Scaling Application-Level Dynamic Taint Analysis to Enterprise-Scale Distributed Systems

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

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

ABSTRACT
With the increasing deployment of enterprise-scale distributed systems, effective and practical defenses for such systems against various security vulnerabilities such as sensitive data leaks are urgently needed. However, most existing solutions are limited to centralized programs. For real-world distributed systems which are of large scales, current solutions commonly face one or more of scalability, applicability, and portability challenges. To overcome these challenges, we develop a novel dynamic taint analysis for enterprise-scale distributed systems. To achieve scalability, we use a multi-phase analysis strategy to reduce the overall cost. We infer implicit dependencies via partial-ordering method events in distributed programs to address the applicability challenge. To achieve greater portability, the analysis is designed to work at application level without customizing platforms. Empirical results have shown promising scalability and capabilities of our approach.

CCS CONCEPTS
• Security and privacy → Distributed systems security: Software security engineering.

KEYWORDS
Distributed systems, dynamic taint analysis, scalability, new bugs

1 PROBLEM AND MOTIVATION
With increasing demands for various computational tasks, more and more enterprise-scale software systems are becoming distributed. These systems suffer from peculiar security vulnerabilities (e.g., data leaks across processes) due to their great complexity, large scale, and distributed design. For instance, if sensitive data (e.g., username and password) leak, there may be serious resulting losses and damages. In this context, we need an appropriate technique, such as a taint analysis, to detect sensitive information flows across multiple decoupled processes of a distributed program to defend against security vulnerabilities. However, there are multiple challenges to most existing taint analyses when applied to real-world, enterprise-scale distributed software systems, including (1) scalability challenge due to the implicit dependencies among their decoupled (distributed) processes as in common distributed programs, and (3) portability challenge resulting from the platform customizations often required by the analyses.

2 BACKGROUND AND RELATED WORK
Most early taint analyses [12, 14–16] are static and suffer from imprecision because of the nature of static analysis. They are also unsound for modern languages with dynamic features [11]. In addition, traditional static analyses can hardly apply to distributed programs because of exacerbated inaccuracies due to implicit dependencies among decoupled components of distributed programs, and thus these static analyses face applicability challenges. On the other hand, since most existing dynamic analyses [8, 9, 17] need customized platforms or architecture-specific emulators/frameworks, they face portability challenges. In particular, while the approach in [3] could compute inter-process dependencies, it has not been implemented nor evaluated on enterprise-scale distributed systems, and its (heavyweight) design implies scalability challenges. Several other dynamic approaches [1, 2] target JavaScript programs and do not work with common distributed systems either.

3 APPROACH
Based on Soot [10], we have developed an application-level dynamic taint analysis scalable to enterprise-scale distributed programs. Our approach computes statement-level taint paths as the final results after a rapid but rough computation of method-level results in a pre-analysis phase to balance the analysis precision and overheads while attaining high scalability. The overall workflow of our solution is depicted in Figure 1. It takes three inputs from the user: a distributed program D under analysis, the program input I for D, and a user configuration C including two message-passing API lists of sources and sinks.

Our technique works in three phases. In the first phase (pre-analysis), it computes approximated method-level taint paths according to the source/sink pairs in C. Then, in the second phase (coverage-analysis), it creates a statement coverage only for executed methods on the method-level taint paths from the first phase. Finally, in the third phase (refinement), the technique derives all valid statement-level taint paths, where the statements are covered and associated method-level paths, as the final results.
Figure 1: The overall workflow of the proposed technique for dynamic taint analysis of enterprise-scale distributed software.

Our tool reuses relevant code from previous work [4–7, 13] for method execution profiling, hybrid dependence abstraction, and threading-induced dependence analysis. It greatly reduces the overall cost by narrowing down the scope of the fine-grained (statement-level) analysis according to the intermediate results, hence overcoming the scalability challenge. To solve the applicability challenge, it computes implicit inter-process dependencies derived from happens-before relations among executed method events. To achieve portability, it is designed as an application-level solution without any platform customizations.

4 EVALUATION

We implemented our technique as an open-source tool and applied it to eight Java distributed system subjects of various application domains, architectures, and scales. The executions analyzed were driven by integration, load, system tests coming along with these systems. All possible pairs of (24) sources and (39) sinks manually curated were considered as taint-flow queries. With this setup, we assessed the scalability and effectiveness of our approach.

Scalability. Our technique was shown as promisingly scalable and efficient for enterprise-scale distributed systems. It took, on average per subject execution, 7 seconds to answer each individual query beyond a 15-minute one-time cost for all possible queries, with an almost-negligible storage cost (only 81MB) and an acceptable (less than 1x) run-time overhead.

Table 1: New vulnerabilities discovered by our technique

<table>
<thead>
<tr>
<th>Subject</th>
<th>Vulnerability</th>
<th>Status</th>
<th>#Cases</th>
<th>#Confirmed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netty</td>
<td>Issue 9456</td>
<td>Fixed</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Thrift</td>
<td>Issue 4924</td>
<td>Confirmed</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Issue 4926</td>
<td>Confirmed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue 4928</td>
<td>Confirmed</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue 4929</td>
<td>Pending</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue 4930</td>
<td>Confirmed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voldemort</td>
<td>Issue 505</td>
<td>Pending</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Issue 506</td>
<td>Pending</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue 507</td>
<td>Pending</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue 508</td>
<td>Pending</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xSocket</td>
<td>Bug 25</td>
<td>Pending</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Effectiveness. Our empirical results also revealed promising capabilities of the proposed solution in terms of effectiveness. Beyond finding 16 out of 22 existing real-world information flow vulnerabilities (that are documented as publicly disclosed CVEs), our approach successfully discovered 11 new security vulnerabilities, as outlined in Table 1. These new bugs are related to several enterprise-scale distributed systems (e.g., Netty, Thrift, Voldemort, and xSocket). All of these 11 cases have been confirmed by our own manual inspection. Furthermore, 5 of these have been confirmed by the developers/maintainers of respective systems, including one case (on Netty) already fixed after a relevant pull request was opened, and a new branch, including the fixed code, was merged to the master branch of the project’s repository.

5 CONCLUSION

We developed a scalable application-level dynamic taint analysis for enterprise-scale distributed systems, addressing several challenges faced by existing peer techniques via a multi-phase, refinement-based analysis strategy working purely at application level (hence avoiding any platform customizations). We implemented our technique for Java and applied it to eight distributed systems against diverse executions. Our empirical results demonstrated its promising scalability for enterprise-scale distributed systems and the capability of finding both existing and new security vulnerabilities.

REFERENCES