



AFRL-RI-RS-TR-2020-183

AUDITR: SECURING SPACE/TIME DEFENSES IN JAVA BYTECODE

UNIVERSITY OF COLORADO BOULDER

OCTOBER 2020

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2020-183 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

WALTER S. KARAS
Work Unit Manager

/ S /

JAMES S. PERRETTA
Deputy Chief, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) OCTOBER 2020		2. REPORT TYPE FINAL TECHNICAL REPORT		3. DATES COVERED (From - To) MAY 2015 - MAY 2020	
4. TITLE AND SUBTITLE AUDITR: SECURING SPACE/TIME DEFENSES IN JAVA BYTECODE				5a. CONTRACT NUMBER FA8750-15-2-0096	
				5b. GRANT NUMBER N/A	
				5c. PROGRAM ELEMENT NUMBER 61101E	
6. AUTHOR(S) Pavol Cěrný Isil Dillig Henny Sipma John R. Black Bor-Yuh Evan Chang Marijn Heule Ashutosh Trivedi Sriram Sankaranarayanan				5d. PROJECT NUMBER STAC	
				5e. TASK NUMBER RA	
				5f. WORK UNIT NUMBER DO	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Colorado Boulder University of Texas at Austin (UT Austin) Boulder CO 80309 Austin TX 78712				8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RIGA DARPA/I20 525 Brooks Road 675 North Randolph Street Rome NY 13441-4505 Arlington VA 22203				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI	
				11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2020-183	
12. DISTRIBUTION AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Final report on the STAC project, Space/Time Analysis for Cybersecurity. Report covers performance of the Colorado Team at CU Boulder, UT Austin, and Kestrel Technology.					
15. SUBJECT TERMS Cybersecurity, Big Data, C4I, Information Technology, Financial Sector, Technology Industry, Interagency, DoD Cyber Priority Steering Council					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 96	19a. NAME OF RESPONSIBLE PERSON WALTER S. KARAS
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (Include area code) N/A

TABLE OF CONTENTS

Section	Page
LIST OF FIGURES	iii
LIST OF TABLES	iv
1.0 SUMMARY	1
2.0 INTRODUCTION.....	3
2.1 LOOPR: CodeHawk based conservative and highly scalable static analysis	4
2.2 PATHR: A precise path- and context-sensitive static analysis	5
2.2.1 QUANTM: Type-directed Bounding of Memory	5
2.2.2 THEMIS: Precise Detection of Side-Channel Vulnerabilities.....	7
2.3 OBSERVER: Solver-Guided Static Analysis	7
2.3.1 REXPLOITER: Availability Vulnerabilities in Programs Using Regular Expressions	7
2.3.2 SCANNER: Asymptotic Resource Side-Channel Vulnerabilities.....	7
2.4 EXPLOITR: Explaining Potential Witnesses and Exploits	8
2.4.1 DPDEBUGGER: Debugging Differential Performance.....	9
2.4.2 FUCHSIA: Data-Driven Debugging for Functional Side Channels.....	9
3.0 METHODS, ASSUMPTIONS, AND PROCEDURES.....	10
3.1 LOOPR: CodeHawk based conservative and highly scalable static analysis	10
3.1.1 CodeHawk Java Analyzer	10
3.1.2 Taint Analysis	14
3.1.3 Cost Model	16
3.1.4 Memory Cost	22
3.1.5 Cost Profiling.....	22
3.1.6 Library Summaries	24
3.1.7 Python API	25
3.2 PATHR: A precise path- and context-sensitive static analysis	29
3.2.1 Quantm: Type-directed Bounding of Collections in Reactive Programs	29
3.2.2 Themis: Precise Detection of Side-Channel Vulnerabilities.....	31

3.3	OBSERVER: Solver-Guided Static Analysis	35
3.3.1	Rexploiter: Availability Vulnerabilities in Programs Using Regular Expressions . . .	35
3.3.2	SCANNER: Asymptotic Resource Side-Channel Vulnerabilities.	37
3.4	EXPLOITR: Explaining Potential Witnesses and Exploits.	41
3.4.1	DPDEBUGGER: Debugging Differential Performance	41
3.4.2	FUCHSIA: Data-Driven Debugging for Functional Side Channels	43
4.0	RESULTS AND DISCUSSION	49
4.1	LOOPR: CodeHawk based conservative and highly scalable static analysis.	50
4.2	PATHR: A precise path- and context-sensitive static analysis	63
4.2.1	QUANTM: Type-directed Bounding of Memory	63
4.2.2	Themis: Precise Detection of Side-Channel Vulnerabilities	66
4.3	OBSERVER: Solver-Guided Static Analysis	69
4.3.1	REXPLOITER: Availability Vulnerabilities in Programs Using Regular Expres- sions	69
4.3.2	SCANNER: Asymptotic Resource Side-Channel Vulnerabilities.	71
4.4	EXPLOITR: Explaining Potential Witnesses and Exploits.	73
4.4.1	DPDEBUGGER: Debugging Differential Performance	73
4.4.2	FUCHSIA: Data-Driven Debugging for Functional Side Channels	76
5.0	CONCLUSIONS	79
5.1	LOOPR: CodeHawk based conservative and highly scalable static analysis.	79
5.2	PATHR: A precise path- and context-sensitive static analysis	81
5.2.1	QUANTM: Type-directed Bounding of Collections in Reactive Programs	81
5.2.2	REXPLOITER: Availability Vulnerabilities in Programs Using Regular Expres- sions	81
5.3	OBSERVER: Solver-Guided Static Analysis	81
5.3.1	THEMIS: Precise Detection of Side-Channel Vulnerabilities	81
5.3.2	SCANNER: Asymptotic Resource Side-Channel Vulnerabilities.	82
5.4	EXPLOITR: Explaining Potential Witnesses and Exploits.	82
5.4.1	DPDEBUGGER: Debugging Differential Performance	82
5.4.2	FUCHSIA: Data-Driven Debugging for Functional Side Channels	82
6.0	REFERENCES	83
	LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS	87
	GLOSSARY OF TERMINOLOGY	88

LIST OF FIGURES

Figure		Page
1	Workflow of the AUDITR Tool.....	2
2	CodeHawk Tool Suite.....	10
3	Commands to Perform Taint Analysis of the malware_analyzer Application.....	16
4	Malware Analyzer: Taint Propagation from source 1: <code>InputStream.read</code> called from <code>Server.createChecksum (pc=23)</code>	17
5	Malware Analyzer: taint propagation from source 2: <code>NanoHTTPD.IHTTPSession.parseBody</code> called from <code>Server.serve (pc=32)</code>	17
6	Malware Analyzer: taint propagation from source 2: <code>NanoHTTPD.IHTTPSession.parseBody</code> called from <code>Server.serve (pc=32)</code> restricted to loop-counter sinks	18
7	Malware Analyzer: taint propagation from source 3: <code>InputStream.read</code> called from <code>Server.serve (pc=2023)</code>	19
8	Cost Analysis Framework.....	19
9	Platform-Dependent Summary Creation Procedure	25
10	Motivating Example: a Simplified Version of a Logging Server.....	29
11	Gabfeed Code Snippet that Contains a Timing Side Channel.....	32
12	Themis Configuration file for Gabfeed	32
13	Workflow of the Themis tool.....	33
14	Motivating example containing Regular Expressions Denial-of-Service (ReDOS) vulnerabilities	35
15	Matching time against malicious string size for vulnerable (left) and hyper-vulnerable (right) regular expressions from Fig. 14.....	36
16	Overview of the REXPLOITER Approach to Availability Vulnerabilities	36
17	Workflow of the SCANNER tool	40
18	Performance clusters (left) in the FOP data set and a Discriminant Regression Tree (DRT) (right) explaining the the clusters in terms of function calls.....	43
19	Two functional observations of execution time.....	46
20	Attacker observations: (a) local (b) remote	46
21	Programs P_1 , P_2 , P_3 are leaking the number of set bits, the length of secret, and the secret values, respectively. In programs P_2 and P_3 , the clustering over functions are required to find the correct number of classes of observations.	47
22	(a) Regex timing functions. Execution times of 435 secret values are modeled as 435 timing functions. How are these functions related? (b) 435 timing functions are clustered into 12 distinguishable classes of observations (clusters) using L_1 -norm distances. The presence of different clusters indicate some properties of secret patterns are leaking. What properties of secret patterns are leaking? (c) Regex decision trees. The number of calls to the basic block at line 3964 of <code>Pattern.match()</code> method discriminates different clusters. The code region shows the value of secret patterns is leaking	48
23	Pattern matching using regex (<code>buf secret</code> , <code>seq public</code>)	49

24	Commands to analyze the E2 Blogger application.....	50
25	Loops report (partial) for the E2 Blogger application	50
26	CodeHawk Annotated Control Flow Graph for UriVerifier.verify	51
27	CodeHawk Reverse Call Graph starting from UriVerifier.verify()	52
28	Commands to analyze the E2 Gabfeed_1 application	52
29	Loops report (partial) for the E2 Gabfeed_1 application	53
30	Cost model report (partial) for the E2 Gabfeed_1 application	54
31	Cost model report (partial) for the E2 Gabfeed_1 application, accounting for ran- domization loops	55
32	gabfeed_1: Control flow graph for ModPow.modPow	56
33	Commands to produce a taint trail for the modPow method in E2:gabfeed_1	57
34	gabfeed_1: Ascendant call graph for ModPow.modPow	57
35	Cost model report (partial) for the E2 Gabfeed_1 application, accounting for ran- domization loops	58
36	gabfeed_1: Descendant call graph for ModPow.modPow	58
37	gabfeed_3: Control flow graph for Sorter.changingSort	59
38	gabfeed_3: Descendant call graph for Sorter.changingSortHelper	59
39	Commands to analyze E2:graph_analyzer.....	59
40	Loops report for the E2:Graph analyzer application	60
41	graph_analyzer: Ascendant call graph forMinimizerBarnesHut.minimizeEnergy	60
42	Commands to analyze E2:snapbuddy_3.....	60
43	Loops report for the E2:SnapBuddy_3 application	61
44	textcrunchr_6: Control flow graph forSorter\$ClasschangingSort.doIt1	62
45	textcrunchr_6: Reverse callgraph forSorter\$ClasschangingSort.doIt1	62
46	Evaluation on existing vulnerabilities: a checkmark indicates that Themis gives the correct result, while cross indicates a false positive	68
47	Evaluating Themis on identifying zero-day vulnerabilities from popular Java ap- plications	68
48	Running times for exponential vulnerabilities (left) and super-linear vulnerabilities (right) for different input sizes.....	70
49	Summary of timing results for positive and negative queries.....	71
50	Summary of our Experimental Results	71
51	Information Inferred by the Attacker	72

LIST OF TABLES

Table		Page
1	Number of JDK Methods Identified that may Receive User Input, per Package	15
2	Some Cost Model Predictions for the Timing Cost of Library Functions.....	21
3	Benchmark Results for QUANTM.....	64
4	Java applications studied using DPDEBUGGER.....	74

- 5 Java applications studied using DPDEBUGGER. Legend: $\#M_T$: total number of functions in application, $\#M_R$: total number of observed functions, $\#N$: number of collected traces, C_T : Clustering algorithm (K for K -linear and S for spectral), T : computation time of DPDEBUGGER in seconds, A : accuracy of classification model, H : decision-tree height, $\#C$: Number of clusters, $\epsilon < 0.1$ sec. 76
- 6 Case Studies. Here $\#M$ is the number of methods in applications, $\epsilon_{0,1}$ is the tolerance for L_1 -norm of the timing model, $\epsilon_{1,1}$ is the tolerance for L_1 -norm of the first derivative of the timing model, A is the accuracy of the tree model, H is the height of the tree, $\#L$ is the number of leaf nodes in the tree, and T is the computation time for decision tree learning (s) 76

1.0 SUMMARY

The University of Colorado Boulder and its teammates at the University of Texas at Austin and Kestrel Technology submit this final report on our contributions to the Space/Time Analysis for Cybersecurity (STAC) project. We created a breakthrough system called AUDITR that combines static and dynamic analysis of Java bytecode and automatically uncovers intricate security vulnerabilities related to space and time complexity that can be exploited by attackers to create denial of service and side channel attacks.

Main Research Directions. The fundamental goal of the AUDITR system is to allow analysts to quickly and reliably pinpoint Java applications with exploitable security vulnerabilities. Specifically, AUDITR focuses on two kinds of security vulnerabilities related to space and time resources:

- **Availability problems**, which arise when a user-provided input controls the asymptotic complexity or termination of a program component. Since attackers can substantially degrade system performance through carefully selected inputs, availability problems can be exploited by malicious parties to launch denial of service attacks.
- **Confidentiality problems**, which arise when two different user inputs result in substantially different resource usage behaviors depending on the value of some confidential data stored in the system. If the program exhibits substantially different space or time usage characteristics depending on user input, attackers can exploit this behavior to learn confidential information through side channel attacks.

AUDITR's key goal in identifying availability and confidentiality problems is to minimize both *false positive* and *false negative*. In addition, every identified potential security vulnerability should be accompanied by *sufficient evidence* to convince a human analyst of the exploitable nature of the vulnerability. In the case of availability problems, the evidence provided by AUDITR should be in the form of an input that triggers excessive resource usage. For confidentiality problems, AUDITR's goal is to provide a pair of inputs that result in disproportionate resource usage and identify the information that an attacker can glean by observing the program's execution on these inputs. Ultimately, AUDITR's job is to maximize the productivity of the human analyst.

Key Technical Challenges. AUDITR will need to overcome a multitude of technical challenges to attain the goals described above. Specifically, the members of the AUDITR team will address the following key research problems:

- *Trade-off between precision, soundness, and scalability:* Because of the fundamental undecidability of program verification, static analysis tools either suffer from false positive, false negative, or may not terminate. Hence, to make AUDITR useful to human analysts for identifying security vulnerabilities, we achieve the right balance between precision, soundness, and scalability. AUDITR overcomes this challenge by carefully staging different program analysis techniques with different precision, soundness, and scalability trade-offs.
- *Analysis of asymptotic resource usage behavior.* To detect security vulnerabilities related to space and time usage, AUDITR not only performs worst-case asymptotic resource usage analysis but also do so in a *path-sensitive* way. In particular, since both availability and



Figure 1: Workflow of the AUDITR Tool

confidentiality problems arise due to *user-controlled* differences in time and space usage, AUDITR is able to correlate resource usage behavior with program paths that are taken due to different user inputs.

- *Providing evidence to analysts.* Even if AUDITR can accurately identify security vulnerabilities, it will not be useful to analysts unless it can also provide sufficient evidence to justify the reported warnings. Hence, AUDITR not only performs sufficiently precise static analysis, but also communicate to the analyst why it believes a certain code snippet to contain an exploitable security vulnerability.

Main Components of Our Approach. To overcome the challenges listed above, AUDITR integrates a variety of techniques. Specifically, AUDITR carefully stages a series of program analyses with increasing precision and progressively narrows down the code under analysis. As illustrated in Fig. 1, the workflow of AUDITR has four components: LOOPR, PATHR, OBSERVER, and EXPLOITR. Below we explain the key ideas underlying each of these components.

