A Dynamic Taint Analyzer for Distributed Systems

Xiaoqin Fu
Washington State University, Pullman, USA
xiaoqin.fu@wsu.edu

Haipeng Cai
Washington State University, Pullman, USA
haipeng.cai@wsu.edu

ABSTRACT

As in other software domains, information flow security is a fundamental aspect of code security in distributed systems. However, most existing solutions to information flow security are limited to centralized software. For distributed systems, such solutions face multiple challenges, including technique *applicability*, tool *portability*, and analysis *scalability*. To overcome these challenges, we present DistTaint, a dynamic information flow (taint) analyzer for distributed systems. By partial-ordering method-execution events, DistTaint infers implicit dependencies in distributed programs, so as to resolve the applicability challenge. It resolves the portability challenge by working fully at application level, without customizing the runtime platform. To achieve scalability, it reduces analysis costs using a multi-phase analysis, where the pre-analysis phase generates method-level results to narrow down the scope of the following statement-level analysis. We evaluated DistTaint against eight real-world distributed systems. Empirical results showed DistTaint's applicability to, portability with, and scalability for industry-scale distributed systems, along with its capability of discovering known and unknown vulnerabilities. A demo video for DistTaint can be downloaded here or viewed here online, and the tool package here.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Software security engineering**.

KEYWORDS

Distributed systems, dynamic taint analysis, scalability, portability

ACM Reference Format:
Xiaoqin Fu and Haipeng Cai. 2019. A Dynamic Taint Analyzer for Distributed Systems. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3338906.3341179

1 INTRODUCTION

With increasing demands for computation at large scale, distributed software has been increasingly developed. As other domains of software applications, distributed software also suffers from varied security vulnerabilities. For example, a real-world distributed system, Apache Zookeeper [6], had a security vulnerability as reported in CVE-2018-8012 [1]: *"No authentication/authorization is enforced when a server attempts to join a quorum in Apache ZooKeeper....."*. With this vulnerability, attackers might easily gain access to a server of Zookeeper, and might lead to severe damages or losses. Especially, compared to centralized programs, distributed systems usually have larger code sizes, with their decoupled components running at physically separated machines without a global timing mechanism. These characteristics, among others, contribute to the greater complexity of distributed software, making it even more difficult to defend the code security for these systems.

Taint analysis is a popular technique [19] used for defending against these vulnerabilities. It helps users identify where their sensitive data may be leaked to untrustworthy parties as revealed by tainted information flow paths (or *taint flow paths*). A number of static taint analysis tools have been developed over the past decades, including JFlow [20], Jif [21], FlowCaml [23], and SPARK Examiner [4], among others. However, as static (conservative) taint analyzers, these tools suffer from possible unsoundness due to the use of dynamic language constructs (e.g., reflective calls and dynamic code loading) in modern software.

In contrast, dynamic taint analysis (a.k.a dynamic information flow analysis) has been regarded as a powerful technique for software security that is more precise than static approaches, since it monitors and/or computes taint flows that are actually exercised during the program executions [22]. Many dynamic taint analyzers exist, (e.g., RIFLE [25], Panorama [26], TaintBochs [13], Dytan [14], Suh [24], Privacy Oracle [18], and TaintEraser [27]). Unfortunately, they were mostly developed for centralized software and cannot be immediately applied/adapted to common distributed systems. A major reason for this applicability issue lies in that these tools compute (both implicit and explicit) information flows based on *explicit* dependencies, which do not exist among distributed (decoupled) components in common distributed software.

Other existing dynamic taint analyzers, such as Panorama [26] and TaintDroid [15], may not be subject to the *applicability* barrier yet typically rely on the underlying runtime platform (e.g., operating system) being modified (e.g. instrumented). These tools thus suffer from *portability* problems—for each of the updated versions of the runtime platform, the tool may also need to modified, which may require substantial effort and not always be possible. To support information flow security defense for distributed software, we have developed DistTaint, a dynamic taint analyzer that addresses both the applicability and portability challenges to existing peer solutions.

