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

Anonymized for blind review[1]

1 Anonymized for blind review

Abstract

Recent work by Aldous and Might presented a theory of analysis for expressive low-level languages that is capable of proving non-interference for expressive languages. We provide an implementation of that analysis with performance metrics. We also 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.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Abstract interpretation, information flow

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

In recent work [1], Aldous and Might 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 [16]. Dalvik bytecode, like many modern low-level languages, contains objects, virtual methods, exceptional flow, conditional jumps, and mutation. Non-interference means that one datum does not affect another, so 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 problems but when it helps analysts to prove the absence of problems. In order to help analysts make guarantees about programs, this automated analysis eliminates false negatives and, consequently, permits false positives. Although the efficacy of a bug-finding analysis can be measured by the (hopefully low) number of false positives produced by the analysis, the efficacy of this analysis is measured more accurately by the (hopefully high) number of true negatives it produces.

In the process of implementing and optimizing this analysis, we discovered that it is possible to separate it into two distinct phases: small-step abstract interpretation (described in Section 2.1) and taint tracking. In this *a posteriori* taint tracking, there is no need to include a taint store or context taint set in the states. We will refer to the original theory of analysis and its implementation as the augmented-state analysis.

We tested both implementations on actual Android applications. Although the augmented-state analysis proved intractably slow, the *a posteriori* analysis performed well. Although the analysis is exponential in the worst case, it yielded precise results for the majority of programs in our test suite.

After a discussion (Section 2) of the techniques upon which our analysis is built, we discuss the details of our optimization (Section 3) and its implications. We demonstrate its

$$\frac{\mathcal{I}(cp) = \texttt{Move}(r_d, r_s)}{(cp, \phi, \sigma, \kappa) \Rightarrow (next(cp), \phi, \sigma', \kappa)}, \text{ where}$$
$$sa_d = (\phi, r_d)$$
$$sa_s = (\phi, r_s)$$
$$\sigma' = \sigma[sa_d \mapsto \sigma(sa_s)]$$

Figure 1 Semantics for a move instruction in a concrete interpreter

effectiveness on a test suite of Android applications in Section 4. Section 5 elaborates on the significance of the results. Finally, Section 6 discusses the large body of related work.

This paper demonstrates the feasibility of and expands upon the ideas briefly outlined in a published extended abstract [2].

2 Background

Our analysis uses a small-step abstract interpreter to prove non-interference. Section 2.1 describes small-step abstract interpretation and Section 2.2 explains non-interference.

2.1 Small-step abstract interpretation

The CESK [11] evaluation model represents states in an interpreter's execution as tuples of control (C), environment (E), store (S), and continuations (K). Each component serves to capture part of the state of an interpreter at a moment during execution. The control represents where the interpreter is in the program. The environment maps variables to addresses and the store maps addresses to values; this indirection simplifies interpretation in the presence of mutation. The continuation contains the information used to return from a function. In an imperative program, these terms roughly approximate (respectively) the program counter, the frame pointer, the heap, and the stack.

Van Horn and Might [34] 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 by modifying their transition rules to permit for multiple successors to each state. An abstract CESK interpreter does not produce a linear program trace, but instead produces a graph of abstract states that model all possible executions from a given initial abstract state.

As their name suggests, small-step abstract interpreters proceed in small steps. Their semantics are derived from those of analogous small-step concrete interpreters. The `Move` instruction is a summary of the various instructions in Dalvik bytecode that copy a datum from one register to another. Figure 1 demonstrates the behavior of the `Move` instruction in a small-step concrete interpreter. It makes use of a statement lookup metafunction $\mathcal{I}$, a metafunction that retrieves the next code point *next*, and an appropriately defined state space. As the semantics demonstrate, control passes from cp to the following instruction $next(cp)$, the frame pointer ϕ is unchanged, the store σ is updated so that the address for the destination register r_d contains the value stored at the address for the source register r_s, and the continuation κ is unchanged. Addresses for registers are pairs of a frame pointer and a register name.