1. **LOOPR.** The LOOPR component performs a conservative and highly scalable static analysis of all relevant byte code to identify all vulnerable algorithmic components. To identify potential availability and confidentiality problems, LOOPR performs resource usage analysis to identify super-linear time or space usage and analyze dependencies between resource usage and user input and confidential data. Since the aim of LOOPR is to identify *all possible* resource usage vulnerabilities, soundness was of primary concern and precision was secondary. LOOPR (see Section 2.1) is built on the highly scalable CodeHawk Java Analyzer.
2. **PATHR.** The task of the PATHR component is to refine the results of LOOPR by performing a more precise path- and context-sensitive static analysis. In particular, PATHR will try to eliminate most of the false positives caused by spurious reachability properties as well as over-approximate resource usage bounds. To refute spurious reachability facts, PATHR will identify how and under what conditions the vulnerable component can be reached. For instance, using a more precise inter-procedural analysis, PATHR might determine that the component in question cannot be controlled by user input. Similarly, to refute overly conservative resource usage bounds, PATHR (see Section 2.2) will perform a more precise analysis and identify the conditions under which the worst-case resource usage behavior can be achieved.
3. **OBSERVER.** The task of OBSERVER is to further refine the results of PATHR by performing *solver-guided static analysis*. Specifically, when an analysts invokes OBSERVER, the tool will analyze sanitizers and determine the condition a user input must satisfy to reach the vulnerable component. OBSERVER (see Section 2.3) will also perform bounded symbolic execution

to generate *witnesses*, to be provided as a proof of vulnerability to the analyst. In the case of availability problems, a witness will be an input of bounded length k that triggers worst-case resource usage behavior for inputs of size k . In the case of confidentiality problems, OBSERVER will generate pairs of inputs that cause significant resource usage imbalance and identify what information, if any, an attacker might learn by observing the program's resource usage on these inputs.

4. **EXPLOITR.** The EXPLOITR component will further help the analyst generalize the witnesses produced by OBSERVER and generate exploits similar to those that might be used by an attacker to launch denial of service or side channel attacks. Specifically, EXPLOITR (see Section 2.4) employs a combination of *smart inductive fuzzing*, a form of dynamic analysis, and *pattern-based learning* to scale to larger inputs and inductively generalize smaller inputs identified by OBSERVER. The results of EXPLOITR will be used to rank the detected vulnerabilities according to their threat level and further convince the analyst of the exploitable nature of the problem.

Overall, we performed fundamental research that results in the capability to statically analyze Java bytecode to automatically detect security vulnerabilities related to space and time complexity that can be exploited by attackers to create denial of service and side channel attacks. The project took 48 months + 13 month of an extension. This research project was performed by our highly collaborative, multi-disciplinary team of researchers at the University of Colorado Boulder, the University of Texas at Austin, and Kestrel Technology.

2.0 INTRODUCTION

Our collective efforts have produced a groundbreaking toolchain called AUDITR to discover and explain resource-usage related security vulnerabilities in software. Most of the software written today contains subtle security vulnerabilities that are hard to detect using traditional software engineering practices, such as testing and manual code inspection. In particular, since resource-related security vulnerabilities only manifest themselves in rarely-executed program paths, they are typically not noticed until after deployment, or even worse, until an attacker discovers how to exploit them. AUDITR aims to change this dismal status-quo by bringing sophisticated program analysis techniques to the fingertips of security analysts.

Because AUDITR is based primarily on *sound static analysis*, it can reason about anomalous resource usage behavior of programs that are only possible to trigger using peculiar, carefully-crafted program inputs. Furthermore, because of its careful staging of several program analyses with different precision vs. scalability trade-offs, AUDITR aims to scale to large programs while retaining sufficient precision to verify safe programs. Finally, AUDITR is *evidence-driven* and uses a combination of solver-guided symbolic execution, pattern inference, and inductive fuzzing to produce inputs that exhibit the resource usage vulnerabilities uncovered by AUDITR's static analyses. Hence, the overarching goal of AUDITR is not only to be sound, precise, and scalable, but also to help the human analyst make sense of the identified security vulnerabilities.

To achieve these goals and make security analysts as productive as possible, AUDITR integrates the following innovative static analysis features:

- **Filtering-based static analysis design.** AUDITR consists of a chain of increasingly-more-precise program analysis tools, called LOOPR, PATHR, OBSERV, and EXPLOITR, where each analysis refines the results of the preceding analysis. This architecture allows analyses higher up in the tool chain to focus on precision, while analyses lower down in the chain can focus on soundness and scalability. Overall, this pipelined design enables AUDITR to strike a good balance between precision, scalability, and usability.
- **Precise resource usage analysis.** To accurately detect potential availability and confidentiality vulnerabilities, AUDITR statically identify (i) code snippets with super-linear resource usage behavior where the computational complexity is under the control of the user, and (ii) user-controlled resource usage imbalances that are correlated with confidential data. The static analyses performed by AUDITR utilizes new abstract domains for reasoning about complexity and container sizes and string lengths. AUDITR employs demand-driven inter-procedural refutation techniques and precise analysis of sanitizers to rule out potential false alarms. In order to improve the efficiency of the human analyst, we need an automated approach that is fast and reports as few false positives as possible. Therefore we proposed novel algorithmic complexity analysis that can find time/space complexity of components (loops) precisely and in a scalable manner.
- **Evidence-driven warning validation.** To make human analysts as productive as possible, AUDITR provides analysts with *evidence* that corroborate the warnings generated through static analysis. For example, AUDITR generates inputs that exhibit worst-case resource usage behavior or that cause substantial resource usage imbalance based on values of secret data. Such evidence will allow a security analyst to confirm many of the warnings as quickly as possible, while leaving only a handful of reports that can be inspected by the analyst.

These different synergistic features allowed our team to create an *end-to-end tool* that accurately identifies security vulnerabilities related to time and space usage. This report is organized around the basic structure of our toolchain along the LOOPR, PATHR, OBSERV, and EXPLOITR components. We will next summarize key contributions in these components, and will use the next three sections to discuss Methods, Assumptions, and Procedures (Section 3), to present main Results and Discussion (Section 4), and to present Conclusions (Section 5) of our research efforts along these four components.

2.1 LOOPR: CodeHawk based conservative and highly scalable static analysis

The role of the CodeHawk Java Analyzer in the AUDITR project was to provide a first quick assessment of the code for potential resource sinks and to enable analysts to perform a wide variety of queries on the code to gain further insights into the code from different perspectives.

Three major types of analysis were developed and extended during this project: numerical analysis, taint analysis (information flow analysis), and resource consumption analysis. The numerical analysis was extended to exhibit more graceful degradation when applied to very large methods. Furthermore, information about lengths or sizes of objects was integrated in the analysis. The taint analysis was completely redesigned to keep track of individual taint sources and to restrict taint propagation to certain selected sources only, as described in Section 3.1.2. The resource consumption analysis was newly developed and is described in more detail in Section 3.1.3.

The CodeHawk Analyzer source code and python API are available open-source on GitHub in the following repositories:

kestreltechnology/codehawk: contains the OCaml source code for the entire CodeHawk Tool Suite including the Java Analyzer. This repository also contains the platform-independent JDK method summaries.

kestreltechnology/CodeHawk-Java: contains python APIs and the scripts to invoke CodeHawk-Java analyzer and to inspect the analysis results and perform various queries. This repository also contains executables of the CodeHawk Java analyzer for MacOS and linux, as well as integrated summaries for the reference platform.

kestreltechnology/CodeHawk-Java-Platform-Summaries: contains platform-dependent JDK method summaries, as well as summaries for several dozens of the third party libraries used in the Engagement problems. This repository also contains the Java source code that is used to perform CPU time profiling of byte-code instructions and library functions.

2.2 PATHR: A precise path- and context-sensitive static analysis

Two key thrusts in the PATHR toolchain are QUANTM and THEMIS. The tool QUANTM focuses on memory-based *availability* vulnerabilities, while THEMIS focuses on precise detection of *confidentiality* vulnerabilities using quantitative extension of Hoare logic.

2.2.1 QUANTM: Type-directed Bounding of Memory. In this thrust, our aim was to statically verify that in a given program, the length of collection variables does not grow beyond a given bound. We propose a scalable type-based technique that checks that each collection variable has a given refinement type that specifies constraints about its length. A novel feature of our refinement types is that the refinements can refer to Abstract Syntax Tree (AST) counters that track how many times an AST node has been executed. This feature enables type refinements to track limited flow-sensitive information. We generate verification conditions that ensure that the AST counters are used consistently, and that the types imply the given bound. The verification conditions are discharged by an off-the-shelf Satisfiability-Modulo-Theory (SMT) solver. Experimental results demonstrate that our technique is scalable, and effective at verifying reactive programs with respect to requirements on length of collections. This work appeared in [1] and was helpful in discharging memory-based availability vulnerabilities in the engagement problems.

Collections, widely used abstract types, are the key contributors of memory usage in Java programs. Collections allow a programmer to flexibly choose different implementations leading to better modularity essential for developing good quality software. Since collections are extensively used, related performance issues have attracted considerable attention [2–4]. Besides performance issues, improper usage of collections may lead to security vulnerabilities such as Denial-of-Service (DoS) attacks. The performance and security issues are more pronounced in reactive programs such as service threads in operating systems or web applications. An important category of DoS vulnerabilities is out-of-memory error caused by collections with excessively large lengths.

The goal of this work is to verify bounds on collection lengths using a scalable type-directed approach. Given constraints on inputs, our technique statically verifies at any point of execution total length of collection variables is less than a given bound. Verifying bound on collection lengths for reactive programs brings the following *challenges*:

Non-termination. Reactive programs do not terminate. The most common method for resource bound analysis is based on finding loop bounds [5–10]. This method therefore does not directly apply to reactive programs.

Scalability. We need a scalable and modular solution, because real world reactive programs such as web servers are large (e.g. up to 30kloc).

Non-inductiveness of invariants. The necessary safety invariants might be *non-inductive*. For instance, collection lengths of a program may be bounded, but this is at first glance not provable by checking each statement in isolation, because a particular statement might simply add an element to a collection, thus breaking an invariant that is naively constructed to help verifying boundedness.

We now describe our approach, with a focus on how the three challenges are addressed. We develop a refinement type system, where the user is able to specify bounds on collection lengths, as well as an overall guarantee on the total length of all collections. These bounds might be symbolic, referring to for instance to bounds on lengths of input collections. Our tool Quantm then type checks the program, and proves (or refutes) the overall guarantee.

First, to address the challenges of non-termination, our system relies purely on *safety properties*, never requiring a liveness property such as termination. We also do not require finding loop bounds. Second, to address the challenge of scalability, we use type-based reasoning only. This entails checking at most one invariant per collection, as opposed to one invariant per each code location (as the approaches based on abstract interpretation [7, 8] might need). Third, to address the challenge of non-inductiveness of invariants, we allow the refinement refer to *AST counters* that count how many times an AST node has been executed. For instance, consider the fragment:

```
while (true) { if (*) { C: s.add(r1); ...; D: t.add(r2); } }
```

and suppose we are interested in the invariant $\text{len}(s) - \text{len}(t) \leq 1$, that is, the difference between lengths of the two collections s and t is at most 1. The invariant is not inductive, the statement $s.add(r)$ breaks it. However, let C be a counter associated with the AST node of $s.add(r1)$, and D with $t.add(r2)$. The invariant $\text{len}(s) + D = \text{len}(t) + C$ holds. We can then add a counter axiom $(D + 1 \leq C) \wedge (C \leq D)$ as the two statements are inside a same basic block. Counter axioms are the place where the limited amount of flow-sensitive information that our system uses is captured.

The inductive invariant and the axiom together imply the property we are interested in: $|\text{len}(s) - \text{len}(t)| \leq 1$. The main contributions of this work are the following.

- **Refinement types for collection lengths.** We encode the total length of collection variables as safety properties of all reachable program states, as opposed to relying on analyzing time bounds. We develop a refinement type system where the refinements allow reasoning about collection lengths.

- **AST counters for inductive invariants.** A novel feature of our refinement types is that the refinements can refer to *AST counters* that track how many times an AST node has been executed. This feature enables type refinements to track limited flow-sensitive information.
- **Empirical evaluation.** Experimental results show that our approach scales to programs up to 30kloc (180kloc in total), within 52 second of analysis time per benchmark. Moreover, we discovered a Denial-of-Service vulnerability in one of our benchmarks because of correctly not being able to verify boundedness.

2.2.2 THEMIS: Precise Detection of Side-Channel Vulnerabilities. In this thrust we developed THEMIS, an end-to-end static analysis tool for finding resource-usage side-channel vulnerabilities in Java applications. We introduce the notion of ϵ -bounded non-interference, a variant and relaxation of Goguen and Meseguer’s well-known non-interference principle. We also introduced Quantitative Cartesian Hoare Logic (QCHL), a program logic for verifying ϵ -bounded non-interference. Our tool, THEMIS, combines automated reasoning in Cartesian Hoare Logic (CHL) with lightweight static taint analysis to improve scalability. We evaluate THEMIS on well known Java applications and demonstrate that Themis can find unknown side-channel vulnerabilities in widely-used programs. We showed that Themis can verify the absence of vulnerabilities in repaired versions of vulnerable programs and that THEMIS compares favorably against Blazer, a state-of-the-art static analysis tool for finding timing side channels in Java applications. This work appeared in [11] and was helpful in several engagement examples.

2.3 OBSERV: Solver-Guided Static Analysis

Two key thrusts in the OBSERV toolchain are REXPLOITER and SCANNER. The tool REXPLOITER focuses on time-based *availability* vulnerabilities, while the tool SCANNER focuses on static detection of asymptotic resource *side-channels* vulnerabilities. In addition, our team made several fundamental contributions [12–16] to the theory and practice of efficient SAT solving and in particular [15] enabled parallel automated reasoning with linear speedups even with thousands of cores on some problems.

2.3.1 REXPLOITER: Availability Vulnerabilities in Programs Using Regular Expressions. In an algorithmic complexity attack, a malicious party takes advantage of the worst-case behavior of an algorithm to cause denial-of-service. A prominent algorithmic complexity attack is ReDOS, in which the attacker exploits a vulnerable regular expression by providing a carefully-crafted input string that triggers worst-case behavior of the matching algorithm. This work proposed a technique for automatically finding ReDOS vulnerabilities in programs. Specifically, our approach automatically identifies vulnerable regular expressions in the program and determines whether an “evil” input string can be matched against a vulnerable regular expression. We have implemented the proposed approach in a tool called REXPLOITER and found multiple exploitable security vulnerabilities in Java web applications. This work appeared in [17] and was useful in several STAC challenges.

2.3.2 SCANNER: Asymptotic Resource Side-Channel Vulnerabilities. Web applications can leak confidential user information due to the presence of unintended side-channel vulnerabilities

in code. One particularly subtle class of side-channel vulnerabilities arises due to resource usage imbalances along different execution paths of a program. Such side-channel vulnerabilities are especially severe if the resource usage imbalance is asymptotic. This work formalized the notion of asymptotic resource side-channels and presents a lightweight static analysis algorithm for automatically detecting them. Based on these ideas, we developed a tool called SCANNER that detects resource-related side-channel vulnerabilities in PHP applications. SCANNER has found 18 zero-day security vulnerabilities in 10 different web applications and reports only 2 false positives. The vulnerabilities uncovered by SCANNER can be exploited using cross-site search attacks to extract various kinds of confidential information, such as a user's medications or purchase history. While—to capture widely-used open-source web-applications such as WordPress, Joomla, OpenCart, HotCRP—this tool is implemented for the PHP libraries, the ideas developed in the work are relevant to our work in STAC as it forms fundamental theoretical advances to characterize and statically detect asymptotic side-channels. This work appeared in [18].

In this thrust, we introduced and formalized asymptotic resource side-channels, a class of vulnerabilities that can be exploited in settings where the attacker cannot precisely measure resource usage (e.g., web applications). To help developers safeguard their applications against such vulnerabilities, we also present a novel static analysis for automatically detecting asymptotic side channels. Our analysis employs a lightweight program abstraction that combines taint information with a coarse summary of resource usage. In particular, our analysis differentiates between variables that are untainted, secret-tainted, and user-tainted, and summarizes the resource usage of each code fragment as zero, constant, or unbounded. The key idea underlying our analysis is to identify branch conditions that are secret tainted and where the resource usage is unbounded along one branch but not the other. As we show experimentally, our proposed program abstraction is effective at detecting asymptotic side-channel vulnerabilities with a low false positive rate and can uncover exploitable vulnerabilities in real-world web applications.