In particular, to overcome the applicability challenge, DistTaint infers and reasons about (statically implicit) inter-process dependencies based on a global partial ordering of methods across the system execution via monitoring

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3341179

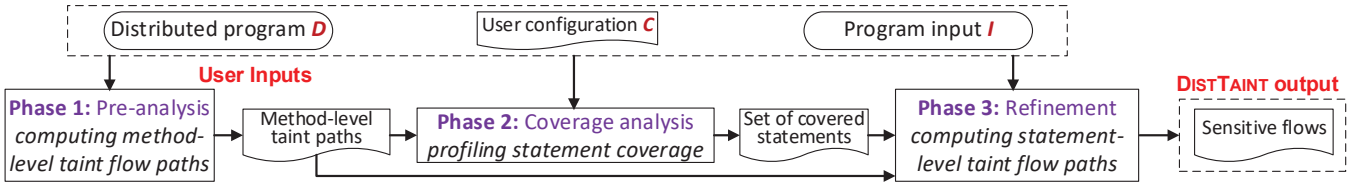

Figure 1: An overview of DistTaint architecture featuring a refinement-based, three-phase analysis scheme.

happens-before relations among execution events of the methods. Secondly, it resolves the portability challenge by working purely at application level without requiring any change of the underlying platforms or other elements of the run-time environment.

However, overcoming these challenges come at the cost of suffering from an even greater *scalability* challenge, relative to that faced by a typical dynamic analysis (due to their commonly non-trivial overheads). This scalability challenge is only aggravated in the context of distributed systems due to their larger code sizes and greater complexity, especially when a practical level of precision and (fine-)granularity is required by an attractive tool option for realistic adoption. To address this issue, DistTaint features a multi-phase analysis strategy to reduce its overheads. In particular, it computes the final (statement-level) taint paths in its fine-grained dynamic dependence analysis phase (Phase 3) that utilizes statement coverage information collected in another phase (Phase 2), as guided by the results of a rough but rapid (method-level) taint analysis in the preliminary phase (Phase 1).

We have implemented DistTaint for working with real-world, enterprise-scale distributed systems mainly written in Java. To validate its merits over existing peer tools, we also have applied DistTaint to eight distributed software systems of different architectures, applications domains, and scales, including Zookeeper [6] and Voldemort [5]. DistTaint worked successfully with all these diverse systems. More importantly, it was able to discover both known and unknown information flow security vulnerabilities in these systems, with promising scalability, efficiency, and effectiveness (precision). We have shared the entire tool package of DistTaint, including its source code and usage documentation, found here, along with a demo video here.

2 ARCHITECTURE

An overview of DistTaint's architecture is given in Figure 1. To balance its cost and effectiveness, DistTaint has three phases for its analysis: pre-analysis, coverage analysis, and refinement. It takes three **user inputs**: the distributed program D under analysis, the arbitrary run-time input set I for D, and the user configuration C. This configuration specifies the sources and sinks of user interest and a list of message-passing APIs that DistTaint probes for monitoring and profiling inter-process communication events.

With these user inputs, DistTaint computes the taint flow paths between any source and any sink of C with respect to I, in *three phases*. In the first phase (**pre-analysis**), DistTaint computes method-level flow paths to avoid otherwise expensive computation (overcoming the scalability challenge) of the next two phases by narrowing down their analysis scope. Then, in the second phase (**coverage-analysis**), the tool only calculates the statement-level coverage for methods on any of the method-level flow paths, as found in the first phase. The coverage information is later used in a statement-level taint analysis of the last phase. In this last (third) phase (**refinement**), DistTaint computes the statement-level information flow paths as the final **DistTaint output** (i.e., fine-grained *sensitive flows*) through a hybrid dynamic dependence analyses as guided by the pre-analysis result.

3 PHASE 1: PRE-ANALYSIS

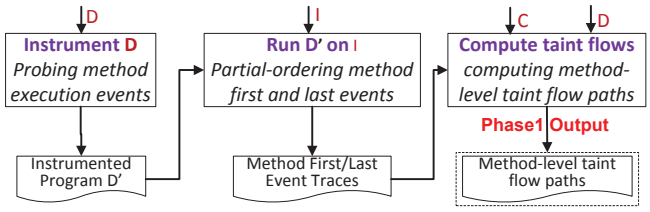

Figure 2: Pre-analysis (Phase 1) workflow of DistTaint.