An analogous rule specifies behavior for the `Move` instruction in a small-step abstract interpreter in Figure 2. In order to guarantee termination in abstract interpretation, abstract

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

$$\widehat{sa}_d = \left(\hat{\phi}, r_d\right)$$

$$\widehat{sa}_s = \left(\hat{\phi}, r_s\right)$$

$$\hat{\sigma}' = \hat{\sigma}[\widehat{sa}_d \mapsto \hat{\sigma}\left(\widehat{sa}_d\right) \sqcup \hat{\sigma}\left(\widehat{sa}_s\right)]$$

Figure 2 Semantics for a move instruction in an abstract interpreter

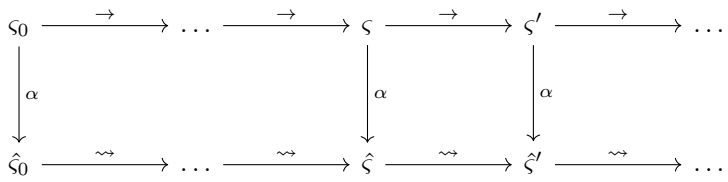

Figure 3 An illustration of the simulation property

states represent many (potentially, infinitely many) concrete states. In small-step abstract interpretation, this is done by abstracting the components of each state. For example, an abstract integer might be the set $\{+\}$, which represents all positive integers. Other state components, such as frame pointers and addresses, are abstracted with allocators. Unlike a fresh concrete components, new abstract components may already be in use because they come from finite sets of possible components. For example, the set of abstract frame pointers might be defined as the set of program locations. New frame pointers are generated upon function invocation, so new frame pointers could be generated with the code point where the function is invoked. With this allocation strategy, recursive calls to the same function would receive the same abstract frame pointer. This phenomenon is called *merging*.

The possibility of merging means that abstract interpretation is, in one sense, incorrect: it may demonstrate behavior that cannot happen in a concrete interpretation. It is, however, correct in another sense: it models all possible behaviors in all corresponding concrete interpretations. This correctness property is called *soundness*. Typically, soundness is proven via simulation (illustrated in Figure 3). Simulation proofs show that abstract interpretation simulates concrete interpretation by showing that the relationship between a concrete state ς and its abstraction $\hat{\varsigma}$ holds for their respective successors. We can formalize the relationship between ς and $\hat{\varsigma}$ with a abstraction relation α, as follows: $\alpha\left(\varsigma, \hat{\varsigma}\right)$. We also define the concrete and abstract succession relations: $\rightarrow\left(\varsigma, \varsigma'\right)$ and $\rightsquigarrow\left(\hat{\varsigma}, \hat{\varsigma}'\right)$. With these relations, simulation can be defined: If ς' is the concrete successor to ς, there must be some abstract successor $\hat{\varsigma}'$ to $\hat{\varsigma}$ such that $\alpha\left(\varsigma', \hat{\varsigma}'\right)$. Given some initial concrete state ς_0 and an initial abstract state $\hat{\varsigma}_0$ whose abstraction includes ς_0 and a proof of this inductive property, we can conclude that the abstract state graph includes all possible behaviors that the concrete trace could exhibit.

