On the Scalable Dynamic Taint Analysis for Distributed Systems

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

ABSTRACT

To protect the privacy and search sensitive data leaks, we must solve multiple challenges (e.g., *applicability*, *portability*, and *scalability*) for developing an appropriate taint analysis for distributed systems. We hence present DistTaint, a dynamic taint analysis for distributed systems against these challenges. It could infer implicit dependencies from partial-ordering method events in executions to resolve the applicability challenge. DistTaint fully works at application-level without any customization of platforms to overcome the portability challenge. It exploits a multi-phase analysis to achieve scalability. By proposing a pre-analysis, DistTaint narrows down the following fine-grained analysis' scope to reduce the overall cost significantly. Empirical results showed DistTaint's practical applicability, portability, and scalability to industry-scale distributed programs, and its capability of discovering security vulnerabilities in real-world distributed systems. The tool package can be downloaded here.

CCS CONCEPTS

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

KEYWORDS

Distributed systems, Dynamic taint analysis, Scalability

ACM Reference Format:
Xiaoqin Fu. 2019. On the Scalable Dynamic Taint Analysis 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, 3 pages. https://doi.org/10.1145/3338906.3342506

1 PROBLEM AND MOTIVATION

Distributed systems have been increasingly developed for various computation tasks. However, Distributed systems are usually complex and have large code sizes. Their decoupled components typically run at different machines without a global clock. They hence suffer from security vulnerabilities, such as data leakage, owing to these characteristics. For example, sensitive information (e.g., account/password) might leak and cause serious losses and damages. Thus, we need to check sensitive data flowing throughout a distributed program (across its decoupled processes) to defend against such information flow threats.

Taint analysis is employed for defending against code vulnerabilities by identifying where the sensitive data may be leaked via taint flows. However, existing (static or dynamic) taint analyses have suffered several challenges. The applicability challenge presents because most of them were developed for centralized software and relied on *explicit* dependencies, but dependencies among decoupled (distributed) components in common distributed software are *implicit*. The portability challenge exists since some existing taint analyzers may depend on customized or modified platforms. The scalability challenge arises due to the great complexity and large sizes of real-world distributed systems.

2 BACKGROUND AND RELATED WORK

Developers have developed both static and dynamic taint analyses. Most early analyses are static [18] (e.g., FlowCaml [20] and JFlow [15]). These solutions suffer the imprecision of static analysis naturally and are unsound owing to the dynamic features of modern languages [14]. Some other static analyses focus on special programs rather than the most common distributed systems [9], such as points-to analysis [19] for Java RMI systems, and EAndroid [16] for Android apps. Thus, they face applicability challenges. On the other hand, most dynamic analyses [8, 11, 12, 21, 22] require architecture-specific frameworks and emulators (e.g., PIN and QEMU) or platform (Java virtual machine or even operating system) customizations, hence portability challenges exist. Several other dynamic analyses [1, 2] focus on special JavaScript programs and they are unsuitable for general distributed systems so that there are also applicability challenges. In particular, the algorithm *CGCA* [3] could induce inter-process dependencies. However, it has not been implemented on large distributed programs so that it might still face the scalability challenge.

3 APPROACH

To support information flow security defense for distributed software, we have developed DistTaint, an application-level dynamic taint analyzer (a.k.a information flow analyzer) based on Soot [13]. It reused relevant components from our previous works for method execution profiling [5, 7] and hybrid dependence abstraction [4, 6], and the Indus framework [17] for threading-induced dependence analysis. DistTaint could solve challenges to existing analyzers. To overcome the applicability challenge, DistTaint infers statically implicit inter-process dependencies from a global partial ordering of executed methods via monitoring happens-before relations among method events in the executions. As an application-level solution, it eliminates the requirement of platform customizations so as to resolve the portability challenge. DistTaint generates the

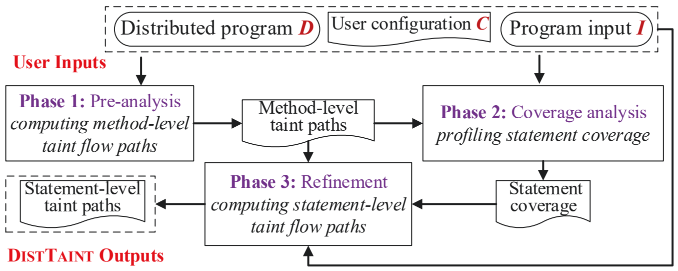
Distributed program D
User configuration C
Program input I
User Inputs
Phase 1: Pre-analysis computing method-level taint flow paths
Method-level taint paths
Phase 2: Coverage analysis profiling statement coverage
Statement coverage
Phase 3: Refinement computing statement-level taint flow paths
Statement-level taint paths
DistTaint Outputs

Figure 1: The overall workflow of DistTaint

final results (statement-level taint paths) after a rapid but rough computation of method-level results in a pre-analysis. By using approximate intermediate results to narrow down the scope of the fine-grained analysis, DistTaint reduces the overall cost greatly to solve the scalability challenge.

The overall workflow of DistTaint is illustrated in Figure 1. To balance effectiveness and cost, DistTaint has three phases for distributed systems: pre-analysis, coverage analysis, and refinement. There are three **inputs** from the user: the distribute program D under analysis, the input I for D needed by DistTaint, and a user configuration C including a list of message-passing APIs and the lists of sources and sinks between which DistTaint will compute all possible taint paths with respect to I.

