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

May 26, 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 provide an independent 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.

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 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, did not discuss tractability. In the process of developing an implementation, we discovered that taint tracking can be optimized by performing it in a phase distinct from abstract interpretation. We discuss the details of this optimization and its implications.

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 [23] 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

DOI 10.1109/SPW.2016.58

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 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 $\varsigma \to \varsigma'$ and $\varsigma \in \gamma(\hat{\varsigma})$ then there exists an abstract state $\hat{\varsigma}'$ such that $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$ 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. 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 [6], who claimed 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. We demonstrated that these common language features allow for subtle information leaks that evade detection even by this postdominance calculation but that the analysis could be modified to handle them properly [1]. More specifically, we calculate postdominance in 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.

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, our proof of non-interference is a proof that any interference will be identified by the small-step abstract interpreter.

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 while preserving our proof of non-interference.

3 Optimization

3.1 A posteriori taint tracking

In our original analysis, context taint grew monotonically during abstract interpretation and spurious context taints were pruned afterwards. In our new analysis, we move all of the information flow calculations to postprocessing.

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 [18], without making it impossible for the taint tracking mechanism to correctly identify where information flows based on the state graph alone.

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). Therefore, context taints can be pruned at each step of the analysis, maximizing its precision and preventing spurious context taints from requiring additional spurious computation.

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 (except that some spurious context taints do not occur). Like graph exploration, propagation of taints continues until it has reached a fixed point. The taint tracking 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.

The asymptotic complexity of this analysis is identical to that of the original analysis. However, there are improvements to the complexity of 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 has the potential to act as does abstract garbage collection [19]; by removing spurious flows, it may improve performance in a way not predicted by asymptotic complexity.

3.2 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 Discussion

4.1 Analysis-agnostic non-interference

The proof of non-interference requires a sound state graph and, since allocators may be nondeterministic, information about the addresses written and read at each state. Because the abstract interpretation's state graph is unchanged in our revised analysis, all modifications and optimizations to abstract interpretation may be used without affecting our proof of non-interference. These modifications and optimizations include but are not limited to store widening, PDCFA [7], and abstract garbage collection [19].

Additionally, if the bookkeeping during analysis were expanded to include stack behaviors, including each execution point visited when searching for an exception handler, it may 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.

4.2 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 removed 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 [19].

5 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 [23] and uses entry-point saturation [16]. Sabelfeld and Myers [22] 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 work by Volpano and his collaborators [25] [26] continued along the same vein.

Many analyses are suitable for finding information flows but not for proving their absence. Kim et al. [15] and Arzt et al. [2] both analyze Dalvik bytecode but do not address implicit flows. Other analyses [27] [14] [17] [8] [13] [21] on other languages handle only some implicits, if they address them at all.

Analyses that guarantee non-interference do exist [24] [10] [3] [20] [4] but target languages without important features. As such, these analyses cannot be applied directly to Dalvik bytecode.

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

6 Conclusion

Our original theory of analysis is theoretically sound but admits optimization. Also, it may be possible to further generalize it, making it a second phase of analysis that is 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 without the need for additional theoretical work. Other analyses may also be performed a posteriori.

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] 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] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.