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

```
private boolean secret;

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

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

Figure 4 An information leak through the stack

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. Denning demonstrated that security types may be lattices [8]. In practice, many security types are binary: the "high" label applies to values with sensitive information and the "low" label applies to other values.

These techniques are effective for **explicit** information flows, in which values propagate directly (*e.g.*, via assignment). However, they fail to detect **implicit** information flows, which depend on control flow to leak information. For example, different constants might be assigned to a variable in the true and false branches of an if statement. More subtly, switch statements, function calls, function returns, and exceptional control flow can change control flow. These control flows can also change values in ways not detectable by traditional taint tracking mechanisms.

In order to track implicit information flows, taints can also be applied to the program's context, per Denning and Denning [9], which claims that a static analysis of postdominance in the control flow graph would allow context tainting to apply to languages with arbitrary goto statements. However, they did not prove non-interference. Furthermore, their analysis does not include function calls or exceptional control flow. Aldous and Might demonstrated that these common language features allow for subtle information leaks that evade detection even by this postdominance calculation [1].

Figure 4 contains a program that successfully leaks a bit without detection by post-dominance in the control flow graph. It does so by creating a situation where a return is conditioned on a sensitive value. As a result, control flow reaches a particular program location regardless of the sensitive value but the level of the stack reflects that value. As such, it can behave differently without detection.

Aldous and Might use a program's **execution point graph** to track information flows

$$\frac{\mathcal{I}\left(cp\right) = \texttt{Move}(r_d,\ r_s) \qquad ct = \{exec_1, \ldots, exec_n\}}{\left(cp, \phi, \sigma, \kappa, ts, ct\right) \Rightarrow \left(next\left(cp\right), \phi, \sigma', \kappa, ts', ct\right)}, \text{ where}$$

$$\begin{aligned} sa_d &= (\phi, r_d) \\ sa_s &= (\phi, r_s) \\ \sigma' &= \sigma[sa_d \mapsto \sigma\left(sa_s\right)] \\ itv_\varsigma &= \{(exec_1, exec_\varsigma), \ldots, (exec_n, exec_\varsigma)\} \\ ts' &= ts[sa_d \mapsto ts\left(sa_s\right) \cup itv_\varsigma] \end{aligned}$$

Figure 5 Semantics for a move instruction with taint tracking in a concrete interpreter

in the context of functions, *etc.* Each **execution point**, or node in the execution point graph, contains a program location and a natural number that represents the height of the stack. An abstract execution point has either a natural number or a value that represents an indeterminate stack height, which can occur in any finite abstraction of a state space. Abstract execution points without exact stack heights are not considered postdominators, as they represent multiple concrete execution points.

The production rule for the `Move` instruction with taint tracking is in Figure 5. It uses $exec_\varsigma$ as shorthand for the state's execution point. The taint store moves taints from sa_s to sa_d. Additionally, it adds taints from context. Implicit taints include the execution point from the context taint set, which represents the program context where the branch occurred, and the execution point at which assignment occurred. Whenever execution from the branching execution point must pass through its postdominator in the execution point graph before reaching the assigning execution point, the context taint may be safely ignored. This is necessary for the augmented-state analysis, which cannot avail itself of the execution point graph until abstract interpretation has completed. Abstraction of the additional components is straightforward.

In order to demonstrate the absence of information flows, even in the presence of exceptional control flow and other rich language features, Aldous and Might proved **non-interference**, which is the property that sensitive information cannot affect (or interfere with) behaviors that are visible to an attacker. More specifically, they proved **termination-insensitive non-interference**, a weaker property that ignores leakage via non-termination. Termination analysis is well understood and beyond the scope of this work. In the spirit of termination-insensitive non-interference, exceptional control flow that propagates to the top level is not included in the execution point postdominance calculation. 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.

3 Implementation

Correctly implementing an analysis for Dalvik bytecode requires attention to minute details. Making this analysis fast and precise requires other modifications. This section describes some of the salient details of our work. Section 3.4 describes the implementation details particular to the *a posteriori* analysis.

3.1 Modularity

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