We have implemented our proposed static analysis in a tool called SCANNER for analyzing PHP applications. While the techniques we describe in this report can be used to detect any resource-related asymptotic side-channel vulnerability, our implementation focuses on two kinds of resources, namely, time and response size. In addition to pinpointing vulnerable components, SCANNER further aids security analysts by identifying confidential database fields that may be leaked due to the detected vulnerability. Furthermore, SCANNER helps users assess the severity of the uncovered vulnerability by semi-automatically generating a Javascript exploit that performs a cross-site search attack. We evaluate SCANNER on 10 open-source PHP applications and show that it uncovers 18 side-channel vulnerabilities and reports only 2 false positives. Furthermore, we are able to exploit these vulnerabilities using cross-site search attacks and show that the attacker can extract various kinds of confidential data, such as a user's purchase history, medical records, and bids placed by the user. We have reported the vulnerabilities to the developers and many of them have been fixed by the developers at the time of this final report.

2.4 EXPLOITR: Explaining Potential Witnesses and Exploits

The EXPLOITR component developed a combination of *smart fuzzing* and dynamic analysis to generalize smaller inputs identified by OBSERVER. We present two key approaches developed as part of EXPLOITR suite: DPDEBUGGER (a data-driven technique for explaining differential performances) and FUCHSIA (a data-driven technique for debugging functional side-channels). DPDE-

BUGGER and FUCHSIA were presented in [19, 20] and were useful in several STAC engagements. In addition to these key thrusts, we developed novel techniques for fuzzing [21], quantifying and mitigating [22, 23], and explaining [24] confidentiality vulnerabilities. More details can be found in [25].

2.4.1 DPDEBUGGER: Debugging Differential Performance. Differential performance debugging is a technique to find performance problems. It applies in situations where the performance of a program is (unexpectedly) different for varying classes of inputs. The task is to explain the differences in asymptotic performance among various input classes in terms of program internals. We give a data-driven technique based on DRT learning problem where the goal is to discriminate among different classes of inputs. We proposed a new algorithm for DRT learning that first clusters the data into functional clusters, capturing different asymptotic performance classes, and then invokes off-the-shelf decision tree learning algorithms to explain these clusters. We focus on linear functional clusters and adapt classical clustering algorithms (K -means and spectral) to produce them. For the K -means algorithm, we generalize the notion of the cluster centroid from a point to a linear function. We adapt spectral clustering by defining a novel kernel function to capture the notion of “linear” similarity between two data points. We evaluated our approach on benchmarks consisting of Java programs where we are interested in debugging performance for confidentiality. We showed that our algorithm outperforms other well-known regression tree learning algorithms in terms of running time and accuracy of classification.

2.4.2 FUCHSIA: Data-Driven Debugging for Functional Side Channels. *Functional side-channels* arise when an attacker knows that a secret value of a server stays fixed for a certain time. Then, the attacker can observe the server executions on a sequence of different public inputs, each paired with the same secret input. Thus for each secret, the attacker observes a (partial) function from public inputs to execution time, for instance, and she can compare these functions for different secrets.

In this thrust, we introduced a notion of noninterference for functional side channels. We focused on the case of noisy observations, where we demonstrate with examples that there is a practical functional side channel in programs that would be deemed information-leak-free or be underestimated using the standard definition. Second, we develop a framework and techniques for debugging programs for functional side channels. We extend evolutionary fuzzing techniques to generate inputs that exploit functional dependencies of response times on public inputs. We adapt existing results and algorithms in functional data analysis (such as functional clustering) to model the functions and discover the existence of side channels. Finally, we used a functional extension of standard decision tree learning to pinpoint the code fragments causing a side channel.

We empirically evaluated the performance of FUCHSIA on a series of micro-benchmarks, as well as on realistic Java programs. On the set of micro-benchmark, we show that FUCHSIA outperforms the state-of-the-art techniques in detecting side-channel classes. On the realistic programs, we show the scalability of FUCHSIA in analyzing functional side channels in Java programs with thousands of methods. In addition, we show the usefulness of FUCHSIA in finding (and locating in code) side channels including a zero-day vulnerability in Open Java Development Kit and another Java web server vulnerability that was since fixed by the original developers.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

This section discusses some of our key methods, assumptions, and procedures in the AUDITR tool chain. For convenience, we have arranged this section along the four major components (LOOPR, PATHR, OBSERV, EXPLOITR) of our tool chain.

3.1 LOOPR: CodeHawk based conservative and highly scalable static analysis

3.1.1 CodeHawk Java Analyzer. Kestrel Technology has developed a sound static analysis platform called *CodeHawk*. CodeHawk is a customizable static analysis tool based on the mathematical theory of *abstract interpretation* [26]. CodeHawk consists of a programming-language independent abstract interpretation engine and three language front ends, as shown in Figure 2. The abstract interpretation engine provides the following components:

1. **CodeHawk Internal Form (CHIF)**, the internal engine language in which programs are represented and on which analysis is performed;
2. **Abstract Domains**, that provide semantics for all constructs in the language possibly augmented with custom operations relative to a particular decidable theory; and
3. **Fixpoint Iterators**, highly optimized algorithms that perform flow-sensitive forward and backward propagation of the semantics encoded in the domains.

The fixpoint iterators are completely transparent to the user and do not need further elaboration. CHIF and the abstract domains are described in some more detail below.

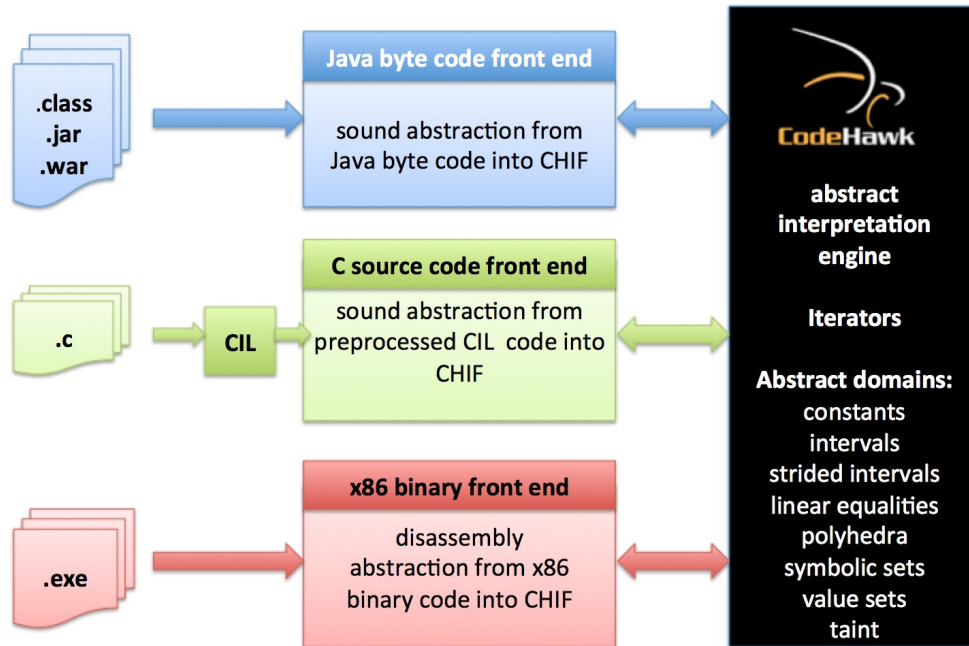


Figure 2: CodeHawk Tool Suite

CHIF. The CodeHawk Internal Form is an imperative language. Data types include integers, floats, symbols, structs, and arrays. Expressions include arithmetic expressions, boolean expressions, and expressions on sets. The language supports both structured and unstructured control flow. Control flow constructs can be arbitrarily nested, for example, loops may contain arbitrary control flow graphs constructed from jumps, as long as the control flow graphs stay within the loops. Breakout blocks are provided to enable representation of otherwise structured loops with exits to the statement immediately following the loop. The language provides assignment statements for all data types. To support operators not directly handled by CHIF expressions (for example, bit-wise operations), The language also includes analyzer directives, including commands to activate or deactivate domains, assert the validity of expressions, transfer values from one domain to another, delineate regions of the code in order to establish summary transfer relations, as well as custom operations directed at custom domains to inject constraints at particular locations.

Abstract Domains. An abstract domain is a decidable theory. It consists of a finite or infinite set of elements that form a lattice, with well-defined meet and join operations and a bottom and top element. Furthermore, it provides forward and backward transformers for all dataflow constructs in the language as well as for assert statements. The transformers are guaranteed to be an over-approximation of the concrete semantics of the operations modeled. Custom domains may include an arbitrary number of domain operations that define transformers for custom constraint generation. The core system provides several numerical and symbolic domains, including intervals, linear equalities [27] and linear inequalities [28], and symbolic sets, as well as domains specialized for binary analysis, including the strided interval domain [29] and the value-set domain [29].

New abstract domains can be easily added to the engine to support specific features of the target system or new properties of interest. For example, as part of the The Intelligence Advanced Research Projects Activity (IARPA) StoneSoup project for binary analysis, we developed two new custom domains:

Strided Interval Domain [29]. This domain is a refinement of the regular interval domain that constrains alignment. It is especially useful in binary analysis where one has to reason about memory locations on different address boundaries.

Value-set Domain [29]. This domain expresses an explicit partitioning of disjoint memory regions to allow reasoning about relative addresses (with respect to a symbolic base of the region) in terms of absolute values (their offsets), which has a much lower complexity than polyhedra and in most cases provides sufficient expressiveness, thus increasing scalability without losing precision.

Front Ends. The abstract interpretation engine forms the analysis engine for three analyzers:

- **Java byte code.** Developed under IARPA StoneSoup for a variety of properties, and further specialized for complexity and side-channel properties as described later in this report.
- **C source code.** Developed with support from DHS [30]. It provides sound static analysis of memory safety, including buffer overflow, integer overflow, use-after-free, double-free, etc: it generates verification conditions (proof obligations) for all conditions that can lead to *undefined behavior* and uses invariants generated by the abstract interpretation engine to

discharge these proof obligations. The analyzer, named CodeHawk-C, is actively being used (and extended) by Dr. Paul Black at NIST to support his work on SATE (Static Analysis Tool Exposition). The analyzer has been applied to 1.6 million lines of C source code.

- **Assembly code.** Developed originally under IARPA Stonesoup for x86 PE files with a focus on Java Native Methods [31], later extended with support from DHS [30] and Defense Advanced Research Projects Agency (DARPA)-MUSE to perform malware analysis and feature generation; most recently extended to accept ELF File formats and MIPS executables with support from the MIT's DARPA TC program.

The core of the CodeHawk Java Front End was developed as part of the IARPA StoneSoup program. It provides the interface between the byte code and the CodeHawk abstract interpretation engine, directs the analysis, interprets the analysis results returned by the abstract interpretation engine, and translates those results back into the context of byte code. The core components of the Java Front End are the following:

- **Byte-code parser:** parses Java class files into internal data structures; as part of the STAC project the parser was extended to handle the extensions introduced by Java 7 and Java 8;
- **Hierarchy analysis:** determines the relationship between classes and performs resolution of virtual and interface calls;
- **Translator:** transforms the data structures produced by the byte-code parser into an over-approximating CHIF representation.

Building on these core components we have developed and extended the target analyses:

- **Numerical analysis:** Non-relational and relational analysis of numeric variables to determine loop bounds; this analysis was originally developed and implemented as part of the StoneSoup project and further improved as part of the STAC project;
- **Taint analysis:** Information flow analysis starting from taint sources identified by the method summaries and/or the user, to resource consumption targets as resource sinks; this analysis was also first developed as part the StoneSoup, but significantly adapted as part of the STAC project to allow user interaction to focus the analysis on only those sources and sinks of interest, greatly increasing its scalability and precision, as described in Section 3.1.2
- **Resource consumption analysis:** A new abstraction framework to compute potentially non-linear bounds on resource consumption of methods or specific paths through methods, as described in more detail in Section 3.1.3

While the goal of the StoneSoup project was a fully automatic analysis without user interaction, the envisaged goal for the STAC project was to offer a wide variety of query capabilities that allows the analyst to quickly gather information about the application of interest. To facilitate this type of interaction we chose to make all analysis results explicitly available and provide a Python API that would enable easily written Python scripts to query the data in any way desired. As the analysis tends to generate a large number of facts, it was crucial to find an efficient representation to limit the file size of these results and speed up access. Below we elaborate some more on the approaches used to achieve such efficient storage. Section 3.1.7 describes the scripts we provide.

Translator. The translator is a combination of a semantics-preserving transformation and a semantic over-approximation. The transformation converts the stack-based representation of Java byte code into a register-based representation. The over-approximation abstracts all byte code instructions into CHIF instructions such that all behaviors of the original byte code are included in the behaviors of CHIF.

The Java Virtual Machine is stack-based, that is, all byte-code operations take their arguments from an operand stack and push the result back on the same stack. The CodeHawk abstract interpretation engine, however, is register-based. To enable translation into CHIF, the stack-based byte-code operations must first be converted into a stack-less representation. This conversion is achieved by simulating the stack explicitly. At each instruction the stack is copied from the previous instruction and modifications are made according to the specification of the byte-code instruction. This approach substantially increases the number of variables as each stack slot at each byte-code instruction contributes a variable; most of these variables, however, can subsequently be eliminated by local simplification, as most values are simply copied from one instruction to the next. The advantage of this two-step approach is its simplicity: each step is simple and can therefore easily be validated. The combination achieves similar code sizes as a more complicated one-pass approach would have achieved.

All byte-code instructions must be translated into CHIF. The expressiveness of CHIF, however, is less than that of byte-code instructions. For example, CHIF does not have any floating-point operations. To ensure a sound analysis, byte-code instructions that cannot be represented precisely must be over-approximated, such that all behaviors of the original program are guaranteed to be included in the set of behaviors of the CHIF representation. The simplest over-approximation is to assume that any variables written may have any value allowed by their type. An alternative technique for over-approximation is offered with uninterrupted operations, provided in CHIF to enable user-defined, analysis-specific semantics. These operations are used for field reads and field updates, method invocations, and special operations such as `newarray` where the semantics can be assigned according to the analysis that is performed. In each case care is taken that no behaviors of the original program are removed by the chosen semantics.

Numeric Analysis. Numeric analysis forms the basis for all other analyses. Its goal is to establish bounds on individual variables, in particular loop counters and arguments passed to other methods, and to determine linear relationships between sets of variables. To accomplish this, analysis is performed in two abstract domains:

- **intervals:** the valuation of variables are approximated by closed or half-closed intervals;
- **linear equalities:** linear relationships between variables are represented, while constants are treated separately, to reduce the size of the representation

As we are interested especially in loop bounds, auxiliary variables are introduced to explicitly represent loop counters, to allow a uniform treatment of loop constructs that may or may not have explicit program variables that serve as loop variables. Generating numeric invariants for loop counters critically relies on the linear equality domain, as no explicit assignments to these variables are made in the program itself, and thus invariants can be generated only in relation to actual program variables that are used in loop exit conditions.

Numerical analysis is performed in a modular fashion. The core analysis is intra-procedural; results are collected in method summaries, which are then used to refine the analysis of subsequent methods. The analysis makes two passes over the methods. In the first pass methods are analyzed in a bottom-up order through the call graph. Summaries of previously analyzed application methods called, as well as externally provided summaries of library functions called, are used to refine the analysis. New summaries are created for each method analyzed that include

- bounds on field values. Field analysis is so-called “field-aware”, that is, field values are joined based on name and type;
- relationships between arguments and return value;
- side effects on objects referred to by arguments

Loops in the call graph are arbitrarily broken and summary results taken to be TOP in those cases. In the second pass methods are analysis top-down through the call graph, providing the opportunity to constrain argument values, assuming all callers to a particular method can be identified. This approach has been proven to scale well even when using relational abstract domains.

