A posteriori taint-tracking for demonstrating non-interference in expressive low-level languages

Peter Aldous
University of Utah
peteya@cs.utah.edu

Matthew Might
University of Utah
might@cs.utah.edu

February 3, 2016

Abstract

We previously presented a theory of analysis for expressive low-level languages that is capable of proving non-interference for expressive languages. We now present an implementation of that analysis, empirical evaluation of that implementation and optimizations to improve performance. In the course of developing an optimization, we provide a independence result for the taint-flow analysis that drives tracking of information. In particular, we show that the taint-tracking can be derived from the results of a taint-free analysis. In addition to improving performance, this independence broadens the applicability of the underlying approach to information-flow analysis. Finally, we present metrics for our analysis on a suite of applications which demonstrate improvements in its performance.

1 Introduction

In our SAS 2015 paper [1], we presented a theory of analysis suitable for proving non-interference (or the absence of information flows) in expressive low-level languages, such as Dalvik bytecode [11]. Dalvik bytecode, like many modern low-level languages, contain objects, virtual methods, exceptional flow, conditional jumps, and mutation. A proof of non-interference could be used to demonstrate that user data, such as passwords or GPS location, are not transmitted to third parties. It could also be used to verify that a cryptographic primitive does not leak its key or to verify that sandboxed applications cannot communicate with each other.

This analysis is distinct from analyses that focus on identification of bugs; it is successful not when it helps analysts to identify and fix problems but when it helps analysts to prove the absence of problems. In order to help analysts to make guarantees about programs, this automated analysis eliminates false negatives (and, consequently, permits false positives).

Our theory of analysis, as presented, had no implementation or performance metrics. We left the question of tractability open. In order to address this question, we present an implementation (named Data) of our analysis. We discuss the details of its implementation and test it on Android applications. We then discuss optimizations to the analysis that make it tractable on an increasing number of these applications.

1.1 Contributions

- We present an implementation of an information-flow analysis for Android applications.
- We discuss optimizations made to the analysis and present performance metrics from executions of the analysis on several Android applications.
- We demonstrate that the analysis generates false positives but that it is sufficiently accurate to help analysts to find information flows and prove their absence.

1.2 Structure

Section 2 describes the techniques upon which our analysis is built as they exist in the literature. Section 3 elaborates on the implementation of Data, including its optimizations to the analysis as presented. Section 4 measures the performance of the different variants of the analysis. Section 5 elaborates on the significance of the results. Section 6 discusses related work.

2 Background

Our theory of analysis uses a small-step abstract interpreter with components added to prove non-interference. Subsection 2.1 describes small-step abstract interpretation and subsection 2.2 explains non-interference.

2.1 Small-step abstract interpretation

The CESK [9] evaluation model represents states in an interpreter's execution as tuples of control (C), environment (E), store (S), and continuations (K). The control represents where the interpreter is in the program, the environment maps variables to addresses, the store maps addresses to values, and a continuation contains the information used to return from a function. In an imperative program, these terms roughly approximate (respectively) the program counter, the frame pointer, the heap, and the stack.

Van Horn and Might [25] demonstrated that CESK interpreters can be turned into **small step abstract interpreters** by abstracting their state spaces so that they can be guaranteed to be finite and modifying their transition rules to permit for multiple successors to each state. An abstract CESK interpreter produces not a linear program trace but a graph of abstract states that model all possible executions from a given initial abstract state.

A finite state space is not sufficient to guarantee termination. It is also necessary to avoid calculating successors for the same state repeatedly. Since the successors to a state are a function of the members of the state (and, in some cases, immutable values such as the program being interpreted), the successors of a state will always be the same and it is never necessary to repeat a calculation. As such, small-step abstract interpreters can be guaranteed to terminate. Instead of interpreting until there are no more successors, interpretation proceeds, ignoring already-explored states, until no new states can be found. The analysis terminates with a finite graph of all reachable abstract states.