There is also considerable flexibility in the allocators for frame pointers, arrays, objects, and continuation addresses and in the abstractions for primitive values. Since `String` objects are treated in some ways like primitives (*e.g.*, there are literal strings but no other literal objects), a separate allocator can be specified for Strings than for other objects. Our analyzer currently supports four styles of abstraction for primitives and more can be added by creating a subclass of the appropriate class; for example, a new abstraction for integers could be created with a new subclass of `IntRegister`. Primitives can be abstracted to bottom (a value that represents any value of its respective type), abstracted to bottom unless they lie within a predefined range, not abstracted (that is, with perfect precision; this choice makes divergence possible), or not abstracted until a predefined number of values have been abstracted and to bottom thereafter. If the store were adapted to allow for strong updates instead of weak updates, it could change from an abstract interpreter to a concrete interpreter by changing its runtime configuration.

3.2 Register de-allocation

The Android SDK coalesces virtual registers during compilation, which leads to an imprecise analysis. In order to address this problem, we reversed the process. We performed a liveness analysis to determine which virtual registers could be separated. In a sense, what we did was to perform a single static assignment transformation on the program, except that we did not use phi nodes. Instead, where a phi node might have been inserted, we simply did not separate the register.

Rather than actually transform the source, we used use-def chains instead of registers in frame addresses. This change has the effect of reversing much of the imprecision caused by the register coalescing done at compile time. Because it distinguishes registers by where they are accessed in the program, it also reverses much of the imprecision in frame addresses caused by global store widening.

3.3 Preparation

In Java, addresses in memory must be written before they are read. This invariant is ensured by a process called *preparation* by the Java virtual machine specification [32]. Preparation precedes initialization and writes default values to instance and class members. Default values are either `null` or `0`, depending on the type of the member. In Dalvik bytecode, per the Dalvik virtual machine specification [17], `null` is represented with the number 0, so preparation writes zeros to all members. In Dalvik bytecode, which is a register-based machine instead of a stack-based machine, default values are written to local variables in a similar way to preparation for class and instance members.

Preparation leads to imprecision in the abstract interpreter because it updates weakly; that is, updates add to a set of values instead of overwriting values. In order to ameliorate this imprecision, we identified cases in which the default value must be overwritten before it

is ever read. In these cases, it is safe to omit preparation. For example, when a new object is created, its constructor is usually called in the next instruction. In this case, we can prepare the object by retroactively writing zero to every member not written upon returning from the constructor. As long as every execution does so, all values that may be unwritten in any execution of the constructor are retroactively prepared.

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

3.4 *A posteriori* taint tracking

In the augmented-state theory of analysis, abstract interpretation includes additional components in each state. These components allow taint flow analysis, including contextual taint flows, to occur in tandem with abstract interpretation. The execution point graph is calculated after abstract interpretation by projecting the state graph. Only then can it be demonstrated that some statements occur at points in the program unaffected by certain branch statements. Having proven that these statements happen independently of the statements that added taint to the context, these taints can be ignored without compromising the proof of non-interference.

As an optimization, we separate taint tracking from abstract interpretation. Without the need to track taints, our abstract state space does not require the additional components. Once the state graph is fully generated, we perform the taint flow analysis over the state graph.

Although there are no additional state components, we modified our abstract interpreter to track the addresses read and written at each state. This allowed us to preserve generality in the case of nondeterministic allocators, as Might and Manolios [27] showed to be possible. In the presence of a nondeterministic allocator, it is impossible to calculate in retrospect which addresses were used by a particular state. As such, our abstract interpreter tracks reads and writes at each state in a data structure external to the state space. This bookkeeping made it possible for us to preserve generality in the abstract interpreter. In fact, any allocator can be used, which means that any form of polyvariance can be employed during abstract interpretation [14].

In the case of the `Move` instruction, abstract interpretation proceeds as in Figure 2. In addition, the interpreter records that $\widehat{sa}_d$ is influenced by $\widehat{sa}_s$ at this state.

Performing information flow analysis after completing abstract interpretation means that the information flow analysis has access to the entire state graph (and, therefore, the entire execution point graph). With the execution point graph, context taints can be pruned at each step of the analysis, preventing spurious context taints from requiring additional computation. It also prevents the loss of precision that can happen when a state's execution point changes; a state may have a determinate stack depth and then the creation of a continuation could change it to something indeterminate. While not an issue for soundness, it is potentially disastrous for precision.