The goal of this phase is to provide a coarse (method) level of analysis result, via a rough (conservative) and rapid analysis, that will reduce the costs of the next two phases hence enabling the overall scalability of our tool. Particularly, in this phase, DistTaint computes method-level taint flow paths between each source-sink pair defined before (in configuration C), in three major steps as shown in Figure 2.

In the **first step**, DistTaint produces the instrumented version D' of program D, by inserting probes to monitor the (*entry* and *returned-into*) method-execution events of each method and each (*sending a message* or *receiving a message*) message-passing event. To identify where to probe the message-passing events, DistTaint refers to the message-passing API list in C. If it is not provided by the user, DistTaint would use a default list of the most commonly used APIs in the Java SDK.

In the **second step**, DistTaint records the *first* entry and *last* returned-into events for every method. The reason only these events are monitored is because they suffice for inferring the happens-before relations among all method-execution events, hence the approximate (control-flow-based) dependencies among associated methods [11] across all processes. Meanwhile, message-passing events, albeit themselves not traced, are handled during this step to partial-order method-execution events based on the Lamport time-stamping algorithm [2, 12], so as to determine the happens-before relation between any two method-execution events later. In the **last step**, DistTaint computes method-level taint flow paths as the output of this phase, by identifying the sequence of methods between any source and any sink exercised during the execution. DistTaint uses by default the lists of sources and sinks we manually curated according to the official documentation of the security-relevant APIs in the Java SDK.

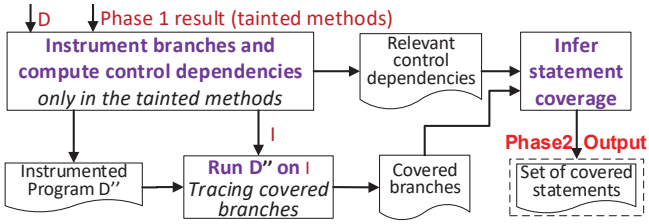

Figure 3: Phase 2 workflow of DistTaint.

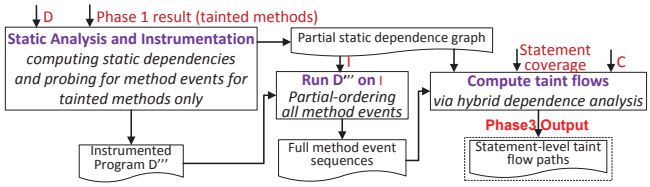

Figure 4: Phase 3 workflow of DistTaint.

4 PHASE 2: COVERAGE ANALYSIS

DistTaint utilizes statement-level coverage data to refine (in Phase 3) the method-level taint paths resulted from Phase 1. Thus, the objective of this phase is to produce the statement coverage, also in three steps as depicted in Figure 3. Guided by the pre-analysis result, only the coverage of statements in the methods on any of the method-level taint paths (i.e., *tainted methods*) is monitored.

In the **first step**, DistTaint computes control dependencies for the tainted methods and inserts probes for each branch (rather than for every statement, in order to reduce the monitoring cost). In the **second step**, DistTaint executes the instrumented program D'' using the same input I as in Phase 1 and records covered branches. In the **last step**, statements that are control dependent on a covered branch are regarded as covered. Thus, all covered statements are computed, as the eventual output of this phase.

5 PHASE 3: REFINEMENT

The objective of the third phase is to compute statement-level (fine-grained) taint paths via refining the method-level (coarse) results computed in the first (pre-analysis) phase. In this phase, DistTaint balances the analysis overhead and precision by utilizing static information (i.e., data/control dependencies) at statement level along with dynamic information at both method (i.e., partial-ordered method-execution event sequences) and statement (i.e., statement coverage) levels. This phase consists of three key steps as shown in Figure 4.