A sound small-step abstract interpretation represents all possible program executions in its finite state space. Typically, soundness is proven by proving simulation. Simulation proofs show that abstract interpretation simulates concrete interpretation by showing that the relationship between a concrete state ς and its abstraction $\hat{\varsigma}$ holds for their respective successors. With a concretization function γ, we can formalize the relationship between ς and $\hat{\varsigma}$: ς is in $\gamma(\hat{\varsigma})$. If ς' is the concrete successor to ς and $\hat{\varsigma}'$ is some abstract successor to $\hat{\varsigma}$, then ς' must be in $\gamma(\hat{\varsigma}')$. Given some initial concrete state ς_0 and an initial abstract state $\hat{\varsigma}_0$ whose concretization includes ς_0 and a proof of this inductive property, we can conclude that the abstract state graph includes all possible behaviors that the concrete trace could exhibit.

More formally, our inductive property states that if

$$\varsigma \to \varsigma' \text{ and } \varsigma \in \gamma(\hat{\varsigma})$$

then there exists an abstract state $\hat{\varsigma}'$ such that

$$\hat{\varsigma} \rightsquigarrow \hat{\varsigma}' \text{ and } \varsigma' \in \gamma(\hat{\varsigma}')$$

2.2 Non-interference

Traditional **taint tracking** mechanisms apply a security type or label, also called a taint, to sensitive values, such as a phone's location or a user's password. Whenever a new value is written, it derives its security type from the values upon which it depends. Although many security types are binary, where values with a high security label are sensitive and must be protected and values low security label are not sensitive, Denning demonstrated that security types may be rich lattices [5].

These techniques are effective for **explicit** information flows but fail to detect **implicit** information flows, which depend on control flow to leak information. Consider the Java code snippet in Listing 1, which demonstrates an implicit flow; despite the fact that y is written with constants, it will always get the same value as x. Traditional taint tracking mechanisms track only explicit flows and fail to catch implicit flows.

In order to track implicit information flows, taints can also be applied to the program's context, per Denning and Denning [6]. Denning and Denning claimed

Listing 1: An implicit information flow

```
if (x == true) {
  y = true;
} else {
  y = false;
}
```

that a static analysis of postdominance in the control flow graph would allow context tainting to apply to languages with arbitrary `goto` statements but did not prove non-interference. Furthermore, their analysis does not include function calls or exceptional control flow. We demonstrated that such an analysis is possible [1], although it was necessary to modify the analysis to properly handle these rich language features.

In order to demonstrate the absence of information flows, even in the presence of exceptional control flow and other rich language features, we proved **non-interference**, which is the property that sensitive information cannot affect (or interfere with) behaviors that are visible to an attacker. Since some programs do not satisfy the requirement of non-interference, the proof of non-interference is a proof that any interference will be identified by the small-step abstract interpreter.

Our analysis uses the suggestion of Denning and Denning to calculate post-dominance, but they use a richer graph (called the **execution point graph**) than the control flow graph. Nodes in an execution point graph are pairs of a code point and a natural number, which is the depth of the stack. As we demonstrated, a subtle class of information leak does exist (in languages with functions) that could elude an analysis that uses a control flow graph to detect implicit flows.

We performed abstract interpretation with taint flow analysis and propagate taints from context. After abstract interpretation we calculated the execution point graph by projecting the state graph. Then, we used the execution point graph to demonstrate that some statements occur at points in the program unaffected by certain branch statements. In these cases, the taints in question can be removed or ignored without allowing interference.

3 Implementation

Our analysis extends the state space of an abstract CESK machine to include a taint store, which acts in parallel with the store but stores taints instead of values, and a context taint set, which tracks the points in the program where branches occurred that depended on tainted values. These modifications allowed us to prove non-interference but added considerable complexity to abstract interpretation.

The addition of the taint store makes no difference in the asymptotic complexity of abstract interpretation (assuming the size of the set of taint values is constant), as the taint store is essentially a small extension to the store. There can be no more taint values than abstract values.