3.1.2 Taint Analysis. Taint analysis tracks how externally provided input to an application flows through an application and affects the value of variables and the behavior of the program. For the Audit project we are interested especially in how user-provided information can affect resource consumption such as CPU time, elapsed time, and memory usage. The taint analysis depends on the numeric analysis to eliminate spurious transfers.

The taint analysis in the CodeHawk Java Analyzer is, like the numeric analysis, modular. It starts with creating variable dependency graphs for each method. The nodes in these graphs are variables (including method parameters), and calls. The edges connect operands to results, arguments to the call, the call to the return value and possibly the arguments (in case of side effects), etc. These direct transfers of taint, however, are not sufficient to capture all effects that user-provided information can have on the execution. For example, variables that are incremented within a loop may not have a direct (assignment) link with any tainted variable, but their value can still be affected by user data if that data influences the loop bound. To capture this indirect transfer our analysis adds edges from all variables that participate in the exit condition of the loop to the auxiliary loop counter variable, and from the loop counter variable to all variables that are assigned within the loop. The numeric analysis, performed before, enables a refinement of this rule: if the loop counter variable was found to have a constant bound (that is, determined within the program), these edges are omitted, as the number of loop traversals is fixed by the program itself and not influenced by the user.

Like the numerical analysis, the taint analysis is performed in two passes. In the first pass taint transfer graphs are created for individual methods, going through the call graph bottom-up. The graphs represent taint intra-procedural taint transfer, using summaries of methods analyzed earlier or library function summaries to determine the taint transfer across calls. From this internal graph a summary is created that abstracts the taint flows to just the arguments and return value of the method, as well as the fields tainted. Static fields are treated like global variables.

Originally the second pass analyzed the methods in top-down order through the call graph, setting the taint on the command-line arguments to `main` as well as argument and return values

Table 1: Number of JDK Methods Identified that may Receive User Input, perPackage

count	package				
163	java.io	7	java.sql		
55	javax.swing	5	javax.xml.parsers	1	java.rmi.server
42	java.net	5	javax.crypto	1	javax.security.auth.kerberos
39	java.nio.channels	5	javax.net	1	org.xml.sax.helpers
37	java.nio.file	5	java.util.jar	1	javax.xml.transform.stream
32	javax.imageio	4	java.util	1	javax.swing.text.html
31	java.lang.reflect	4	javax.swing.text	1	org.xml.sax
24	java.awt	2	javax.swing.table	1	java.beans
23	java.lang	2	javax.xml.validation	1	javax.net.ssl
12	java.util.zip	2	java.util.concurrent	1	java.nio.channels.spi
9	javax.tools	2	javax.sound.sampled	1	java.awt.datatransfer
8	java.security.cert	2	javax.sql	1	com.sun.net.httpserver
8	java.security	2	java.applet	1	javax.management.openmbean
7	java.rmi.registry	2	java.nio		

from library methods identified as taint sources. This approach sufficed for the StoneSoup project, as we were interested only in whether variables were tainted or not. For the STAC project, however we wanted to keep track of the actual sources of taint that could affect particular variables, such that the analyst could assess whether the taint was relevant. It quickly became clear that propagating all taint sources individually significantly increased the memory requirements of the analysis and was rather unwieldy and wasteful in general. Therefore we decided to decouple the two passes.

Taint transfer graphs from the first pass now are saved and the analyst is presented with a list of all taint sources present in the application. The second pass is then initiated by the analyst for a single taint source, which then propagates taint forward only from the location where the selected source of taint is introduced, rather than for the whole program. This two-step approach significantly reduced the analysis time and also reduced the amount of results to be interpreted.

Taint Sources Taint sources can originate in several places. Means of direct input to the program include the command-line arguments to `main` as well as the arguments to a number of other methods, including `doGet`, `doPut`, `doPost`, `doDelete` that are used in web frameworks, and `handle` (in `HttpHandler`) as these methods tend to receive their arguments from outside the application. The majority of the taint sources, however, are in library methods, and identified as such by the library method summaries. As of writing this report more than 500 library methods in the JDK have been identified as receiving user input. Table 1 summarizes the packages with these methods.

Example: Malware Analyzer (E4) To illustrate the taint analysis process we show the commands involved to analyze the `malware_analyzer` application from Engagement 4. Figure 3 shows the interaction via the python scripts (described in more detail in Section 3.1.7):

Taint source 4 is essentially the same as taint source 3, and taint source 5 does not propagate

anywhere as the command-line arguments to the `malware_analyzer` are not used. Figures 4, 6 and 7 show results of the `report_taint_trail`. The colors in the graphs indicate the following:

- **red:** the node represents a loop counter, indicating that user input may be able to influence the number of iterations of the associated loop;
- **yellow:** the node represents a branch condition, indicating that user input may be able to influence which branch is taken;
- **green:** the node represents a return value;
- **purple:** the taint source.

The taint-propagation graphs are inter-procedural, as illustrated by Figures 6 and 7.

3.1.3 Cost Model. We designed and implemented a cost framework to compute and propagate timing cost values in CodeHawk. The framework includes the following principal components:

- a generic cost model that (1) supports the computation of costs for individual basic blocks based on the costs of the instructions included in the basic block, and (2) accumulates costs for entire methods that are used to determine the cost of method calls;

```
> python chj_analyze.py malware_analyzer
.....
> python chj_analyze_taint.py malware_analyzer
.....
> python chj_report_taint_origins.py malware_analyzer
* =====
* CodeHawk Java Analyzer: TaintOrigins
* application: malware_analyzer
* date       : 2020-05-26 23:57:04
* =====
index  taint origin site
-----
1  STUB:Server.createChecksum @23: InputStream.read
2  STUB:Server.serve @32: NanoHTTPD$IHTTSession.parseBody
3  STUB:Server.serve @2023: InputStream.read
4  STUB:Server.serve @2023: ByteArrayInputStream.read
5  VAR:com.ainfosec.MalwareAnalyzer.MalwareAnalyzer.main: (java.lang.String):r0

> python chj_analyze_taint_propagation.py malware_analyzer1
.....
> python chj_report_taint_trail.py malware_analyzer1
.....
> python chj_analyze_taint_propagation.py malware_analyzer2
.....
> python chj_report_taint_trail.py malware_analyzer2
.....
> python chj_report_taint_trail.py malware_analyzer2 --loops
.....
> python chj_analyze_taint_propagation.py malware_analyzer3
.....
> python chj_report_taint_trail.py malware_analyzer3
```

Figure 3: Commands to Perform Taint Analysis of the `malware_analyzer` Application

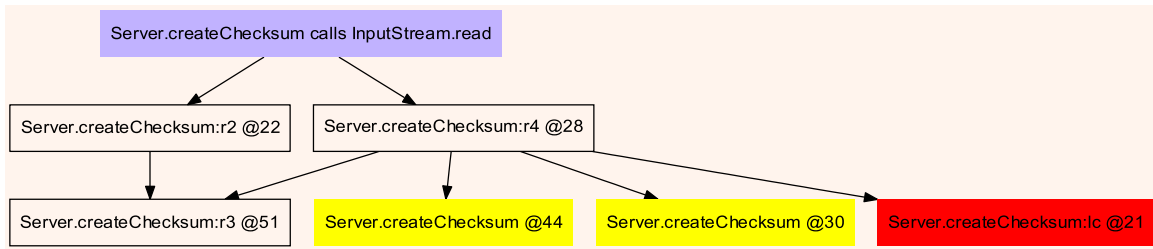


Figure 4: Malware Analyzer: Taint Propagation from source 1: `InputStream.read` called from `Server.createChecksum` (pc=23)

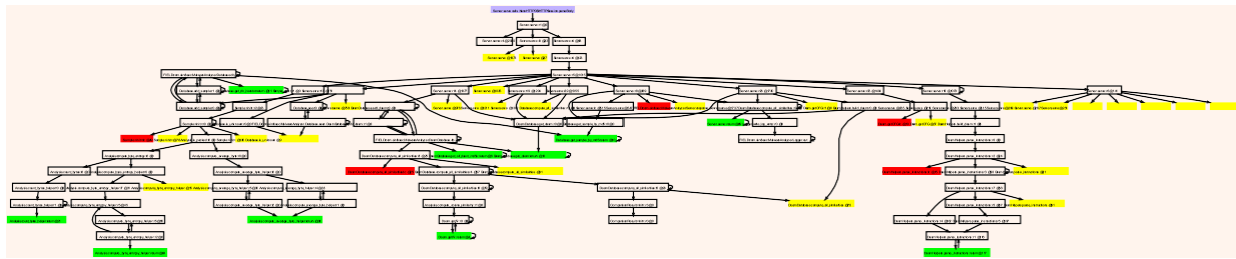


Figure 5: Malware Analyzer: taint propagation from source 2: `NanoHTTPD.IHTTPSession.parseBody` called from `Server.serve` (pc=32)

- a method cost abstractor that propagates the basic block costs provided by the cost model through the control flow graph using the iterators of the CodeHawk abstract interpretation engine. The method abstractor can be provided with external loop bound expressions that either originate directly from the numerical analysis, or are provided interactively by the human analyst.

Methods are analyzed bottom-up through the call-graph to collect costs for method calls.

Cost model abstract domain. Cost is represented as a set of symbolic expressions over field values, array lengths, and function argument values, divided into lower bounds and upper bounds. The analysis is field-aware, but not field-sensitive, that is, both static and object fields are collapsed into a single field per field definition. Our experience so far is that this level of granularity is adequate, since we are interested in bounds and orders of magnitude, rather than in tracking individual object values. Expressions are propagated bottom-up through the call graph. Costs of loops are calculated based on the loop bounds computed by the regular polyhedra numerical analysis. Computation of the cost model with symbolic expressions is fast, typically a few seconds for an application like collab.

To illustrate the type of expressions produced by this domain, consider the cost expression for the `SchedulingSandbox.replace` method in the collab application:

```
c.SchedulingSandbox.replace:
```

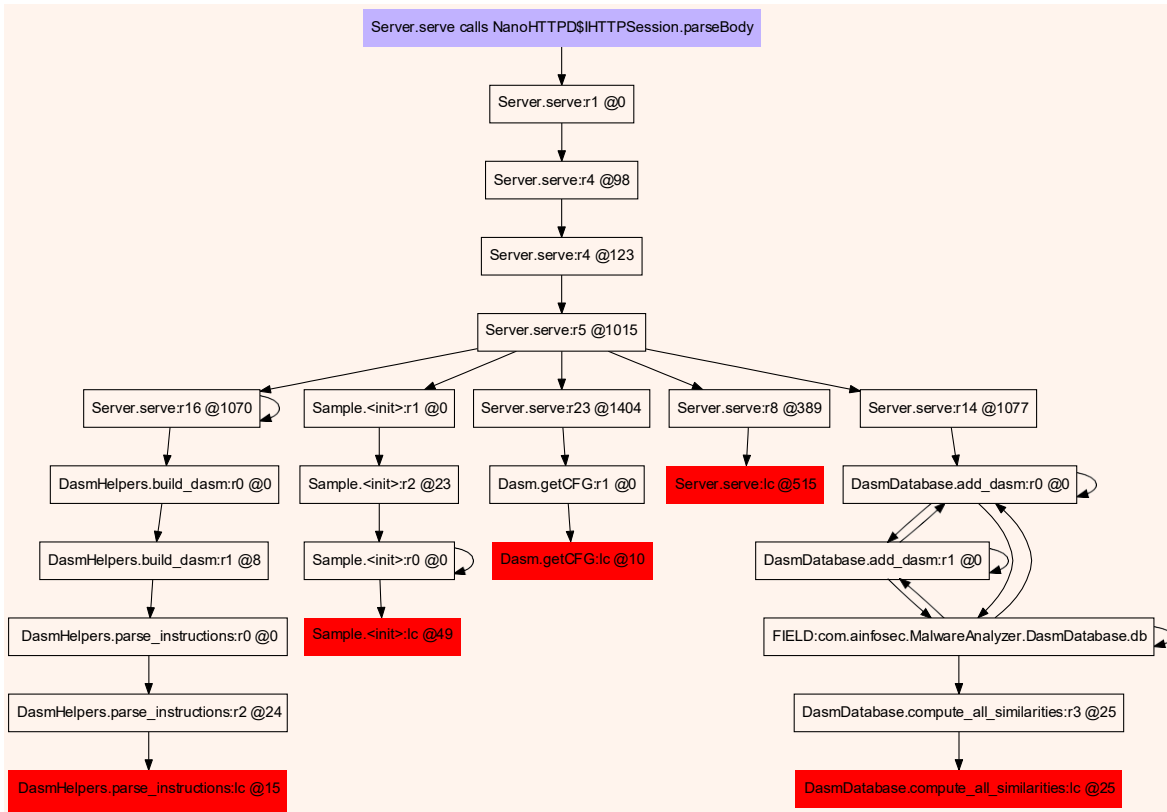


Figure 6: Malware Analyzer: taint propagation from source 2: NanoHTTPD.IHTTPSession.parseBody called from Server.serve (pc=32) restricted to loop-counter sinks

```

4148 +
    68 * array-length(fieldvalue(m_userData)) * array-length(fieldvalue(children)) +
    32 * array-length(fieldvalue(m_userData)) * array-length(fieldvalue(m_userData)) +
    515 * array-length(fieldvalue(children)) +
    136 * array-length(fieldvalue(m_userData))

```

which consists of the propagated cost values of (among others):

```

c.SchedulingSandbox.sort:
    179 +
    32 * array-length(var1) * array-length(var1) +
    68 * array-length(var2) * array-length(var1) +
    68 * array-length(var1) +
    68 * array-length(var2)

c.SchedulingSandbox.makeindex:
    209 + 60 * array-length(fieldvalue(children))

cd.TempIndexNode.add:
    1221 + 129 * array-length(fieldvalue(children))

```

Computing Timing Cost of Library Functions. We created some infrastructure to facilitate the cost analysis of library functions. We have used this to add computed method costs to some

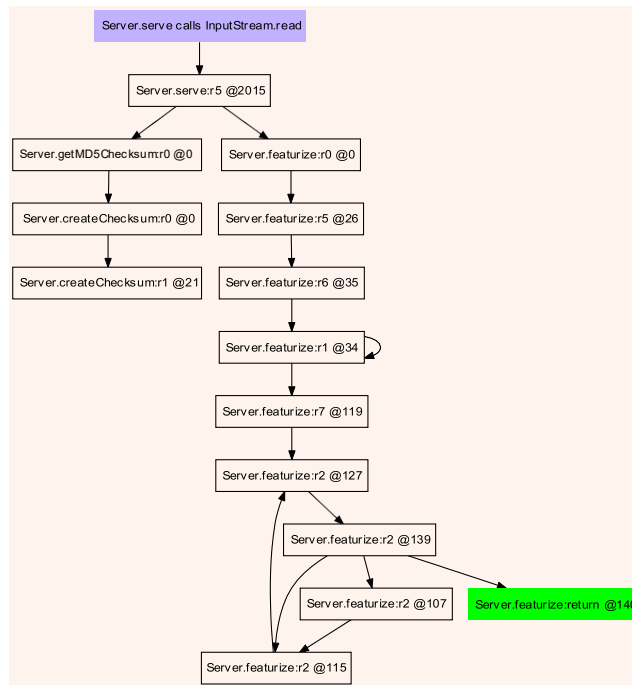


Figure 7: Malware Analyzer: taint propagation from source 3: `InputStream.read` called from `Server.serve` (pc=2023)

Cost Analysis: Framework

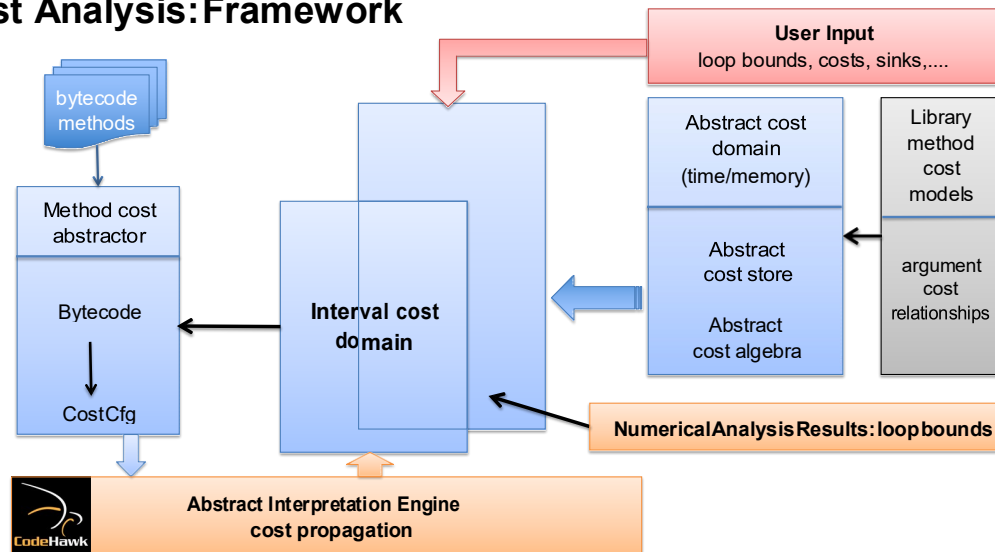


Figure 8: Cost Analysis Framework

library functions for which we do not yet have profiling information. This is especially advantageous where the timing cost is dependent on the values of one or more arguments, which is more difficult to obtain via profiling. Some of the results obtained are shown in Table 2. Some of these results rely on the timing values of individual byte code instructions only (see Section 3.1.5) while others also depend on profiling results for other library functions (such as the value for `java.lang.IllegalArgumentException.<init>(java.lang.String)`). In particular, if library functions have any callees or descendant callees that are native functions, timing costs can only be bootstrapped with either timing values for those native functions (which is hard, because they are often not public methods) or the timing value of a function higher up in the call stack.

