ignores exceptional control-flow. Jia et al. [17] present a system that dynamically enforces annotations, including security labels and declassification. Myers [24] created JFlow, an extension to Java that allows programmers to annotate values and that uses a type system with both static and dynamic enforcement. It does not guarantee non-interference.

Liang, et al. [20] introduce *entry-point saturation* to properly model Android programs, which use several entry points instead of one. Entry-point saturation injects repeatedly into all entry points until encountering a fixed point and would allow the analysis in this paper to be applied to full Android programs. Van Horn and Might [28] demonstrated that abstract interpreters can be constructed automatically from concrete interpreters. Earl, et al. [9] demonstrated an abstract interpretation that operates in a pushdown automaton.

The official specifications for the bytecode language [15] and the dex file format [16] provide detailed information about Dalvik bytecode.

6 Conclusion

As we claimed, Denning and Denning's principle does generalize and extend to expressive low-level languages such as Dalvik bytecode. The key twist was to extend postdominance from control-flow graphs to interprocedural execution point graphs, and to extract these graphs as projections from small-step abstract interpretations over concrete semantics bearing taint-tracking machinery.

Acknowledgements 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. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. 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.
2. A. Askarov, S. Hunt, A. Sabelfeld, and D. 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.
3. G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *Proceedings of the 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '05, pages 103–112, New York, NY, USA, Jan. 2005. ACM.
4. L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In D. Zamboni, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 143–163. Springer Berlin Heidelberg, 2008.
5. W. Chang, B. Streiff, and C. 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.

6. U. Combinator research group. Tapas: Dalvik bytecode analysis in Scala. https://github.com/Ucombinator/Tapas, 2014.
7. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
8. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
9. C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 177–188, New York, NY, USA, 2012. ACM.
10. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. pages 1–6, 2010.
11. J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, 1974.
12. S. Genaim and F. Spoto. Information flow analysis for java bytecode. In R. Cousot, editor, *Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer Berlin Heidelberg, 2005.
13. R. Giacobazzi and I. 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.
14. J. A. Goguen and J. Meseguer. Security policies and security models. In *2012 IEEE Symposium on Security and Privacy*, pages 11–11. IEEE Computer Society, 1982.
15. Google. Bytecode for the Dalvik VM. http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html, 2014.
16. Google. Dalvik executable format. http://source.android.com/devices/tech/dalvik/dex-format.html, 2014.
17. L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake. Run-time enforcement of information-flow properties on android. In J. Crampton, S. Jajodia, and K. Mayes, editors, *Computer Security – ESORICS 2013*, volume 8134 of *Lecture Notes in Computer Science*, pages 775–792. Springer Berlin Heidelberg, 2013.
18. M. G. Kang, S. McCamant, P. Poosankam, and D. 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, Feb. 2011.
19. J. Kim, Y. Yoon, K. Yi, and J. Shin. Scandal: Static analyzer for detecting privacy leaks in android applications. Mobile Security Technologies, 2012.
20. S. Liang, A. W. Keep, M. Might, S. Lyde, T. Gilray, P. Aldous, and D. 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.
21. S. Liang and M. Might. Hash-flow taint analysis of higher-order programs. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, PLAS '12, pages 8:1–8:12, New York, NY, USA, 2012. ACM.
22. Y. Liu and A. Milanova. Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In *Software Main-*

tenance and Reengineering (CSMR), 2010 14th European Conference on, pages 146–155, Mar. 2010.
23. S. Moore, A. Askarov, and S. 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.
24. A. C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 228–241, New York, NY, USA, 1999. ACM.
25. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(1):117–158, Jan. 2003.
26. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Sept. 2006.
27. A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. In S. D. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 40–58. Springer Berlin Heidelberg, 1999.
28. D. Van Horn and M. 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.
29. V. N. Venkatakrishnan, W. Xu, D. C. DuVarney, and R. Sekar. Provably correct runtime enforcement of non-interference properties. In P. Ning, S. Qing, and N. Li, editors, *Information and Communications Security*, volume 4307 of *Lecture Notes in Computer Science*, pages 332–351. Springer Berlin Heidelberg, 2006.
30. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2–3):167–187, Jan. 1996.
31. D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 156–168, June 1997.
32. W. Xu, S. 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.

A Scaling up to full Dalvik bytecode

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

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

The Android SDK allocates registers compactly, creating spurious information flows. To remedy this, our interpreter performs a liveness analysis and separates registers into use-define chains.

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

Dalvik bytecode differentiates between methods and "encoded methods". Invocation instructions have a method ID, which is an index into an array of methods. Each method has a prototype, a name, and a class ID used to identify the class that defines the method. Each encoded method includes a method id, flags such as private or final, and an offset into the code table. From that offset, it is possible to ascertain the instructions, register count, and exception handling information for a method. Arguments, including the receiver, if there is one, are placed at the end of the invocation frame in order and registers are zero-indexed. A method with a register count of six that takes two (single-register) arguments and a receiver expects the receiver to be in virtual register 3 and the two arguments to be in registers 4 and 5.

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

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

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

Our computational model is not concurrent. As such, we do not support monitor instructions.

Line numbers are not part of the Dalvik bytecode specification; the bytecode uses offsets into code. Most disassemblers create labels for legibility; we use zero-indexed line numbers instead, as we use the parser from the Tapas [6] project, which also uses them. Code points in our interpreter for full Dalvik bytecode are represented as pairs of a line number and an encoded method. Additionally, there are special singleton objects for the end position, which is reached upon invocation of the halt continuation, and an error position.