The context taint set, on the other hand, can add considerable complexity. Unlike the taint store, the context taint set is a fully fledged additional component to each state, which expands the size of the state space and can greatly expand the size of the state graph for a particular program. Worse still, widening the store has no effect the context taint set and there is no obvious way to widen the context taint set separately.

Subsection 3.1 details the salient features of the implementation. Subsection 3.2 addresses the expansion of the state graph size due to context taint by removing taint components from the state space and performing taint flow analysis after abstract interpretation.

3.1 Implementation details

3.1.1 Modularity

A canonical abstract interpreter in the style of Abstracting Abstract Machines [25] has one allocation function. Since Data operates on Dalvik bytecode, it needs abstract allocators for frame pointers, continuation addresses, array addresses, and object addresses. Additionally, Dalvik bytecode has four primitive types: int, long, float, and double and each primitive type requires abstraction. In the spirit of the work of Jenkins, et al. [13], using concrete allocators and concrete primitive types makes Data a concrete interpreter.

Allocators for different components of the interpreter are all defined as abstract classes so that they can be exchanged. Continuation addresses can be allocated with a 0CFA allocator or a 1CFA allocator. Other allocators can easily be added to the source code.

Primitive type abstractors are also abstract classes. Implementations exist for each primitive type that represent concrete values, sets of concrete values, and the set of all values. Additional abstractions can easily be written, although it is common for multiple abstraction types to co-exist and it may be necessary to augment the existing abstractors so they can perform arithmetic operations with the additional abstractors.

Naturally, any combination of these allocators and abstractors can be chosen with command-line parameters.

3.1.2 Register de-allocation

Although Dalvik bytecode uses virtual registers, the Android SDK allocates registers sparingly. Register allocation algorithms use called liveness analysis to determine which registers must co-exist and which registers can be combined.

Liveness analysis goes backwards through the program. When a variable is read, it is “live”; in other words, it has been read. When it is written, it is “dead”; portions of the program above its assignment cannot see the result of this assignment. Any two variables that are live at the same time are said to

Listing 2: A sample program for register allocation

```
x = 3;                    // {}
y = x;                    // {x}
z = 4;                    // {y}
x = 5;                    // {y, z}
System.out.println(y);    // {y, z}
System.out.println(z);    // {z}
```

interfere (this is not to be confused with non-interference as used by information flow). Variables that interfere with each other cannot share the same register.

We illustrate liveness analysis with a simple program snippet in Listing 2. The last statement reads z, so our live set is $\{z\}$. The previous statement adds y to our live set: $\{y, z\}$. Writing to x does nothing, as x is not in the live set. The statement prior writes to z, so z is removed from the set and y is left alone: $\{y\}$. The previous statement "kills" y by writing it and "generates" x by reading it, so the live set is $\{x\}$. Finally, the first statement kills x and the live set is left empty.

Since x and z are never in the live set together, they can be combined into a single register during allocation. If they are combined, y must be allocated to a different register because the live set includes both y and z at the second-to-last line of code.

Uses and definitions of variables can be combined into use-def chains. In the example in Listing 2, x would be put into two separate chains. The first includes its definition on the first line and the other includes its definition before the `println` statements. Separating distinct uses of variables enables further optimizations; the different use-def chains for x can be allocated independently.

Our analyzer performs liveness analysis, but in reverse: it creates use-def chains of registers and uses these chains to create addresses for stack values in the abstract store. By separating registers into use-def chains, we prevent values from merging in the abstract store that would otherwise be conflated.

3.1.3 Ancillary components

In order to discover entry points, it is necessary to make sense of Android's binary XML formats. Each Android application has a manifest file called `AndroidManifest.xml` that contains a list of its entry points. It also contains a resource table in a file called `resources.arsc`, which in turn identifies different layout files. Layout files contain descriptions of the various UI components, which in turn define additional entry points; the press of a button or the entry of text into a field may be necessary in order to complete some information flows. Each of these files comes in Android's binary format, which is created and read by copying C structs to and from memory.