Side-channel Checks. The cost model provides the option for the user to specify observation points in the form of sources and sinks. The model is then used to compute the individual path costs for all paths connecting a given source to a given sink. It also enumerates, for every path, the conditional expressions encountered along that path. For example, for the canonical example Category 3, the following is computed for the `checkSecret` method:

```
Category3_vulnerable.checkSecret
decision-pc    : 0
observation-pc : 125
conditions at decision-pc:
  T: var0>1234
  F: var0<=1234
```

path through pc=	cost
12:	1,000,010
42:	21,000,314
64:	8,001,112,034
72:	1,000,010
103:	401,005,634
108:	8,001,112,034

Paths through 12

```
[0, 7, 12, 125]
  7: var1==1
```

Paths through 42

```
[0, 7, 19, 24, 26, 32, 26, 42, 125]
  7: var1!=1
 19: var1==2
 26: L:var2<20 (bound: 21)
 26: var2>=20
```

Paths through 64

```
[0, 7, 19, 45, 47, 54, 47, 64, 125]
  7: var1!=1
 19: var1!=2
 47: L:var2<8000 (bound: 8001)
 47: var2>=8000
```

Table 2: Some Cost Model Predictions for the Timing Cost of Library Functions

method	cost expression (ns)
java.io.File boolean equals(java.lang.Object)	125
java.io.PrintStream println(java.lang.String)	$22,025 + (46 \times \text{len}(\text{arg}_1))$
java.lang.Boolean boolean equals(java.lang.Object)	150
java.lang.IllegalArgumentException <init>(java.lang.String)	2500
java.lang.Integer boolean equals(java.lang.Object)	160
java.lang.Long boolean equals(java.lang.Object)	180
java.lang.Math double sqrt(double)	15
java.lang.Object <init>()	143
java.lang.StringBuilder <init>() <init>(java.lang.String) <init>(int) java.lang.String toString() int length() java.lang.StringBuilder append(java.lang.String) java.lang.StringBuilder append(char) java.lang.StringBuilder append(boolean)	$ \begin{aligned} &576 \\ &699 + \frac{2 \times \text{len}(\text{arg}_1)}{3} \\ &900 + \frac{\text{arg}_1^8}{1} \\ &982 + \frac{\text{len}(\text{this})^4}{2} \\ &156 \\ &55 \\ &538 + \frac{14 \times \text{len}(\text{this})}{10000} \\ &602 + \frac{15 \times \text{len}(\text{this})}{10000} \end{aligned} $
java.lang.Thread void sleep(long)	$1,000,000 \times \text{arg}_1$
java.net.InetAddress boolean equals(java.lang.Object)	180
java.net.InetSocketAddress boolean equals(java.lang.Object)	493
java.util.Arrays static void fill(boolean[],boolean)	$257 + \frac{\text{len}(\text{arg}_1)}{5}$
java.util.logging.Level boolean equals(java.lang.Object)	12,400

```

Paths through 72
[0, 67, 72, 125]
67: var1==1
Paths through 103
[0, 67, 79, 84, 86, 93, 86, 103, 125]
67: var1!=1
79: var1==2
86: L:var2<400 (bound: 401)
86: var2>=400
Paths through 108
[0, 67, 79, 106, 108, 115, 108, 125]
67: var1!=1
79: var1!=2
108: L:var2<8000 (bound: 8001)
108: var2>=8000

```

3.1.4 Memory Cost. Most of our work so far was directed at computing timing cost. Although the cost framework created is sufficiently general that it can be used for any type of resource cost, the actual application of the framework is still dependent on the availability of the “raw data”, such as profiling data in case of timing cost. In some respects obtaining raw data for memory analysis is more straightforward (and less platform dependent) as memory requirements can be deduced from the size of types, and arguments like `initialCapacity` to the constructor of `java.util.ArrayList`, the computation of memory in use by propagation is complicated by having to keep track of how long memory stays in scope and when it can be garbage collected.

We put in place some of the basic components for memory analysis, such as adding the facility to specify memory requirements as part of library function summaries (currently we have specified relationships between memory requirements and argument values for some 100 JDK library functions). Furthermore, as an aid to the human analyst we have provided python reports that compute and report the sizes in bytes of all application classes and indications whether fields point to potential other memory resources, described in Section 3.1.7.

Further research and development is required to be able to perform a resource requirement propagation for memory similar to that for timing cost.

3.1.5 Cost Profiling. Profiling is performed by collecting the execution time of many of the executions of the object of interest and taking the average of these times as the result. Below we describe the approach as applied to library methods and to bytecode instructions.

Library Methods. For each library method we wish to profile we produce a custom profiling method. This method must be able to call the library method repeatedly, collect the execution time of each run, and pass arguments to the method as needed. The average execution time is taken as a basis for further analysis.

We use `System.nanoTime` to get a point in time. In the case of library methods we simply make calls to `System.nanoTime` immediately before and after executing the method. It is important to recognize that a call to `System.nanoTime` actually takes place over a range of time and returns some time from within this range. To attempt to account for this variability we profile a

separate loop containing only two immediately sequential calls to `System.nanoTime`. This value is subtracted from the total value obtained by the profiling method, eliminating the portion of the measured time that bookends the library call of interest but is not part of it.

We have generated an infrastructure, written in Java that provides utilities to generate both the skeleton of a profiling method and the glue code needed to integrate the profiling method with existing methods. While this does not yet allow to generate a profiling method fully automatically, with a little experience profiling methods and information can be generated quite quickly.

The infrastructure provides facilities to automatically profile a method against many different input, in either one or more variables, to collect the outputs, and to perform a regression over the outputs to establish an execution time as a linear expression of the method's input variables. The Java source code for this infrastructure is available open-source in the

`kestreltechnology/CodeHawk-Java-Platform-Summaries`

GitHub repository

Byte code. We use similar techniques to profile individual byte codes. For this case, however, we must take additional care to account for the extremely short execution times of most byte codes. We write a profiling method similar to those used for library methods, but in this case call `System.nanoTime` only twice in the exterior of the loop to ensure that the timing characteristics of the calls to `System.nanoTime` do not totally obscure the execution time of the byte code.

We measure the entire execution of the loop, including the control flow and body of the loop. To account for the contribution of the loop's control flow to the timing of the byte code, we separately measure a loop entirely devoid of instructions, and subtract the execution time of this loop from the execution time of our profiling loop.

Finally, the portion of the loop contained within the timing measurement may contain additional byte codes, either by necessity, or as an artifact of the code chosen to exercise the byte code instruction. In this case we do a separate analysis of the dependent byte code instruction when possible and subtract this time from the total time. If separating the byte codes is not possible we use a different strategy to determine an exact time, as described below in Section 3.1.5

Challenges. Some pairs of byte codes always appear together, so in general it is not possible to isolate each byte code from all others for observation. For these cases the techniques described above are not able to precisely determine a timing for each individual byte code.

Examples of such pairs are

- Byte codes that load from a local variable and store to a local variable, e.g., `lload` and `lstore`;
- Byte codes that push a constant onto the stack and store to a local variable, e.g., `lconst_0` and `lstore`;
- Byte codes that invoke a method and return a value, e.g., `invokestatic` and `areturn`

For the first two cases we simply split the execution time of the pair of byte codes and assign half to each. In all cases this results in an execution time of about 2ns for each byte code. As the

byte codes always appear together this should not introduce any significant inaccuracies for the computed execution time of a method containing them.

For the third case we observe two useful subcases. First, the execution time of an `invoke` instruction and the corresponding `return` instruction (which returns `void`) shows an execution time of at least 12 ns. Otherwise, an `invoke` instruction and any other `return` instruction (`areturn`, `dreturn`, `freturn`, `ireturn`) shows an execution time of at least 160 ns. We therefore take `invoke` instructions to have an execution time of at least 6ns and `return` instructions to have an execution time of either 6ns (for `return`) or 154ns for all other `return` instructions. In all our tests the `invoke` instructions are calling a very small method, so we take 6ns to be only the lower bound on such instructions. More work is required to obtain better execution times for these instructions within more environments.

3.1.6 Library Summaries. Java applications typically make heavy use of library functions, foremost JDK library functions, but increasingly also functions from third-party libraries such as the Apache libraries (e.g., `commons-math`) or the Google serialization `protobuf` libraries. In particular, the STAC Engagements made use of dozens of different third-party libraries. Both taint and timing cost analysis crucially depend on knowledge of the contributions made by library functions, albeit in very different ways.

During the IARPA StoneSoup project we developed an infrastructure and representation to express summary information for JDK library functions. The information included:

- **preconditions:** safety conditions on arguments that guarantee that a particular exception is not raised;
- **postconditions:** relations on the method return value and argument values guaranteed to hold for every invocation;
- **taint:** transfer relations between arguments and between arguments and return value, or external introduction of taint on arguments or return value, for methods that receive external inputs;
- **string sinks:** string arguments that require a particular format (e.g., a class name or a number) and/or are used for a particular purpose (e.g., pattern matching or dynamic loading).

This information about methods was obtained from the Oracle JDK Application Programming Interface (API) documentation, that is, all of it should be valid across all JDK implementations. While templates for summaries including signature and checked exceptions were generated automatically from some `rt.jar` file, the summary information described above was manually added. During the StoneSoup project some 1000 classes with more than 10,000 methods were summarized in this way. It should be noted however, that preconditions are necessarily incomplete, that is, many potential exceptions raised are not guarded by preconditions, due to both incomplete documentation with respect to runtime exceptions (or deliberately underspecified methods, leaving the choice to the implementation) or simply because of the wide variety of conditions that can give rise to exceptions being raised that our analysis does not cover, and thus for which we don't provide predicate expressions. A similar situation exists for postconditions. In this case, however, omission of a postcondition merely reduces analysis precision, as the default postcondition of `true` (that is,

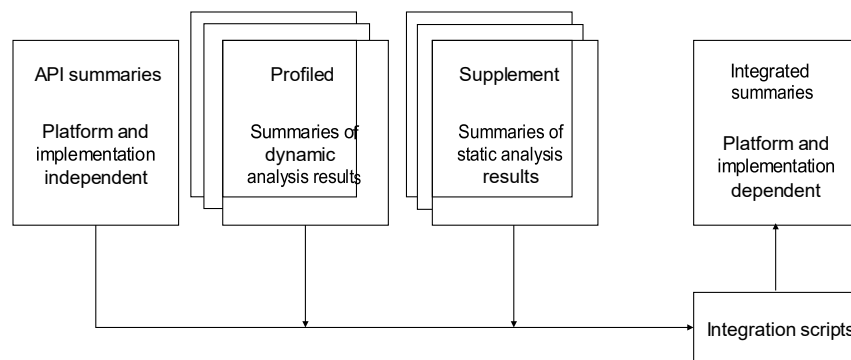


Figure 9: Platform-Dependent Summary Creation Procedure

the return value can have any value allowed by its type), is always a valid over-approximation of the actual value, and thus does not invalidate the analysis results, as potentially missed preconditions could. For taint transfer our goal has always been to provide an over-approximation of the transfer and introduction of taint, which, unlike postconditions, always requires explicit specification.

Platform Independence. Execution time is an inherently platform and implementation-dependent property of a method. It is platform dependent as the timing of individual instructions depends on the machine the program is running; it is implementation dependent as the same functionality can be implemented in many different ways with very different performance characteristics. Hence we could not simply add timing costs to our existing summaries without instantly making those summaries platform and implementation dependent, an undesirable consequence.

To address this issue we separate the information obtained from different sources into separate summaries, as shown in Figure 9. API summaries contain information that is obtained from the API documentation and should be valid across all implementations and platforms. Profiled summaries contain information obtained by dynamic analysis of a particular implementation of a given library on a particular platform. This includes timing cost as obtained by profiling (see Section 3.1.5). Supplement summaries provide information that may have been obtained by analysis, e.g., we can run our cost analysis on a particular implementation of a library method, or we can perform taint analysis, or we may simply add information gleaned by inspection of the implementation of library functions, such as the use of private fields that relate directly to particular method arguments, and thus can make the taint analysis more precise. Thus, as shown in Figure 9 we have a single API summary and we may have multiple profiled and supplement summaries from which we can choose to construct an integrated summary that is then to be used in the analysis of the target application. To facilitate rapid update of summaries the integration process is automated using python scripts.

3.1.7 Python API. The CodeHawk Java Analyzer uses python scripts both to set up and perform the analysis and to inspect the results. Taking advantage of the uniform structure of the challenge problems throughout all Engagements, we collected all of the relevant information regarding any particular Challenge problem, such as which libraries it used, and where those could be found, in a json file, such that analyses could be started by just specifying the name of the Challenge

problem, and all other information would be obtained from these json data files. Scripts to inspect the analysis results are written against an extensive python API that extracts the encoded analysis results from file and make it available via object accessors.

Analysis Setup

- **analyze.** Invoked with the name of one of the Engagement problems this script will retrieve the names third-party libraries on which the application depends, determine if summaries exist for these libraries, and invoke the CodeHawk Java Analyzer on the application jar(s) passing the paths to the relevant summary jars. The analyzer performs the numeric analysis and saves the results in xml files.
- **analyze_taint.** Invoked with the name of one of the Engagement problems this script will invoke the CodeHawk taint analysis, passing in again any dependent library summaries. The analyzer performs the first pass of the taint analysis, using the results saved by the numeric analysis to ensure variables within a loop do not unnecessarily get tainted by the loop bound if the loop in fact has a constant bound. The resulting local taint dependency graphs are saved in xml files, including all taint sources encountered in the application.
- **analyze_taint_propagation.** Invoked with the name of one of the Engagement problems and the id of a taint source, this script will invoke the CodeHawk taint analysis, which performs the second pass of the taint analysis restricted to the selected taint source only, using the saved taint graphs from the first pass. The results are again saved in xml files.
- **analyze_cost.** Invoked with the name of one of the Engagement problems this script will invoke the CodeHawk time cost analysis, passing in the paths to the relevant third party libraries, which may provide some cost information for library methods. The analyzer creates a cost model for all methods in the application itself, using the results of the numeric analysis to determine lower and upper bounds on loop iterations, and using costs provided by the platform-dependent library summaries, if available. If not available, computations are performed using symbolic costs. The results are stored in xml files.
- **scan_application.** Invoked with the name of one of the Engagement problems this script will invoke the CodeHawk byte-code parser to give a quick indication of the number of classes and methods contained in the application, and the library functions referenced. No results are saved.
- **translate_application.** Invoked with the name of one of the Engagement problems this script will direct the CodeHawk Analyzer to parse the application and translate it to CHIF, to quickly identify any potential problems. No results are saved.