The state graph, together with the annotations about addresses that are read and written at each state, contain enough information to reconstruct all of the program's behaviors. As such, it is possible to construct the same information flows as in the augmented-state model (except that some spurious context taints do not occur). This propagation occurs at an abstract state $\hat{\varsigma}$ with an abstract taint store $\widehat{ts}$ and an abstract context taint set $\widehat{ct}$. This is identical to the in-state taint tracking except that the abstract taint store and abstract context taint set have been lifted out of the state. Updates to $\widehat{ts}$ and $\widehat{ct}$ propagate to the

successors of $\hat{\varsigma}$, just as in the in-state taint propagation. With the execution point graph fully populated, updates to $\widehat{ct}$ can be pruned at each step, preventing the spurious propagation of taint flow and speeding up the analysis.

The asymptotic complexity of this *a posteriori* analysis is identical to that of the augmented-state analysis. However, there are improvements to the complexity in parts of the analysis. For example, the state space is much smaller, so the calculation of the execution point graph (quadratic in the size of the state graph) is less complex. In addition, the removal of spurious context taints could have a similar effect on runtime as does abstract garbage collection [28]; by removing spurious flows, it may improve performance in a way not predicted by asymptotic complexity.

Taint propagation proceeds by flattening the state space into a sequence. We found a depth-first ordering to be fastest. Iterations over this sequence proceed until no changes are made either to the taint store, which is global, or to the map of context taints, where each execution point has a unique context taint.

The proof of non-interference is the same for our *a posteriori* taint tracking method as for the augmented-state analysis and hinges on the same notion of similarity.

4 Empirical results

4.1 Methodology

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

In order to compare the efficiency of the *a posteriori* analysis to that of the augmented-state analysis, we implemented both, carefully ensuring that the semantics of both analyses were identical and that any optimizations applied to one were also applied to the other, except where these optimizations were not applicable.

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

Analyses were executed on our server, which has 12 cores and 64 GB of RAM and which runs MacOS 10.8.5, Scala 2.11.7, and Java 7. We chose to limit ourselves to four analyses at a time in an attempt to ensure that there would be no contention for hardware resources.

Scala set operations happen in a nondeterministic order. It is possible to force them to happen deterministically by sorting, but this approach can sometimes mask bugs. Instead of sorting our sets, we repeated our *a posteriori* analyses three times each. The augmented-state analyses were run once each, as they all timed out. We checked the results of the repeated analyses to ensure that identical numbers of states were discovered and identical numbers of

Application	states	instructions	AI time (s)	total time (s)
BattleStat	3951	2117	43.9	365.7
chatterbocs	–	–	–	–
ConferenceMaster	8926	3150	1023.3	1226.3
Filterize	3405	1460	13.9	26.6
ICD9	–	–	–	–
keymaster	13708	4574	28736.7	30442.7
Noiz2	–	–	–	–
PassCheck	10865	4911	365.3	503.5
pocketsecretary	48962	5421	69123.7	72316.7
rLurker	1105	915	9.5	22.8
splunge	–	–	–	–
Valet	1791	1445	20.6	42.5

Figure 6 Space and time measurements

instructions were covered. We also checked the flows reported by each repeated analysis for consistency. Section 5.2 discusses in more detail the situations in which nondeterminism can occur and the effects it can have on the results of the analysis.

All analyses used the same configuration, which seems to be reasonably well optimized for speed and precision. The configuration includes global store widening, no abstract garbage collection, Pushdown CFA for free (P4F) [15] continuation address allocation, a pointwise allocator for objects (including `String` objects), an abstraction for integers that maintains perfect precision for integers that occur as IDs in layout XML or that lie between the range of -1 and 10, and abstractions for the other primitive types that have no precision. Analyses timed out after 24 hours and each had 3 GB of RAM available to it. Taints are sets of labels that identify the location in the program where they originated.

4.2 Results

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

Figure 6 shows the space and time metrics of our *a posteriori* analysis on our test suite. It lists the number of states found and instructions covered by abstract interpretation, the average time spent performing abstract interpretation (excluding initialization, parsing, and information flow), and the average time spent on the entire analysis.

Figure 7 shows the precision metrics of our analysis on our test suite. It shows the number of measured true positives (possible information flows that were correctly identified by the analysis), false positives (impossible information flows incorrectly identified), true negatives (impossible information flows not identified), and false negatives (possible information flows that were missed). There were no false negatives in our measurement. This figure also shows the false positives and true negatives as percentages of the flows measured.

Figure 8 shows the size of each application in bytecode instructions. It also includes coverage information for each application whose analysis completed. The last column displays the percentage of total instructions covered by the analysis. The fact that not all instructions are covered indicates that there is dead code in the programs; as such, the effective size of the program in some respects may be significantly smaller than the apparent size of the program.

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

Figure 7 Correctness and precision metrics: true positives (TP), false positives (FP), *etc.*