The **first step** is mainly a static analysis that builds a static dependence graph of the program D, but it also instruments D for monitoring the full sequence of method-execution (i.e., *entry* and *returned-into*) events. The static dependencies are computed at statement level, to be used as an essential type of information by the hybrid data flow analysis in the last step. Specifically, DistTaint computes data/control dependencies [17] within and across threads. DistTaint does not explicitly separate or recognize decoupled components in terms of their functionality roles (e.g., a *server* component versus *client* components). Thus, during this static dependence computation, it does not compute the implicit dependencies among processes—interprocess dependencies are later inferred from the the global partial ordering of method-execution events. As a result, the fixed-point iteration of the data flow analysis automatically stops at component boundaries, recognizing the components in an implicit manner. On the other hand, when computing such static dependencies, in order to cover all of the components in D, the static analysis searches all possible entry points (i.e., all classes containing the `main` method) of D and starts the data flow computation from each of the entry points found. All the control dependencies have been computed in Phase 2, which are simply reused here.

Another task of the first step is to instrument the program D by which probes are inserted such that *every single* instance of method entry and method returned-into event will be traced at runtime. The reason that this fine-grained method execution tracing is instrumented is because DistTaint employs a precise activation-based hybrid dependence analysis algorithm [8] for each component. The key idea is that the algorithm carefully propagates data flow facts (i.e., forward dynamic dependencies here [10]) from one method to another based on the semantics of each method-execution event (e.g., static dependencies induced by parameter passing are exercised/activated only by the calling method's entry event being *immediately* followed by the callee's entry event). As such, only the first and last events as used in the first phase (which suffice for inferring roughly approximated dependencies purely based on happens-before relations) are not sufficient for the precise analysis here; instead, the full sequence of (instance-level) method-execution events is needed [9].

It is important that in the first step both the static dependence computation and instrumentation are limited to the tainted methods only (i.e., the methods found on any method-level taint flow path as a result of the first phase)—hence the *partial* static dependence graph, and partial instrumentation. This is the key for DistTaint to achieve practical efficiency and scalability—a whole-system statement-level static dependence analysis would not be practical in terms of its time cost, according to our empirical experience with large distributed systems. For example, such a whole-system analysis for ZooKeeper did not finish after 12 hours in our experiment environment (see Section 6).

In the **second step**, DistTaint runs the instrumented program D''' using the same input I as used before to generate traces in all processes to record the instance-level method-execution events. We note that in our implementation of DistTaint, we merged the profiling steps of Phases 2 and 3. That is, although technically statement coverage is a separate analysis, instrumentation for branch coverage is merged into the instrumentation for this phase. This makes one pass of execution produce both the covered branches and the instance-level method-execution events (for the tainted methods only). The merging is possible because there is no dependency between Phase 2 and Phase 3—albeit both depend on Phase 1. The benefit of the merging is to avoid possible under- and/or over-tainting caused by deviations between the otherwise two separate executions for Phases 2 and 3 due to potential non-determinism in the execution of D. In the **last step**, DistTaint takes as inputs multiple forms of data, including the set of covered statements, the full sequence of method-execution events, and the partial static dependency graph. Then, DistTaint computes statement-level taint flow paths with respect to the sources and sinks specified in C, both intraprocess and

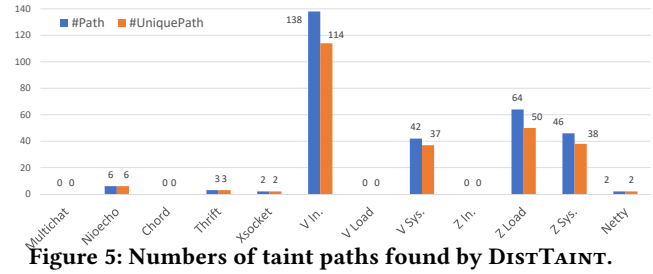

Figure 5: Numbers of taint paths found by DistTaint.

interprocess ones, as the final output of DistTaint, through the activation-based hybrid dependence analysis mentioned above.