Inspection of Primary Results

- **report_loops.** Invoked with the name of an Engagement problem this script will retrieve the results produced by the numeric and taint analysis and list all of the loops in the application, together with their nesting depths, loop bounds, if available, and whether the loop bounds are tainted. Methods are identified by name and id.

- **report_taint_origins.** Invoked with the name of an Engagement problem this script will retrieve the results of the taint analysis and list all of the taint sources encountered in the application. Taint sources are identified by name and id. The id of a taint source of interest is to be used in the `analyze_taint_propagation` script described above.
- **report_taint_trail.** Invoked with the name of an Engagement problem and a taint source id, this script will retrieve the results of the second-pass taint analysis performed for this taint source and create a graphviz dot file and a pdf file that show all of the program constructs that participate in the transfer of taint from the selected source. These files are saved in the same directory as the application.
- **report_costmodel.** Invoked with the name of an Engagement problem this script will retrieve the results of the cost analysis and lists CPU time cost bounds for all application methods divided into three groups:
 - methods with constant cost;
 - methods with a cost range with constant lower and upper bound; these typically are methods that have multiple paths; the methods are ordered by increasing ratio between upper and lower bound;
 - methods for which the cost expression includes symbolic cost values such as cost values for methods called that do not have a constant or constant range cost, or loop counters for which no bounds are available.

Various Informational Queries.

- **report_branchconditions.** Invoked with the name of an Engagement problem this script prints an alphabetical list of all branch conditions in the application. The purpose of this script is to identify potentially unusual conditions that may point at side-channel vulnerabilities.
- **report_class_methodcalls.** Invoked with the name of an Engagement problem and a fully qualified classname this script prints out a list of all calls made that target a method of that class. This can aid in assessing whether a particular class contributes to a resource problem, e.g., to identify methods that add entries to container classes.
- **report_exceptionhandlers.** Invoked with the name of an Engagement problem this script will list all exception handlers, the extent of the try blocks they cover and they type of exception they handle.
- **report_loaded_strings.** Invoked with the name of an Engagement problem, and optionally a substring the string should contain, this script will list per methods all strings referenced in that method. This list can give an analyst a quick impression of what the application is all about and what happens where, as strings are often used in prompts, error messages, but also regular expressions or loaded class names may stand out. For example the following query gives a quick result for potential regular expressions:

```
> python chj_report_loaded_strings.py textcrunchr_1 --substring +
.....
WordFrequencyProcessor.tokenize
  9  [^\p{Alnum}]+
WordStatsProcessor.tokenize
  0  [^\p{Alnum}]+
```

- **report_named_method_calls.** Invoked with the name of an Engagement application and a method name this script lists all calls and their locations of methods with the given name. This can identify calls like `put` and `add` on container classes that may contribute to excessive memory usage.
- **report_object_field_writes.** Invoked with the name of an Engagement application this script will list all of the reads and writes to all object fields in the application.
- **report_object_sizes** Invoked with the name of an Engagement application this script will list for all classes used in the application what their scalar size is (that is, the memory, in bytes, allocated when they are created) as well as the references they contain to other objects, e.g., library objects such as lists or hash tables.
- **report_objects_created.** Invoked with the name of an Engagement application this script will list all objects created by the application and the locations where they are created. This information gives insight into the memory consumption of different parts of the application.
- **report_reflective_calls.** Invoked with the name of an Engagement application this script will list all calls made using reflection. This information is useful to determine if methods may be called indirectly.

```
> python chj_report_reflective_calls.py blogger
.....
JavaWebServer.serve
  23 invokeinterface    fi.iki.elonen.NanoHTTPD$IHTTPSession.getMethod: ()fi.iki.elonen.NanoHTTPD$Method

Example.render
  12 invokeinterface    fi.iki.elonen.NanoHTTPD$IHTTPSession.getMethod: ()fi.iki.elonen.NanoHTTPD$Method

Next.render
  1 invokeinterface     fi.iki.elonen.NanoHTTPD$IHTTPSession.getMethod: ()fi.iki.elonen.NanoHTTPD$Method

Write.render
  12 invokeinterface     fi.iki.elonen.NanoHTTPD$IHTTPSession.getMethod: ()fi.iki.elonen.NanoHTTPD$Method

JavaWebServerPlugin.serveFile
  32 invokestatic       java.lang.Class.forName: (java.lang.String)java.lang.Class

JavaWebServerPlugin.serveFile
  129 invokeinterface    fi.iki.elonen.NanoHTTPD$IHTTPSession.getMethod: ()fi.iki.elonen.NanoHTTPD$Method

Login.render
  1 invokeinterface     fi.iki.elonen.NanoHTTPD$IHTTPSession.getMethod: ()fi.iki.elonen.NanoHTTPD$Method

NanoHTTPD.serve
  9 invokeinterface     fi.iki.elonen.NanoHTTPD$IHTTPSession.getMethod: ()fi.iki.elonen.NanoHTTPD$Method

SimpleWebServer.serve
  40 invokeinterface     fi.iki.elonen.NanoHTTPD$IHTTPSession.getMethod: ()fi.iki.elonen.NanoHTTPD$Method
```

```

1 void driver(@Inv("true") List<String> input) {
2   @Guarantee("len(blogDB)<2")
3   @Inv("len(blogDB)=c8-c10") List<String> blogDB = new List<String>();
4   @Inv("iterOf(input)") Iterator<String> it = input.iterator();
5   String blog;
6   while(*) {
7     blog = it.next();
8     c8:   postNewBlog(blog, blogDB);
9     c9:   showBlog(blogDB);
10    c10:  deleteBlog(blogDB);
11  } }
12 @Summary{"len(blogDB')=len(blogDB)+1"}
13 void postNewBlog(String blog, List<String> blogDB) { //callback: add post
14   blogDB.add(blog);
15 }
16 @Summary{"len(blogDB')=len(blogDB)-1"}
17 void deleteBlog(List<String> blogDB) { //callback: delete last post
18   blogDB.remove();
19 }
20 @Summary{"len(blogDB')=len(blogDB)"}
21 void showBlogs(@Inv("true") List<String> blogDB) {
22   @Guarantee("len(toShow)<=len(blogDB)+2")
23   //callback: display blog contents
24   @Inv("len(toShow)-idx(it)=c28+c30+c33-c32") List<String> toShow = new List<String>();
25   @Inv("iterOf(blogDB)") Iterator<String> it = blogDB.iterator();
26   String blog;
27   blog = "Welcome!\n";
28   c28: toShow.add(b);
29   blog = "Blog begins:\n";
30   c30: toShow.add(b);
31   while(*) {
32     c32: blog=it.next();
33     c33: toShow.add(blog);
34   }
35 // render toShow as an HTML page
36 }

```

Figure 10: Motivating Example: a Simplified Version of a Logging Server

- **report_static_field_initializers.** Invoked with the name of an Engagement application this script lists for all static fields the byte-code instructions that write to those fields with, if available, the value written. This can be useful to determine, e.g., if a static field, which can be considered a global variable, is ever changed after initialization by the class initializer.

3.2 PATHR: A precise path- and context-sensitive static analysis

3.2.1 Quantm: Type-directed Bounding of Collections in Reactive Programs. We demonstrate our approach for verifying the total collection lengths for reactive programs on a motivating example in Figure 10. Overall, a user interacts with our tool Quantm as follows. First, they write a driver that encodes a particular usage pattern that they are interested in. Then they specify invariants as type annotations. After these two steps, our type system will take care of the rest by automatically checking if the invariant relations are valid. If the invariants are indeed valid, Quantm will automatically discharge a query to an off-the-shelf SMT solver, returning the result “verified” or “not verified”. The “verified” answer is conclusive, as our method is sound. The “not verified” is inconclusive: either the bound does not hold, or the user has not provided sufficient invariants to answer the verification problem.

We simplified code from a Java web server based on the Spring framework that allows users to upload a blog post, delete a blog post and render a list of posts as an html page. Callback methods `postNewBlog`, `deleteBlog`, and `showBlogs` implement these functionalities. Method `driver` encodes an infinite input sequence that a user of our tool is interested in: it first reads a blog from input and appends it to the database, then renders the blog as an HTML page, and finally removes the blog from database. Our goal is to verify the boundedness of total collection lengths in every method separately, when input variables satisfy given constraints (e.g., inputs can have upper bounds on their length). In particular, callback methods `postNewBlog` and `deleteBlog` do not declare collection-typed variables and therefore they are vacuously bounded. More interestingly, we would like to verify the following bounding predicates denoted by `@Guarantee` in Figure 10:

- The length of collection variables in method `driver` is less than 2, i.e. $\text{len}(\text{blogDB}) < 2$
- Total length of collection variables in method `showBlogs` is less than or equal to length of input variable `blogDB`, i.e. $\text{len}(\text{toShow}) \leq \text{len}(\text{blogDB}) + 2$

We emphasize that our approach is able to verify above bounds when there exist neither time bounds nor input bounds, because input variables `input` and `blogDB` have no constraint at all, i.e. a `true` constraint. The notation `@Inv` in Figure 10 denotes a refinement type. The content inside the brackets following `@Inv` is the refinement of that particular type. For example, $\text{len}(\text{blogDB}) = c8 - c10$ is a type refinement on variable `blogDB`.

We now explain the role of the AST counters in the invariant. For example, if we look at the inner loop at line 31-34 in Figure 10, the property we most likely need for `list toShow` is $\text{len}(\text{toShow}) \leq \text{idx}(it) + 2$, where $\text{idx}(it)$ represents the number of elements that has been visited using iterator `it`. However, this property is actually not inductive because it breaks after line 28 (as well as line 30), as $\text{len}(\text{toShow})$ is incremented by 1 but nothing else is updated in the invariant. However, we can add AST counters to the invariant, and obtain $\text{len}(\text{toShow}) - \text{idx}(it) = c28 + c30 + c33 - c32$. We thus obtain an inductive invariant that is then used as the type of `toShow`.

The purpose of these counters is to enable writing expressive invariants. The interesting invariants usually do not depend on the value of the counters (the value grows without bound for nonterminating programs), just on relations between counters of different AST nodes. These could be seen on the example in the previous section.

As another example, consider how we reason about the non-terminating loop at line 6-11, we first summarize the effects of callback `postNewBlog` and `deleteBlog` on any collection variable passed in as argument(s), which is to add 1 element to or remove 1 element from list `blogDB`. Method summaries are automatically applied at invocation sites. Next, since we have AST counters, we are now able to easily define the length of variable `blogDB` as an inductive invariant $\text{len}(\text{blogDB}) = c8 - c10$ (shown at line 2) that hold at before and after every execution step. Note that this invariant serves as a safety property of all program states under the existence of non-terminating executing traces, which is the root cause of the mainstream approach in resource bound analysis to fail under the scenario of reactive programs.

Inside Quantm. Our type system is based on Liquid types [32], where the refinements can express facts about collections and AST counters. Our type checking rules are standard, with added rules that capture the semantics of collections (lists) and counters. Constraints on AST counters are generated from the Abstract Syntax Tree structure of the program. We formalize this and other relations on counters in a set of axioms. We generate verification conditions that ensure that the

AST counters are used consistently, and that the types imply the given bound. For instance, now that we have invariants describing lengths of list `blogDB` and `toShow` in the method `showBlogs`, we can plug in counter axioms and check the required implications. For instance, the type of `toShow` is $\text{len}(\text{toShow}) - \text{idx}(it) = c28 + c30 + c33 \leq 32$. From the counter axioms, we have that $c28 \leq 1$ (as the corresponding statements are executed once at most) and $(c32 - c33 \leq 1)$ (as the corresponding statements are sequentially executed). We then use an off-the-shelf SMT solver to check that the inductive invariant and the counter axioms imply the guarantee that the user specified: $\text{len}(\text{toShow}) \leq \text{len}(\text{blogDB}) + 2$.

3.2.2 Themis: Precise Detection of Side-Channel Vulnerabilities. We give an overview of our technique with the aid of a motivating example and explain the threat model.

Motivating Example. Suppose that Bob, a security analyst at a government agency, receives a Java web application called Gabfeed, which implements a web forum that allows community members to post and search messages. In this context, both the user names and passwords are considered confidential and are therefore encrypted before being stored in the database. Bob’s task is to vet this application and verify that it does not contain timing side-channel vulnerabilities that may compromise user name or password information. However, Gabfeed contains around 30,000 lines of application code (not including any libraries); hence, manually searching for a vulnerability in the application is akin to finding a needle in the haystack.

A security analyst like Bob can greatly benefit from Themis by using it to automatically verify the absence of side-channel vulnerabilities in the target application. To use Themis, Bob first identifies application-specific confidential data (in this case, `secretKey`) and annotates them as such in a Themis-specific configuration file, as shown in Figure 12. In the same configuration file, Bob also tells Themis the type of side channel to look for (in this case, timing) by specifying the `costModel` field and provides a reasonable value of ϵ , using the `epsilon` field. Here, Bob wants to be conservative and initially sets the value of ϵ to zero.

Using the information provided by Bob in the configuration file, Themis first performs static taint analysis to identify methods that are dependent on confidential data. In this case, one of the methods that access confidential data is `modPow`, shown in Figure 11. Specifically, Themis determines that the second argument (exponent) of `modPow` is tainted and marks it as a “hot spot” that should be analyzed more precisely using relational verification techniques.

In the next phase, Themis uses its QCHL verifier to analyze `modPow` in more detail. Specifically, the QCHL verifier considers two executions of `modPow` that have the same values of base and modulus but that differ in the value of exponent. In this case, the QCHL verifier fails to prove that the resource usage of any such two runs is identical and therefore issues a warning about a possible timing side channel in the `modPow` procedure.

Next, Bob wonders whether the imbalance in resource usage is large enough to be actually exploitable in practice. For this reason, he plays around with different values of the bound ϵ , gradually increasing it to larger and larger constants. In the case of timing side channels, ϵ represents the difference in the executed number of Java bytecode instructions. However, no matter what value of ϵ Bob picks, Themis complains about a possible timing side channel. This observation indeed makes sense because the difference in resource usage is proportional to the secret and can therefore not be bounded by a constant.

Bob now inspects the source code of `modPow` and realizes that a possible vulnerability arises

```

1  BigInteger modPow(BigInteger base,
2      BigInteger exponent, BigInteger modulus) {
3      BigInteger s = BigInteger.valueOf(1);
4      // BigInteger r;
5      int width = exponent.bitLength();
6      for (int i = 0; i < width; i++) {
7          s = s.multiply(s).mod(modulus);
8          if(exponent.testBit(width - i - 1))
9              s = s.multiply(base).mod(modulus);
10         //else r = s.multiply(base).mod(modulus);
11     }
12     return s;
13 }

```

Figure 11: Gabfeed Code Snippet that Contains a Timing Side Channel

due to the resource imbalance in the secret-dependent branch from line 8. To fix the vulnerability, Bob adds the code from lines 4 and 10, with the goal of ensuring that the timing behavior of the program is not dependent on exponent. To confirm that his fix is valid, Bob now runs Themis one more time and verifies that his repair eliminates the original vulnerability.

Threat Model. We assume that an adversary can observe a program’s total resource usage, such as timing, memory, and response size. When measuring resource usage, we further assume that any variations are caused at the application software level. Hence, side channels caused by the microarchitecture such as cache contention and branch prediction are out of the scope of this work. Physical side channels can, in principle, be handled by our system as long as a precise model of the corresponding resource usage is given. We assume that the attacker is not able to observe anything else about the program other than its resource usage.