In order to ensure that all program behaviors are analyzed in the absence of a main function, Data uses **entry-point saturation**, which was first presented by Liang, et al. [17]

3.2 Optimization

3.2.1 A posteriori taint tracking

The context taint set in our original analysis grows monotonically. We removed context taints in postprocessing, after calculating the entire state graph.

In our new analysis, we move all of the information flow calculations to postprocessing. Since any additional steps taken during abstract interpretation do not affect state graph exploration, separating abstract interpretation and information flow analysis allows known modifications to small-step abstract interpretation, such as store widening, to be used to their full effect.

In order to preserve generality, the abstract interpreter must keep track of the relationships between addresses read and written at each state. This bookkeeping is done outside of the state space and allows the abstract interpreter to use arbitrary allocators, as described by Might and Manolios [19], without making it impossible for the taint tracking mechanism to correctly identify where information flows based on the state graph alone.

Since the entire state graph (and, therefore, the entire execution point graph) is available to the information flow analysis, it is not necessary to propagate context taints from their origins to the end of the program and remove or ignore taints later. Instead, the taint tracking mechanism can avail itself of the fully-formed execution point graph to calculate which assignments require taint from context.

The state graph, together with the annotations about addresses read and written at each state, contain enough information to reconstruct all of the program's behaviors. As such, it is possible to construct the same information flows as in the original model. Propagation of taints uses a fixed point algorithm, much like graph exploration. At each state, taints at addresses read propagate to the addresses they influence. Similarly, taints are added to addresses from context taint whenever writes occur. Taint propagation operates in the same style as in our original analysis except that no spurious context taints are propagated.

The propagation algorithm must also track the context taints seen at code points that invoke functions in order to faithfully recreate context taint when removing frames from the stack (as happens upon returning from a function or throwing an exception).

Much like state exploration, where a state may have multiple successors, taint propagation can proceed to multiple successor states.

Like small-step abstract interpretation, our taint tracking algorithm proceeds to a fixed point; when the propagation reaches a state it has seen previously, when its taint store is subsumed by the taints previously seen at that state, when the taints in the context have all been seen at the state's execution point, and when the context taints in the stack are subsumed by context taints previously seen on the stack, further propagation will generate no new information.

3.2.2 Non-interference in a posteriori taint tracking

The proof of non-interference is essentially the same for a posteriori taint tracking method as in our original theory of analysis. It hinges on the same notion of similarity and proceeds along the state graph as before.

Traces in this proof are series of concrete states, as in the original formulation. Each trace follows some path through the (already explored) state graph. As in the original proof, taint propagation may traverse fewer states than are members of its corresponding concrete trace because the fixed point algorithm may prove that no additional information can be found. As before, this does not invalidate soundness or non-interference.

4 Experimental results

4.1 Methodology

The Automated Program Analysis for Cybersecurity (APAC) program performed several experiments in which analysts were presented with Android applications and were tasked with identifying malware that had been implanted in the applications. We tested both the implementation of the analysis from Aldous and Might and the implementation of a posteriori taint tracking on each application from one of these experiments. Leaks that depend on the inner workings of library methods were considered out of scope.

In order to measure accuracy, we paired the occurrences of sources and of sinks in the apps. Since these pairings number in the millions, it was impossible to verify each one by hand. Instead, we counted the pairings in Filterize, the application with the smallest number of sources and sinks used (16 sources and 19 sinks for a total of 304 pairings, with four of those flows relying on library methods and therefore being out of bounds). Of the twelve applications chosen, six timed out for all analyses.

4.2 Results

Figure 1 shows the time each analysis took, the number of states discovered by each analysis, and the number of lines of code covered by abstract interpretation. Applications timed out after 24 hours. Blank cells indicate that the analysis timed out on the application in question. Figure 2 details the accuracy of the analysis on Filterize.

The data show that the separation of abstract interpretation from taint tracking improves the tractability of the analysis, allowing it to successfully analyze six applications instead of four, and to analyze the programs whose analyses were tractable more quickly.