The approach starts with the pre-analysis phase which computes an approximated set of method-level taint paths with respect to the sources and sinks in C (*Phase 1*). Next, coverage-analysis phase produces a statement coverage only for the methods on any of the method-level taint paths (*Phase 2*). Finally, refinement phase refines the approximated results to statement-level using method-level flows and the statement coverage (*Phase 3*). DistTaint checks all possible pairs of sources and sinks exercised during the analyzed system execution, and reports as the final results valid statement-level taint paths from any source to any sink.

4 EVALUATION

Table 1: Experimental subjects

Subject (version)	#Method	#SLOC
MultiChat (r5)	37	470
NIOEcho (r69)	27	412
OpenChord (v1.0.5)	736	9,244
Thrift (v0.11.0)	1,941	14,510
xSocket (v2.8.15)	2,209	15,760
Voldemort (v1.9.6)	20,406	115,310
ZooKeeper (v3.4.11)	5,383	62,194
NettY (v4.1.19)	12,389	167,961

As shown in Table 1, we applied DistTaint to eight distributed Java programs, most of which are real-world systems, with various codes. Three types (Integration, load, and system) of testing were implemented. The subject sizes are measured by numbers of numbers of methods defined in the subject (*#Method*) and non-blank non-comment Java source lines of code (*#SLOC*). We considered pairs of all (24) sources and (39) sinks from the default user configuration as queries to each execution. We aimed to estimate the effectiveness, scalability, and practicality of DistTaint, and answer three research questions through the evaluation:

RQ1 How effective is DistTaint in terms of its precision?
Answers: DistTaint significantly reduced the taint checking effort by users so that it was effective. The evaluation showed high precision and potentially promising recall of DistTaint after we validated with all or 20 sampled paths for each subject manually and then found that the paths were all true positives.
RQ2 How efficient/scalable is DistTaint?
Answers: DistTaint was promisingly scalable and reasonably efficient for distributed systems. It took 15 minutes for analyses on all possible queries from a given user configuration and 7 seconds on a query (a source/sink pair). The executions had less than 1x run-time slowdown and a small (81MB) storage cost.

Table 2: Real-world vulnerabilities discovered by DistTaint

Subject	Vulnerability	Found	#Cases	#FNs
Thrift	CVE-2015-3254	Yes	1	0
Voldemort	Issue 101	Yes	7	1
	Issue 381	Yes		
	Issue 387	Yes		
	Issue 352	Yes		
	Issue 378	Yes		
	Issue 377	No		
	Issue 155	Yes		
ZooKeeper	CVE-2014-0085	Yes	3	0
	Bug 2569	Yes		
	CVE-2018-8012	Yes		

RQ3 Can DistTaint discover real-world vulnerabilities?
Answer: DistTaint showed promising capabilities in successfully discovering 9 of 10 real-world security vulnerabilities, shown in Table 2. Meanwhile, only one false negative demonstrated that DistTaint relies on the vulnerabilities exercised during the analyzed executions of distributed systems. Nevertheless, DistTaint is able to find real vulnerabilities related to sensitive information flows. For example, the real Zookeeper vulnerability CVE-2018-8012 [10], *"No authentication/authorization is enforced when a server attempts to join a quorum in Apache ZooKeeper......"*, could be found after DistTaint detected relevant tainted data passing through distributed processes (e.g., Client, Container, and Server), which might be on different machines, of Apache Zookeeper system.

5 CONCLUSION

We developed DistTaint, an application-level dynamic taint analysis for distributed systems, to output taint paths as the results. It is able to overcome several practicality challenges to existing taint analysis approaches. DistTaint approximates inter-process dependencies based on happens-before relations among methods to address the applicability challenge (implicit dependencies). It transparently works on distributed systems without changing underlying platforms to overcome the portability challenge. Finally, DistTaint resolves the scalability challenge by using a multi-phase analysis strategy. We implemented DistTaint for Java and applied it to several large-scale distributed systems against diverse executions, and demonstrated its promising scalability and effectiveness, along with its capability of discovering various real security vulnerabilities in industry-scale distributed software.

REFERENCES

[1] Thomas H Austin and Cormac Flanagan. 2009. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. ACM, 113–124.

[2] Thomas H Austin and Cormac Flanagan. 2010. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*. ACM, 3.

[3] Soubhagya Sankar Barpanda and Durga Prasad Mohapatra. 2011. Dynamic slicing of distributed object-oriented programs. *IET software* 5, 5 (2011), 425–433.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2011. *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley Publishing Company.

[10] NATIONAL VULNERABILITY DATABASE. [n.d.]. Vulnerability Details : CVE-2018-8012.

[11] 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.

[12] Christophe Hauser, Frédéric Tronel, Colin Fidge, and Ludovic Mé. 2013. Intrusion detection in distributed systems, an approach based on taint marking. In *Communications (ICC), 2013 IEEE International Conference on*. IEEE, 1962–1967.

[13] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. 2011. Soot - a Java Bytecode Optimization Framework. In *Cetus Users and Compiler Infrastructure Workshop*.

[14] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (2015), 44–46.

[15] 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.

[16] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. 2013. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis. In *Proceedings of USENIX Security Symposium*.

[17] Venkatesh Prasad Ranganath and John Hatcliff. 2007. Slicing concurrent Java programs using Indus and Kaveri. *International Journal on Software Tools for Technology Transfer* 9, 5-6 (2007), 489–504.

[18] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[19] Mariana Sharp and Atanas Rountev. 2006. Static analysis of object references in RMI-based Java software. *IEEE Transactions on Software Engineering* 32, 9 (2006), 664–681.

[20] Vincent Simonet and Inria Rocquencourt. 2003. Flow Caml in a nutshell. In *Proceedings of the first APPSEM-II workshop*. 152–165.

[21] 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.

[22] 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.