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

Figure 8 Total instructions in each application

5 Discussion

5.1 Guarantees

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

On all of the small- and moderate-sized applications in our test suite, the majority of these pairings (as measured in the sample) were demonstrated to be safe, excluding the possibility of termination leaks and leaks that rely on knowledge of the semantics of library code. It is possible to warn the analyst in these cases. This analysis does not save time on all applications and does not eliminate all of the work a human analyst would have to do, but the results suggest that it does eliminate the majority of the work a human analyst would be required to do without it.

5.2 Nondeterminism

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

One way in which it can occur is the use of stateful primitive abstractions. Choosing not to abstract the first n integers encountered and then to abstract the remainder to bottom is sound. However, different sets of integers might be abstracted to bottom in different executions. This could cause the analysis to sometimes explore a branch that cannot be reached by any concrete execution.

5.3 Generalized state graph postprocessing

In a sense, the augmented-state approach to analysis is more intuitive. However, it proved to be inefficient. When designing future analyses based on small-step abstract interpreters, it is likely that it will be similarly practical to perform the analysis after abstract interpretation rather than extending the state space. For example, it is likely that this same technique could be applied to abstract counting [28].

5.4 Analysis-agnostic non-interference

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

6 Related work

Our analysis is an implementation and optimization of the augmented-state analysis [1]. It builds upon prior work in small-step abstract interpretation, such as the seminal work by Van Horn and Might [34] and uses **entry-point saturation** [24]. Entry-point saturation allows an analysis to operate on a program without a main function, as is the case for Android programs. It relies on monotonicity of the analysis and injects into all entry points until it can discover no new information. Sabelfeld and Myers [33] present a succinct summary of the concepts in information flow tracking.

Denning [8] and, later, Denning and Denning [9] pioneered work in taint tracking. Subsequent work by Volpano, Irvine, and Smith [36] continued along the same vein. These authors' work focuses on simple languages that lack function calls and exceptional control flow.

Many analyses are suitable for finding information flows but not for proving their absence. Kim *et al.* [23] and Arzt et al. [3] both analyze Dalvik bytecode but do not address implicit flows. Xu, Bhatkar, and Sekar [37] perform a source-to-source transformation on C programs that tracks explicit information flows and some, but not all, implicits. Kang *et al.* [22] perform a dynamic analysis on x86 binaries that identifies some implicit information flows. Liang and Might [25] perform a static analysis on a rich core calculus of modern scripting languages but track only explicit flows. Enck *et al.* [10] perform a dynamic analysis on Android programs that identifies only explicit information flows.

Systems that require programmer collaboration are useful but do not apply to software that third parties wish to verify. Jia *et al.* [21] perform a programmer-assisted dynamic analysis that guarantees compliance with Android permissions semantics but makes no effort to trace information beyond identifying intents. It proves non-interference in a sense but does not solve the problem that this work addresses. Myers [31] presents JFlow, which uses programmer annotations to perform a hybrid analysis on Java programs. This analysis relies on the static structures of the Java language not present in Dalvik bytecode and does not guarantee non-interference. Heule, et al. [18] perform a dynamic analysis on a rich language with a lambda construct. It requires users to install and use a library, which forces sandboxed components of the program to interact via an MPI-like communication mechanism. Buiras, Vytiniotis, and Russo [7] present a Haskell library that allows programmers to use a hybrid analysis to prove non-interference in their programs.

Many analyses exist that guarantee non-interference but that target languages without important features. As such, these analyses cannot be applied directly to Dalvik bytecode. For example, Venkatakrishnan, et al. [35] prove non-interference in a language that lacks any jumps or exceptional flow, including return statements. Giacobazzi and Mastroeni [13] demonstrate non-interference in a simple imperative language. Askarov, et al. [4] discuss different variants of non-interference, presenting them in terms of a simple language that lacks functions and exceptional flow. Moore, Askarov, and Chong [30] discuss a stricter form of non-interference in another simple imperative language.

Barthe and Rezk [5] discuss non-interference in a JVM-like language. They talk about using a control dependence analysis, as in Denning and Denning [9], in an incrementally presented analysis. When they add functions, they explain that they do not prove non-interference and suggest that it might be proven mechanically.

Mohr, Graf, and Hecker [29] briefly describe an analysis for Android applications but do not provide sufficient details to reproduce their analysis, nor do they provide a proof of non-interference.

Bello, Hedin, and Sabelfeld [6] discuss precision improvements to information flow analyses.