As its theoretical basis suggests, there are no false negatives in the analysis. Wherever a source can reach any sink, the analysis reports that it can reach all of the sinks that can be reached. Overtainting is unsurprising, since the abstract

App name	Runtime		States found		LOCs covered	
	in state	a posteriori	in state	a posteriori	in state	a posteriori
BattleStat	233 s	139 s	221	190	188	188
chatterbocs	3034 s	399 s	1067	499	447	447
Filterize	–	80238 s	–	6844	–	1375
Filterize (1CFA)	–	27917 s	–	3944	–	1255
ICD9	2086 s	918 s	593	539	412	415
rLurker	1917 s	636 s	706	609	572	572
Valet	–	43822 s	–	2382	–	1254

Table 1: Performance statistics for analyses on APAC apps

	True		False	
Analysis	positives	negatives	positives	negatives
0CFA	4	190	106	0
1CFA	4	224	72	0

Table 2: Accuracy of analysis of Filterize

interpretation performed was 0CFA; 0CFA merges control flow upon returning from a function.

Even 0CFA, with its lack of precision, succeeds in demonstrating that there are no information flows in 190 (62.5%) of the 304 source/sink pairs in Filterize. This alleviates more than half of the workload for a human analyst charged with proving the absence of information flows.

Since merging seemed to be a significant factor in the precision of the analysis on Filterize, we ran 1CFA on Filterize. 1CFA was more precise, so it covered many fewer lines of code and discovered many fewer states. As a result, it was significantly faster in a posteriori taint tracking (it timed out in in-state taint tracking with both allocators). Its tracking of information flows was also more precise; 1CFA demonstrated that there were no information flows in 224 (73.7%) of the source/sink pairs in Filterize.

5 Discussion

5.1 Assisted assurance

Unlike many analyses, Data is not designed to identify bugs or vulnerabilities. Instead, it is designed to help experts to prove their absence. As such, its utility is not usefully measured by comparisons against tools that identify bugs. It is useful, rather, when it decreases the amount of work that must be done to verify the absence of information flows.

5.2 Analysis-agnostic non-interference

The proof of non-interference requires a sound state graph and, since allocators may be non-deterministic, information about the addresses written and read at each state (or, for a less precise analysis, at each code point in the program or even at any point in the program). As mentioned previously, this separation allows store widening to be effective. However, the separation of these two analyses is powerful enough to permit much more than widening, as the proof of non-interference is agnostic to the style of analysis performed. As a result, a more precise analysis such as PDCFA [7] could be performed on a program and the non-interference proof would still apply. The same is true of optimizations, such as abstract garbage collection [21].

Our proof of non-interference, like that in our original theory of analysis, relies on induction on the structure of Dalvik bytecode. It relies particularly on the fact that Dalvik's instructions allow only four stack operations: no change, a single push, a single pop, and an arbitrary number of pops (for thrown exceptions). However, it may be possible to generalize the proof to any language as long as the language uses only combinations of pushes and pops in its semantics. Even analyses of languages with call/cc could prove non-interference with this technique.

Additionally, if the bookkeeping during analysis were expanded to include stack behaviors, including each execution point visited when searching for an exception handler, it might be possible to perform taint tracking on any sound state graph for any analysis on any language and get a proof of non-interference without additional theoretical work.

5.3 Generalized state graph postprocessing

We originally created an analysis by extending the state space of a CESK machine with data that do not affect the execution of the program. Subsequently, we improved its performance drastically by removing the additional information from the state space and calculating it from a finished state graph.

When designing future analyses based on small-step abstract interpreters, it is likely that it will be similarly practical to perform the analysis after abstract interpretation rather than extending the state space. For example, it is likely that this same technique could be applied to abstract counting [20].

6 Related work

Our analysis is an implementation of the analysis presented by Aldous and Might [1]. It builds upon prior work in small-step abstract interpretation, such as the seminal work by Van Horn and Might [25] and uses entry-point saturation [17]. Sabelfeld and Myers [24] present a succinct summary of the concepts in information flow tracking.

Denning [5] and, later, Denning and Denning [6] pioneered work in taint tracking.