One possible real-world setting in which the aforementioned assumptions hold could be that the attacker and the victim are connected through a network, and the victim runs a server or P2P software that interacts with other machines through encrypted communications. In this scenario, the attacker and the victim are physically separated; hence, the attacker cannot exploit physical side channels, such as power usage. Furthermore, the attacker does not have a co-resident process or VM running on the victim’s machine, thus it is hard to passively observe or actively manipulate OS and hardware-level side channels. What the attacker can do is to either interact with the server and measure the time it takes for the server to respond, or observe the network traffic and measure request and response sizes. In our setting, we assume that data encryption has been properly implemented and the attacker cannot directly read the contents of any packet.

System Design and Implementation. We next discuss the design and implementation of Themis, our end-to-end static analysis tool for verifying bounded non- interference. While the QCHL verifier is one of the key components of Themis, it is not necessary (and also not scalable) to employ such precise relational reasoning throughout the entire program. Hence, as mentioned earlier, our

```

1  {
2      "epsilon": "0",  "costModel": "time",
3      "secrets": [ "<com.cyberpointllc.stac.auth.
4                  KeyExchangeServer:java.math.BigInteger
5                  secretKey>" ]
6  }

```

Figure 12: Themis Configuration file for Gabfeed

approach employs taint analysis to identify program parts requiring precise analysis.

Figure 13 gives a high-level schematic overview of Themis’s architecture. In addition to the QCHL verifier discussed in detail in [11], Themis also incorporates pointer and taint analyses and instruments the program to explicitly track resource usage. We now give a brief overview of each of these components.

Pointer analysis. Given the byte code of a Java application, Themis performs (field- and object-sensitive) pointer analysis to build a precise call graph and identify all variables that may alias each other. The resulting call graph and alias information are used by the subsequent taint analysis as well as the QCHL verifier.

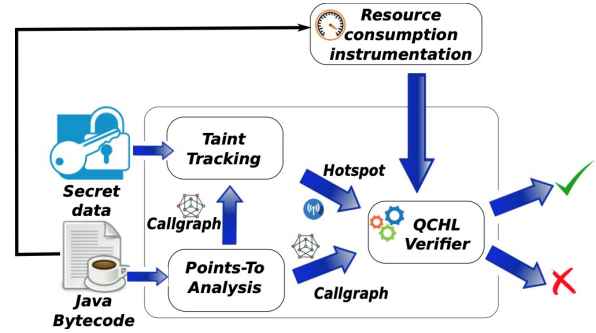


Figure 13: Workflow of the Themis tool

Taint analysis. The use of taint analysis in Themis serves two goals: First, the QCHL verifier uses the results of the taint analysis to determine whether two loops can be synchronized. Second, we use taint analysis to identify hotspots that need to be analyzed more precisely using the QCHL verifier. The taint analyzer uses the annotations in Themis’s configuration file to determine taint sources (i.e., high inputs) and propagates taint using a field- and object-sensitive analysis. Our taint analyzer tracks both explicit and implicit flows. That is, a variable v is considered tainted if (a) there is an assignment $v := e$ such that e is tainted (explicit flow), or (b) a write to v occurs inside a branch whose predicate is tainted (implicit flow).

We use the results of the taint analysis to identify methods that should be analyzed by the QCHL verifier. A method m is referred to as hot spot if it reads from a tainted variable. We say that a hot spot m dominates another hot spot m' if m' is a transitive callee of m but not the other way around. Any hot spot that does not have dominators is given as an entry point to the QCHL verifier. In principle, this strategy of running the QCHL verifier on only hot spots can cause our analysis to report false positives.

Resource usage instrumentation. The language we considered for our formalization [11] is equipped with a `consume(x)` statement that models consumption of x units of resource. Unfortunately, since Java programs do not come with such statements, our implementation uses a cost model to instrument the program with such consume statements.

In principle, our framework can detect different classes of side channels, provided that the tool is given a suitable cost model for the corresponding resource type. Our current implementation provides cost models for two kinds of resource usage, namely, timing and response size. For timing, we use a coarse cost model where every byte code instruction is assumed to have unit cost.

For response size, each string s that is appended to the response consumes $s.length()$ units.

Counterexample generation. If Themis fails to verify the bounded non-interference property for a given ϵ , it can also generate counterexamples by using the models provided by the underlying SMT solver. In particular, when the verification condition (VC) generated by Themis is invalid, the tool asks the SMT solver for a falsifying assignment and pretty-prints the model returned from Z3 [33] by replacing Z3 symbols with their corresponding variable names. Since the VCs depend on automatically inferred loop invariants, the counterexamples generated by Themis may

be spurious if the inference engine does not infer sufficiently strong loop invariants.

System Implementation. The Themis system is implemented in a combination of Java and OCaml and leverages multiple existing tools, such as Soot [34], Z3 [33], and Apron [35]. Specifically, our pointer analysis builds on top of Soot [34] and we extend the taint analysis provided by FlowDroid, which is a state-of-the-art context-, field-, flow-, and object-sensitive taint analyzer, to also track implicit flows. Our QCHL verifier is implemented in OCaml and uses the Z3 SMT solver to discharge the generated verification conditions. To prove the Hoare triples that arise as premises in the QCHL proof rules, we perform standard weakest precondition computation, leveraging the Apron numerical abstract domain library to infer standard loop invariants. Recall that we infer relational loop invariants using the monomial predicate abstraction technique described in [11]. Our formal description of QCHL uses a simplified programming language that does not have many of the complexities of Java. Themis handles these complexities by first leveraging the Soot framework [34] to parse the Java bytecode to Soot IR, and then using an in-house “front-end” that further lowers Soot IR into a form closer to what is presented in [11]. In particular, the transformation from Soot to our IR recovers program structures (loops, conditionals etc.) and encodes heap accesses in terms of arrays. The verifier performs strongest postcondition calculation over our internal IR and encodes verification conditions with SMT formulae. In the remainder of this section, we explain how we handle various challenges that we encountered while building the Themis frontend.

- **Object encoding.** Since objects are pervasive in Java applications, their encoding has a significant impact on the precision and scalability of the approach. In Themis, we adopt a heap encoding that is similar to ESC-Java. Specifically, instance fields of objects are represented as maps from object references (modeled as integer variables) to the value of the corresponding field. Reads and writes to the map are modeled using select and update functions defined by the theory of arrays in SMT solvers. If two object references are known not to be the same (according to the results of the pointer analysis), we then add a disequality constraint between the corresponding variables.
- **Method invocation.** Since the simplified language from [11] did not allow function calls, we only described an intraprocedural version of the QCHL verifier. We currently perform interprocedural analysis by function inlining, which is performed as a preprocessing step at the internal IR level before the analysis takes place. Since the QCHL verifier only needs to analyze hot spots (which typically constitute a small fraction of the program), we do not find inlining to be a major scalability bottleneck. However, since recursive procedures cannot be handled using function inlining, our current implementation requires models for recursive procedures that correspond to hot spots.
- **Virtual calls and instanceof encoding.** The result of certain operations in the Java language, such as virtual calls and the instanceof operator, depends on the runtime values of their operands. To faithfully model those operations, we encode the type of each allocation site as one of its field, and we transform virtual calls and instanceof to a series of if statements that branch on this field. We handle the instanceof operator in a similar way.

```

1 public class RegexValidator {
2     boolean validEmail(String t) { return t.matches(".*@.*\\.([a-z]+)"); }
3     boolean validComment(String t) {
4         return !t.matches("(\\p{Blank}*\\(\\r?\\n\\)\\p{Blank}*)+"); }
5     boolean safeComment(String t) { return t.matches("[^\\/<>]+"); }
6     boolean validUrl(String t) {
7         return t.matches("www\\.shoppers\\.com/.+/.+/.+/.+"); }
8 }
9 public class CommentFormValidator implements Validator {
10     private Admin admin;
11     public void validate(CommentForm form, Errors errors) {
12         String senderEmail = form.getSenderEmail();
13         String productUrl = form.getProductUrl();
14         String comment = form.getComment();
15         if (!RegexValidator.validEmail(admin.getEmail())) return;
16         if (senderEmail.length() <= 254) {
17             if (RegexValidator.validEmail(senderEmail)) ... }
18         if (productUrl.split("/").length == 5) {
19             if (RegexValidator.validUrl(productUrl)) ... }
20         if (RegexValidator.safeComment(comment)) {
21             if (RegexValidator.validComment(comment)) ... }
22     }

```

Figure 14: Motivating example containing ReDOS vulnerabilities

3.3 OBSERVER: Solver-Guided Static Analysis

3.3.1 Rexploiter: Availability Vulnerabilities in Programs Using Regular Expressions. We illustrate our technique using the code snippet shown in Fig. 14, which shows two relevant classes, namely `RegexValidator`, that is used to validate that certain strings match a given regular expression, and `CommentFormValidator`, that checks the validity of a comment form filled out by a user. In particular, the comment form submitted by the user includes the user’s email address, the URL of the product about which the user wishes to submit a comment1, and the text containing the comment itself. We now explain how our technique can determine whether this program contains a denial-of-service vulnerability.

Regular expression analysis. For each regular expression in the program, we construct its corresponding nondeterministic finite automaton (NFA) and statically analyze it to determine whether it’s worst-case complexity is linear, super-linear, or exponential. For our running example, the NFA complexity analysis finds instances of each category. In particular, the regular expression used at line 5 has linear matching complexity, while the one from line 4 has exponential complexity. The regular expressions from lines 2 and 7 have super-linear (but not exponential) complexity. Fig. 15 plots input size against running time for the regular expressions from lines 2 and 4 respectively. For the super-linear and exponential regular expressions, our technique also constructs an attack automaton that recognizes all strings that cause worst-case behavior. For each regular expression, we determine a lower bound on the length of any possible attack string using dynamic analysis.

Program analysis. The presence of a vulnerable regular expression does not necessarily mean that the program itself is vulnerable. For instance, the vulnerable regular expression may not be matched against an attacker-controlled string, or the program may take measures to prevent the user from supplying a string that is an instance of the attack pattern. Hence, we also perform static analysis at the source code level to determine if the program is actually vulnerable.

Going back to our example, the `validate` procedure (lines 11–22) calls `validEmail` to check whether the website administrator’s email address is valid. Even though `validEmail` contains a

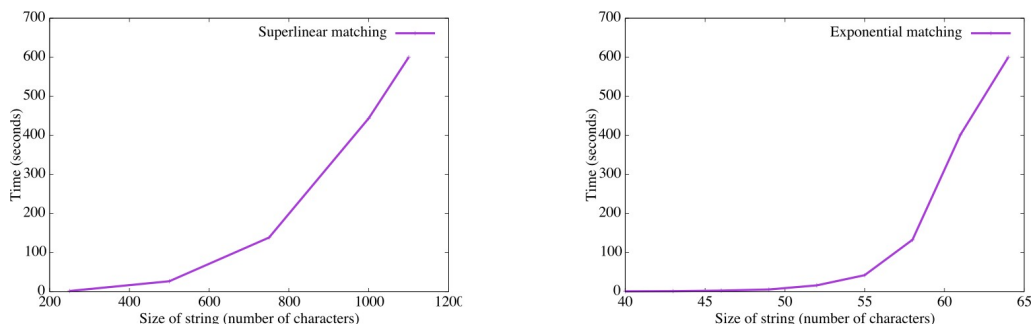


Figure 15: Matching time against malicious string size for vulnerable (left) and hyper-vulnerable (right) regular expressions from Fig. 14

super-linear regular expression, line 15 does not contain a vulnerability because the administrator's email is not supplied by the user. Since our analysis tracks taint information, it does not report line 15 as being vulnerable. Now, consider the second call to `validEmail` at line 17, which matches the vulnerable regular expression against user input. However, since the program bounds the size of the input string to be at most 254 (which is smaller than the lower bound identified by our analysis), line 17 is also not vulnerable.

Next, consider the call to `validUrl` at line 19, where `productUrl` is a user input. At first glance, this appears to be a vulnerability because the matching time of the regular expression from line 4 against a malicious input string grows quite rapidly with input size. However, the check at line 18 actually prevents calling `validUrl` with an attack string: Specifically, our analysis determines that attack strings must be of the form `www.shoppers.com /b /* x`, where `x` denotes any character and `b` is a constant inferred by our analysis (in this case, much greater than 5). Since our program analysis also reasons about input sanitization, it can establish that line 19 is safe.

Finally, consider the call to `validComment` at line 21, where `comment` is again a user input and is matched against a regular expression with exponential complexity. Now, the question is whether the condition at line 20 prevents `comment` from conforming to the attack pattern `\n\t\n\t(\t\n\t)ka`. Since this is not the case, line 21 contains a serious DoS vulnerability.

Summary of challenges. This example illustrates several challenges we must address: First, given a regular expression E , we must reason about the worst-case time complexity of its corresponding NFA. Second, given vulnerable regular expression E , we must determine whether the program allows E to be matched against a string that is (a) controlled by the user, (b) is an instance of the attack pattern for regular expression E , and (c) is large enough to cause the matching algorithm to take significant time.

Our approach solves these challenges by combining complexity analysis of NFAs with sanitization-aware taint analysis. The key idea that makes this combination possible is to produce an attack automaton for each vulnerable NFA. Without such an attack

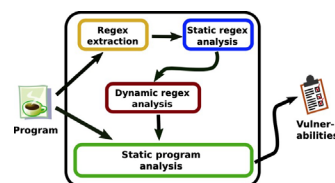


Figure 16: Overview of the REXPLOITER Approach to Availability Vulnerabilities

automaton, the program analyzer cannot effectively determine whether an input string can correspond to an attack string. As shown in Fig. 16, the REXPLOITER tool incorporates both static and dynamic regular expression analysis. The static analysis creates attack patterns $s_0 s^k s_1$ and dynamic analysis infers a lower bound b on the number of occurrences of s in order to exceed a minimum runtime threshold. The program analysis uses both the attack automaton and the lower bound b to reason about input sanitization.

3.3.2 SCANNER: Asymptotic Resource Side-Channel Vulnerabilities. First, we formally define the class of side-channel vulnerabilities studied in the context of SCANNER and justify our decision to focus on this subclass.

Definition 1 (Resource Side-Channel Vulnerabilities.). *Let h, l denote high and low inputs of a program, respectively, and let $R_p(i)$ denote the resource usage of program P on input i . We say that program P has a resource side-channel vulnerability if $\exists h_1, h_2, l \cdot R_p(h_1, l) \neq R_p(h_2, l)$.*

The above definition above is effectively an instantiation of the well-known non-interference policy with respect to resource usage. As in traditional non-interference terminology, high variables represent secret values, while low variables denote values that are not security-sensitive. Hence, according to the above definition, a resource side-channel vulnerability arises if it is possible to observe different resource usages when program P is run on the same low input but different high inputs. Hence, an adversary can glean information about the secret simply by observing the program's resource usage.

Since Definition 1 does not specify the kind of resource, it is quite general and can be instantiated in a variety of ways to yield different classes of side-channel vulnerabilities previously discussed in the literature. For instance, if the resource of interest is CPU cycles, then this vulnerability corresponds to a timing side channel. On the other hand, if we instantiate R_p with power consumption, then the vulnerability could be exploited to cause a power monitoring attack.