Lourenço and Caires [26] present a type system that allows them to guarantee non-inteference in a language based on the lambda calculus. They do not present an empirical evaluation of their type system's speed or precision. It would require considerable engineering to apply their type system to a language like Dalvik bytecode.

Ferrante, Ottenstein, and Warren [12] present the program dependence graph, which combines control flow and data flow into a single dependence graph. They further describe the idea of program slicing to identify which statements influence the values of variables at program locations. Horwitz, Prins, and Reps [19] showed that program dependence graphs are sufficiently rich to identify classes of equivalent programs; that is, programs that produce different results have different program dependence graphs. As regards the detection of information flows, this work is essentially equivalent to that of Denning and Denning (although the program dependence graph may be a more convenient representation). In particular, program dependence graphs are assembled using control flow graphs and so are susceptible to the class of information flow leakage demonstrated in section 2.2. Also, it is not clear how a program dependence graph could be constructed in the presence of functions and exceptional flow. As such, existing techniques for program slicing in the context of program dependence graphs do not address the problems inherent to analysis of expressive languages.

Jenkins *et al.* [20] pioneered interpreters that operate in both the concrete and the abstract. Gilray, Adams, and Might [14] showed that all forms of polyvariance in abstract interpretation can be achieved through the use of allocators. Gilray *et al.* [15] devised Pushdown CFA for free.

The specification for Dalvik bytecode [16], for the dex file format [17], and for the Java virtual machine [32] provide details of the languages and their semantics.

7 Conclusion

The augmented-state theory of analysis is theoretically sound but admits optimization. In particular, separating taint tracking from abstract interpretation improves its performance dramatically, making it tractable on moderate-sized Android applications. This optimized implementation demonstrates that the analysis is useful on real-world programs: for a majority of the applications in our test suite, it returned results within a reasonable period of time. For each of these applications, the analysis showed that no information flows exist for a majority of the possible pairings of sources and sinks.

It may be possible to further generalize the analysis, making the second phase of analysis completely agnostic to the target language and even to the style of analysis performed in the first phase.

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. Other analyses may also be performed *a posteriori*.

7.0.0.1 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. 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. Peter Aldous and Matthew Might. A posteriori taint-tracking for demonstrating non-interference in expressive low-level languages. In *LangSec Workshop at IEEE Security & Privacy*, 2016.
3. 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.
4. 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.
5. 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.
6. Luciano Bello, Daniel Hedin, and Andrei Sabelfeld. Value sensitivity and observable abstract values for information flow control. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 63–78. Springer, 2015.
7. Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. HLIO: mixing static and dynamic typing for information-flow control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 289–301, New York, NY, USA, August 2015. ACM.
8. Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.
9. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.
10. 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.
11. 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.
12. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.
13. 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.
14. Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: a unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 407–420, New York, NY, USA, September 2016. ACM.
15. Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, POPL '16, pages 691–704, New York, NY, USA, January 2016. ACM.

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

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

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

19 Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 146–157, New York, NY, USA, January 1988. ACM.

20 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.

21 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.

22 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.

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

24 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.

25 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.

26 Luísa Lourenço and Luís Caires. Dependent information flow types. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 317–328, New York, NY, USA, January 2015. ACM.

27 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.

28 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.

29 Martin Mohr, Jürgen Graf, and Martin Hecker. Jodroid: Adding android support to a static information flow control tool. In *Software Engineering (Workshops)*, pages 140–145, 2015.

30 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.

31 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.

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

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

34 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.

35 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.

36 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.

37 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.