Subsequent papers applied and expanded on their work, such as the work by Volpano, Irvine, and Smith [27] that established that the analysis presented by Denning and Denning is sound. Volpano and Smith [28] extend the work of Denning and Denning so it handles some termination leaks and some exceptional flow. Both of these papers use languages without conditional jumps, making it impossible to apply them to Dalvik bytecode or similar languages without significant adaptation.

Many analyses are suitable for finding information flows but not for proving their absence. Kim et al. [16] analyze Dalvik bytecode but do not address implicit flows. Chang, Streiff, and Lin present a tool that transforms C programs by enforcing policies but also address only explicit flows. Arzt et al. [2] present FlowDroid, an analyzer for Android applications. Their paper states that they do not address implicits, although a blog post claims that they have since added support for them.

Xu, Bhaktar, and Sekar [29] transform C programs in a way that tracks information flows. They track some implicits but not all of them. Accordingly, they do not claim or prove non-interference. Kang, et al. [15] perform an analysis on Windows x86 binaries but specifically allow for false negatives. Liang and Might [18] analyze Python programs and detect explicit information flows.

Venkatakrishnan, et al. [26] prove non-interference with a dynamic analysis that can terminate a program before it leaks information. Their analysis targets a language similar to Jif; it is a simple imperative language that uses security annotations and that lacks exceptional control flow. TaintDroid [8] is a dynamic extension to Android that tracks explicit flows. Significantly, TaintDroid tracks information in memory and in storage. It does not prevent information leaks but may detect them. Jia, et al. [14] present a dynamic analysis that allows programmers to provide annotations that are enforced dynamically. Myers [23] created JFlow, which extends Java and allows programmers to provide annotations which it enforces. JFlow is a hybrid of static and dynamic analyses and relies on branches with syntactic bounds. It permits several species of covert flows.

There are analyses that guarantee non-interference. In every case, the analysis targets a language without features common to expressive low-level languages and, as such, cannot guarantee non-interference on programs written in these languages. Giacobazzi and Mastroeni [10] prove non-interference in IMP, which lacks exceptional control flow, including the ability to break from a loop. Askarov, et al. [3] and Moore, Askarov, and Chong [22] prove non-interference in Jif, a Java-like language with security annotations that lacks exceptional control flow. Barthe and Rezk [4] prove non-interference in a language modeled after the JVM that lacks function calls. As such, exceptions are reduced to jumps and their analysis cannot be applied to languages with functions and exceptional flow.

The specification for Dalvik bytecode [11] and for the dex file format [12] provide details of the languages and their semantics.

7 Conclusion

Our original theory of analysis is theoretically sound. Our implementation shows that, with some optimizations, it is effective at diminishing the work that an analyst must do to prove the absence of information flows in moderately sized Android applications.

Our analysis also demonstrates two possible theoretical results: it may be possible to completely separate the proof of non-interference from the language being analyzed and from the analysis performed, allowing any analysis to add trivial bookkeeping and a postprocessing step in order to produce a proof of non-interference. It may also be possible to perform other refinements, such as abstract counting, to small-step abstract interpretation a posteriori, thus yielding the same optimizations while shrinking the state space.

Additionally, we have shown that any sound state graph, combined with information about addresses written and read during the course of execution, can be used to prove non-interference without the need for additional theoretical work. Other analyses may also be performed a posteriori.

References

[1] 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.

[2] 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.

[3] 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.

[4] Gilles Barthe and Tamara 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, January 2005. ACM.

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

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

[7] Christopher Earl, Ilya Sergey, Matthew Might, and David 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] Maria Jenkins, Leif Andersen, Thomas Gilray, and Matthew Might. Concrete and abstract interpretation: Better together. In *Proceedings of the 2014 Workshop on Scheme and Functional Programming*, 2014.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 13–25, New York, NY, USA, 2006. ACM.

[21] Matthew Might and Olin Shivers. Improving flow analyses via γCFA: abstract garbage collection and counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 13–25, New York, NY, USA, September 2006. ACM.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.