6 APPLYING DISTTAINT

This section describes two use cases of DistTaint and presents its efficiency and scalability results. We then discuss its limitations.

6.1 Use Cases

We have successfully applied DistTaint to eight distributed Java programs, most of which are real-world industry-scale systems. These programs cover various architectures, domain, code sizes, and message-passing protocols, including a peer-to-peer system OpenChord [7], a distributed datastore Voldemort [5] (used at LinkedIn for highly-scalability storage services [3]), and a commonly used distributed coordination service Zookeeper [6] (by Yahoo! and Apache Hadoop). As run-time inputs, we used varied types of test cases that come with these subject systems [16], including integration tests, load tests, and system tests. Our experiments were all performed on Ubuntu Linux 16.04 workstations with an Intel 2.27GHz CPU and 32GB RAM.

Case 1: general effectiveness. Figure 5 shows the number of flow paths (y axis) for each subject and execution (listed on the x axis, with abbreviations of Chord for OpenChord, V for Voldemort, Z for ZooKeeper, In. for integration test, and Sys. for system test). For each subject, DistTaint computed the total (*#Path*) and unique (*#UniquePath*) number of paths from valid user-defined source/sink pairs—the latter is essentially the number of source/sink pairs having at least one taint flow path in between. Our manual inspection revealed that among all these reported flow paths, there were no false positives, showing the promising precision of our tool. Yet, without the ground truth for these systems, we were unable to assess the recall of our tool.

Case 2: finding real vulnerabilities. From varied sources (e.g., bug repositories and CVE database), we collected 10 real-world vulnerability cases: 7 for Voldemort and 3 for ZooKeeper, and checked whether our tool can discover these real vulnerabilities. Our results were highly promising: among these 10 cases, DistTaint successfully discovered 9—the only one it missed was a case with Voldemort. We manually looked into this failure case and found that the reason was because the vulnerability was not exercised in the executions we considered. We also found that all the successfully identified cases involved interprocess information flows. Thus, conventional taint analyzers limited to centralized software would miss these vulnerabilities.

6.2 Efficiency and Scalability

For the total costs on static analysis and instrumentation, DistTaint took 19 minutes by average over the eight subjects and their test cases, with a maximum of 113 minutes on Voldemort due to its greatest complexity. The run-time slowdown ranged from 9% to 74% across the three phases. DistTaint took 7 seconds to analyze one pair of source and sink on average in the last phase. The storage cost of DistTaint was small, 138MB on average with a maximum of 243MB. While we were not aware of a baseline tool suitable for a fair comparison, these numbers suggest the highly promising efficiency and scalability of our tool.

6.3 Limitations

As a purely application-level approach, DistTaint relies on instrumenting the given system for its dynamic analysis. Thus, for systems that do not allow for instrumentation, the tool would not apply. Also, the effectiveness of DistTaint can be immediately affected by whether the message-passing events are completely captured. If a system uses non-standard APIs (not from Java SDK) for interprocess communication, the APIs would need to be added to the list of message-passing APIs as part of the configuration *C*.

As a dynamic tool, DistTaint naturally suffers from under-tainting since it only analyzes the specific executions given; the underlying static analysis used may also be unsound due to uses of dynamic language constructs (e.g., dynamic code loading) in the subject system. Another limitation is that, although it purposely merges the Phase 2 and 3 executions into one to avoid non-determinism induced execution deviations, it is subject to the same concern for the possible deviations between this (merged) single execution and Phase-1 execution. As a result, DistTaint may suffer additionally from over-tainting issues. One way to overcome this limitation is to profile all the dynamic data needed (including those by Phase 3) at once (in Phase 1 execution), at the cost of losing the benefit of reducing instrumentation scope in Phase 2 and method-execution tracing scope in Phase 3. Losing this benefit may not compromise the scalability of our tool, though, since the overall cost is dominated by that of the static dependence analysis, which is not affected by the non-determinism issue.

7 CONCLUSION AND FUTURE WORK

We developed DistTaint, an application-level dynamic taint analysis tool for distributed systems that overcomes several practicality challenges to existing peer tools. It transparently works on distributed systems without changing underlying platforms to avoid portability issues. It approximates interprocess dependencies based on happens-before relations among methods to address the dependence implicity challenge that constitutes a major applicability barrier for existing solutions. Finally, DistTaint resolves the scalability challenge by using a multi-phase analysis strategy with which coarse yet cheap pre-analysis results are used to narrow down the scope of later fine-grained analyses. We implemented DistTaint for Java and applied it to several large-scale distributed software against diverse executions, and demonstrated its promising scalability and effectiveness, along with its capability of discovering various known and unknown vulnerabilities. Based on DistTaint, we plan to develop a distributed, online dynamic analysis framework for continuously running distributed services.

REFERENCES

[1] 2014. Vulnerability Details : CVE-2014-0085. https://www.cvedetails.com/cve/CVE-2014-0085/.

[2] 2017. Wikipedia. Lamport timestamps. https://en.wikipedia.org/wiki/Lamport_timestamps.

[3] 2019. Wikipedia—Voldemort (distributed data store). https://en.wikipedia.org/wiki/Voldemort_(distributed_data_store).

[4] AdaCore. 2010. SPARKPro. https://www.adacore.com/sparkpro.

[5] Apache. 2015. Voldemort. https://github.com/voldemort.

[6] Apache. 2015. ZooKeeper. https://zookeeper.apache.org/.

[7] Bamberg University. 2015. Open Chord. http://sourceforge.net/projects/open-chord/.

[8] Haipeng Cai. 2018. Hybrid Program Dependence Approximation for Effective Dynamic Impact Prediction. *IEEE Transactions on Software Engineering* 44, 4 (2018), 334–364.

[9] Haipeng Cai and Raul Santelices. 2014. DIVER: Precise Dynamic Impact Analysis Using Dependence-based Trace Pruning. In *Proceedings of International Conference on Automated Software Engineering*. 343–348.

[10] Haipeng Cai, Raul Santelices, and Douglas Thain. 2016. DiaPro: Unifying Dynamic Impact Analyses for Improved and Variable Cost-Effectiveness. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 2 (2016), 18.

[11] Haipeng Cai and Douglas Thain. 2016. DISTEA: Efficient Dynamic Impact Analysis for Distributed Systems. *arXiv preprint arXiv:1604.04638* (2016).

[12] Haipeng Cai and Douglas Thain. 2016. DistIA: A cost-effective dynamic impact analysis for distributed programs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 344–355.

[13] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. 2004. Understanding data lifetime via whole system simulation. In *USENIX Security Symposium*. 321–336.

[14] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 196–206.

[15] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[16] Xiaoqin Fu and Haipeng Cai. 2019. Measuring interprocess communications in distributed systems. In *Proceedings of the 27th International Conference on Program Comprehension*. IEEE Press, 323–334.

[17] Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 1 (1990), 26–60.

[18] Jaeyeon Jung, Anmol Sheth, Ben Greenstein, David Wetherall, Gabriel Maganis, and Tadayoshi Kohno. 2008. Privacy oracle: a system for finding application leaks with black box differential testing. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 279–288.

[19] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. 2011. Dta++: dynamic taint analysis with targeted control-flow propagation.. In *NDSS*.

[20] Andrew C Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 228–241.

[21] Andrew C Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2001. Jif: Java information flow. *Software release. Located at http://www.cs.cornell.edu/jif* 2005 (2001).

[22] Alejandro Russo and Andrei Sabelfeld. 2010. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd IEEE International Conference on Computer Security Foundations Symposium (CSF)*. IEEE, 186–199.

[23] Vincent Simonet. [n.d.]. Flow Caml. https://www.normalesup.org/~simonet/soft/flowcaml/.

[24] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. In *ACM Sigplan Notices*, Vol. 39. ACM, 85–96.

[25] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A Reis, Manish Vachharajani, and David I August. 2004. RIFLE: An architectural framework for user-centric information-flow security. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 243–254.

[26] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 116–127.

[27] David Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. 2011. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review* 45, 1 (2011), 142–154.