However, since non-interference is a very strong condition that is violated by almost any program, we believe that such an approach is not practical. For instance, consider a program that has a very minor resource imbalance (e.g., a few CPU cycles) across two different execution paths. While such a program has a resource side-channel vulnerability according to Definition 1, it is very unlikely to be exploitable because an attacker cannot reliably observe this minor imbalance in resource usage. Motivated by the observation that standard non-interference is too strong of a policy to reason about exploitable resource side channels, we instead focus our attention on asymptotic resource side channels, which can be reliably exploited by attackers

Definition 2 (Asymptotic Resource Side-Channel Vulnerabilities.). *Let h denote the high inputs of a program, and let $R_p(I)$ denote the resource usage of program P on input I . We say that program P has an asymptotic resource side-channel vulnerability if: $\exists h_1, h_2. R_p(h_1) \neq \Theta(R_p(h_2))$.*

In this definition, observe that the low inputs are unconstrained, so $R_p(h_1)$ and $R_p(h_2)$ are both functions of low inputs l . Thus, this definition states that it is possible to find a pair of secrets h_1 and h_2 for which the resource usage of P will be asymptotically different with respect to the low-inputs. Since the attacker can control the program's low inputs, he can easily tell whether the secret is h_1 or h_2 by running the program on arbitrarily large values of the low input.

Unfortunately, in practice it is still difficult to statically reason about worst-case resource usage, and existing techniques for reasoning about complexity typically do not scale to large programs.

Hence, to analyze realistic web applications with a low false positive rate, we further restrict our attention to the following subclass of asymptotic side-channel vulnerabilities that can be detected using lightweight static analysis:

Definition 3 (Constant-Superconstant Resource Side-Channel Vulnerabilities.). *Let h, l denote the high and low inputs of a program, and let $R_P(I)$ represent the resource usage of program P on input I . Program P has a constant-superconstant resource side-channel vulnerability if:*

$$\exists h_1, h_2 \cdot R_P(h_1) = O(1) \wedge R_P(h_2) = O(L).$$

In other words, we are interested in detecting a subclass of asymptotic side-channel vulnerabilities where the resource usage is constant for some values of the secret but a function of L for other values of the secret. Note that, in this definition, $R_P(h_2) = O(1)$ implies that the resource usage is a non-constant function of L when the high inputs are fixed to h_2 . Our detection algorithm targets this specific subclass of vulnerabilities because it is possible to find instances of Definition 3 in a practical way without requiring heavy-weight resource usage analysis.

Static Detection of Asymptotic Resource Side-Channel Vulnerabilities. We now turn our attention to the static analysis algorithm for detecting asymptotic (i.e., constant-superconstant) vulnerabilities. We first explain the key ideas underlying our algorithm and then provide a more formal description of the analysis. The key idea underlying our static analysis is to look for two different paths π_1, π_2 in the program such that π_1, π_2 satisfy the following conditions:

1. Paths π_1, π_2 differ on the value of the secret,
2. Resource usage of π_1 is a function of low input l , and
3. Resource usage of π_2 is not dependent on l .

To see how these conditions relate to Definition 3, observe that condition (1) partitions the high inputs into two disjoint classes H_1, H_2 such that $h_1 \in H_1$ satisfies the predicates on path π_1 , while $h_2 \in H_2$ satisfies the predicates in path π_2 . Furthermore, condition (2) states that $\forall h_2 \in H_2 R_P(h_2) = O(1)$, while condition (3) ensures $\forall h_1 \in H_1 R_P(h_1) = O(L)$.

Our static analysis combines two kinds of taint information (namely, secret taint and input taint) with a coarse summary of resource usage for each program fragment. Input taint information is used to summarize the resource usage of each code snippet as zero, constant, or potentially infinite, where the last value indicates that resource usage is controlled by the attacker. In contrast, secret taint information is used to determine if two program paths differ with respect to some confidential data. Combining these two crucial pieces of information, our analysis reports an error if it encounters a secret-tainted condition such that the resource usage is potentially infinite along one branch, but not in the other.

Design and Implementation of SCANNER. SCANNER analyzes PHP applications and detects two specific kinds of vulnerabilities involving time and response-size. Specifically, timing side channel vulnerabilities allow the attacker to infer confidential data by observing server response times. In contrast, response-size side channels enable the attacker to glean secret information by observing the size of the response. The workflow of the SCANNER tool is shown in Figure 17. Internally, the

SCANNER tool consists of three different modules that perform complementary tasks:

- The detection module performs static analysis to flag potential asymptotic timing and response-size side-channel vulnerabilities.
- The error diagnostic module performs additional static analysis to report descriptive warnings. In particular, this module identifies confidential database fields that may be leaked by the application due to the uncovered vulnerabilities.
- The exploit generation module performs backwards symbolic execution to semi-automatically generate a Javascript program that can be used to exploit the uncovered vulnerability using a cross-site search attack.

SCANNER is itself written in PHP and consists of 5,000 lines of code. Our implementation uses the Z3 SMT solver [33] to solve the constraints collected during its symbolic execution phase for semi-automated exploit generation.

SCANNER’s detection module extends the static analysis in two important ways: First, since PHP programs are not annotated with source and consume statements, we must reason about taint introduction and removal as well as resource consumption at the level of PHP statements. Second, our implementation is interprocedural and must deal with object-oriented features of the PHP language. To use SCANNER, web developers need to annotate database fields that are considered to be confidential. In particular, rather than directly annotating the

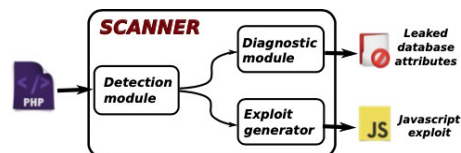


Figure 17: Workflow of the SCANNER tool

source code, SCANNER users need to provide a separate file containing the security-sensitive attributes of each database table. Our implementation considers two kinds of taint sources, namely, user inputs and database operations. User inputs are taint sources with label L and correspond to reads from pre-defined PHP arrays, such as GET, POST, and SESSION. In contrast, taint sources with label H correspond to database queries that retrieve private data from a database. Given a list of confidential database attributes, SCANNER automatically infers whether or not a given SELECT statement is a taint source.

Recall that asymptotic resource side-channel attacks take advantage of the fact that the attacker can arbitrarily inflate resource usage by providing sufficiently large inputs. Hence, if the application sanitizes user input by bounding its size to a small value, then the attacker can no longer inflate resource usage. In our implementation, we consider checks that impose an upper bound on string length to be sanitizers. Similarly, string operations (e.g., substr) that limit the size of the resulting string are also considered to be sanitizers. Resource consumption. In our implementation, consume statements from our formalization are instantiated in different ways depending on the type of side-channel vulnerability. For timing vulnerabilities, we consider every instruction to consume one unit of resource; hence, resource usage only becomes ∞ if a loop bound is tainted by user input. For response-size vulnerabilities, resource consumption corresponds to print statements.

The static analysis described so far only allows SCANNER to detect the existence of a possible side-channel vulnerability. To help the user understand the severity and implications of the uncovered vulnerability, SCANNER performs an additional static analysis. Specifically, SCANNER’s error diagnostic module outputs the set of confidential database attributes that may be leaked by the application. For instance, if SCANNER’s output includes Employees.Age, this means that the

attacker can infer something about the age an employee stored in the Employees database table. To provide such diagnostic information, SCANNER performs a backwards static analysis that utilizes the information produced by the vulnerability detection module. Specifically, the input to the error diagnostic module is the predicate of a conditional branch along which there is a resource usage imbalance. Given such a predicate C , SCANNER then collects all secret-tainted variables used in C and performs backwards symbolic execution to trace each variable v to the database query that caused v to become tainted. Confidential database attributes that are mentioned in the WHERE clause of the query are then reported as being potentially leaked.

3.4 EXPLOITR: Explaining Potential Witnesses and Exploits.

3.4.1 DPDEBUGGER: Debugging Differential Performance. Developers often face the problem of finding and fixing performance problem in their programs. Performance bugs manifest themselves only on certain pathological inputs. For instance, there can be two inputs of the same size on which the performance is unexpectedly different in an otherwise correct program.

We introduced and studied the *differential performance problem*, where the goal is to explain the difference in performance between two classes of inputs in terms of program internals, such as which functions were called and how many times were they called. This information is useful, as it allows a programmer or an analyst to better assess whether the performance difference is inherent to the problem, or is a result of a coding inadequacy. The problem is hard for both traditional static as well as dynamic analysis techniques. Static analysis commonly target logical correctness properties (and not the performance), and are not as scalable as techniques based on machine learning. On the other hand, dynamic analysis focus on individual traces, whereas for the differential performance problem we need to compare the performance on different traces.

We propose a technique called *differential performance debugging*, based on inference of DRT. DRTs are regression trees where the goal is to classify input data. In contrast, the objective of standard regression tree learning is to predict the output for a previously unseen input. The input to the differential performance problem is a set of program traces. Each trace is represented as follows. We have input variables (such as the size of the user input), auxiliary variables (such as the functions called), and the output variable (such as the running time). The output to the differential performance problem is the DRT. The internal nodes of the tree has predicates on auxiliary variables. The leaf nodes model the output variable as a function of input variables. The leaf nodes represent the performance for different classes of inputs capturing asymptotically different performance behaviors.

In accordance with Occam's razor, we are interested in finding a DRT with a small number of clusters, while minimizing the modeling error. Furthermore, the DRT should be a human readable explanation, which also suggests that smaller number of clusters is preferable. There are two major steps in our algorithm. First, we project the data into the input and output variables and cluster the data in this domain. Second, we consider the auxiliary variables only and identify what separates the clusters in terms of these variables. We use an off-the-shelf decision tree learning algorithm for the second step. The first, clustering, step thus reduces the regression tree inference problem to the decision tree inference problem.

For our approach, we need a clustering algorithm that produces *functional* clusters, that is, clusters that represent functions from input variables to the output variable. We adapt two classical clustering algorithms. First, we extend the K -means algorithm to produce linear functional clusters.

This is done by generalizing the notion of the cluster centroid from a point to a linear function. Second, we adapt the spectral clustering algorithm by defining a new notion of similarity between two data points that we dub *alignment* kernel. Here, two data points are more similar when the line defined by them captures more data points.

We show how our prototype tool DPDEBUGGER can be used for diagnosing performance problems on a real-life example. We also use the example to explain how the tool works and compare it to existing approaches.

Performance problem with Apache FOP. Apache FOP (Formatting Objects Processor) is a Java application that reads a formatting object such as an XML file and renders the resulting pages to a specified output format such as PDF and PS. The formatting document can specify that an external image in, for example, a PNG or JPEG format should be included. A user had a suspicion that there is a performance bug in handling PNG images. They reported in a forum post in 2011 that they have two PNG images, which have the same size, but one of them takes seven times as much to render as the other one¹.

Performance debugging with DPDEBUGGER. Our tool, DPDEBUGGER, can be used exactly in this situation, to help an analyst to explain the differences in performance. The analyst can then decide whether the differences are inherent to the problem or they are a manifestation of a coding error. Song and Lu [36] reported that in 60% of bugs analyzed by them, users notice huge performance differences among similar inputs.

The analyst has to collect a number of inputs which in this case are PNG and JPEG images of various sizes. We remark that in many cases, the inputs can be collected from log files of a system or generated by existing software fuzzers [37].

Given the collection of inputs, the tool produces the two diagrams in Figure 18. The analyst can diagnose the performance problem using these two figures. The left diagram is a plot of the image size (input variable) and the running time (output variable). From the plot, the analyst can see that there are two performance clusters. However, the analyst does not know what separates these two clusters. It is instructive to emphasize that looking at the two groups of inputs does not explain the difference. All the JPEG images are in the lower (red) cluster, but there are PNG images of similar size in both clusters.

We thus turn to the DRT on the right side of Figure 18 for the explanation. It says that if for an input, the function `encodeRender...RGB` is not called, then the input will be in the red cluster. The user can analyze the reverse call graph to see how the function is called, and realize that it is called for PNG files, but not for JPEG files. Further, the node to the right of the root has the function `getICCprofile`. This function is what distinguishes the red (fast) cluster from the blue (slow) one among PNG files. It is called once for every PNG file, but it is called more than once only for PNG files with a color scheme that needs to be deflated. After code analysis, we see that one source of the performance problem is that some PNG files have a compressed color scheme that needs to be deflated. Another way how a PNG image can be in the blue cluster is that the dimension of the input image overflows the allowed size (see the lowest internal node).

The user thus learns from the DRT that what separates the two clusters is the fact that the images in the blue clusters either need to be deflated first, or overflow. So in this case, the diagnosis is that the difference in performance is not a coding mistake but it is inherent to the problem. The user can conclude this without needing to sift through almost 40,000 methods that Apache FOP has.

¹https://bz.apache.org/bugzilla/show_bug.cgi?id=51465

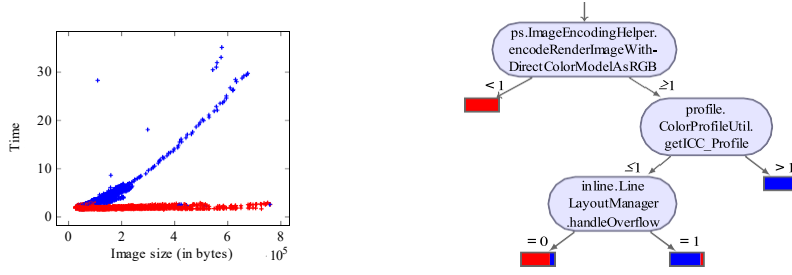


Figure 18: Performance clusters (left) in the FOP data set and a DRT (right) explaining the clusters in terms of function calls

Inside the tool DPDEBUGGER. We now describe how DPDEBUGGER obtains the diagrams in Figure 18. The diagram on the left side is obtained as follows. The program is run on all the inputs, and the graph of input size versus running time is plotted. Then we need to cluster the data. As a preliminary step, we need to get the values of the auxiliary variables. In this example, they indicate how many times a function was called. We have a variable for each (non-standard library) function. Then, we find functional clusters in the data from the left diagram. That is we consider input and output variables only (not auxiliary variables) and identify a given number K of clusters. These clusters are intended to capture the classes of inputs with different (asymptotic) performance. Spectral clustering identifies the clusters marked red and blue in Figure 18.

To obtain the right diagram, the next step is decision tree learning. We want to learn what separates the clusters in terms of predicates on the auxiliary variables. Note that here each data point is labeled with one of the K labels (indicating to which cluster the data point belongs), therefore efficient decision tree learning (such as CART) can now be used to construct the regression tree. The final discriminant regression tree is on the right part of Figure 18.

We applied the state-of-the-art algorithms for learning regression trees (M5Prime [38] and GUIDE [39]) to this problem. Our goal is different from the goal of these algorithms: we aim to classify data, whereas both Guide and M5Prime aim to predict the output for previously unseen input. We believe that this accounts for the following differences. M5Prime finds a linear regression tree with 23 different linear models in leafs. Guide finds 4 linear models – two of them same as our algorithm, but two of the in-between, perhaps to account for noise. Note that DPDEBUGGER identified 2 clusters. Furthermore, the time of the algorithms are as follows: M5Prime 97 sec., Guide 1233.6 sec., and DPDEBUGGER 14.4 sec.

3.4.2 FUCHSIA: Data-Driven Debugging for Functional Side Channels. We first illustrate what an attacker can infer based on functional observations, even in the presence of noise. Second, we show that it is critical to use functional observations to evaluate the resulting threats of side channels. We consider the *known-message* threats where the attacker knows public values, but she cannot control them. We focus on the situation where secret values remained unchanged for some amount of time (e.g., passwords, social security number, and friends of a user in social media). The attacker who has access to the source code tries to infer the secret by observing the execution time and knowing the public values.

Let us consider the classical definition of confidentiality: – noninterference. A program is *unsafe* iff for a pair of secret values s_1 and s_2 , there exists a public value p such that the behavior of

the program on (s_1, p) is observably different than on (s_2, p) . If our observable is the execution time T , then the program is unsafe iff:

$$\exists s_1, s_2, p : T(s_1, p) \neq T(s_2, p). \quad (1)$$

In our setting, for each secret value, we observe the execution time of a program on a number of public values. Thus, the program is unsafe iff:

$$\exists s_1, s_2 : (\lambda p. T(s_1, p)) \neq (\lambda p. T(s_2, p)). \quad (2)$$

In other words, the program is unsafe if the two secret values do not correspond to the same function of public inputs.

Side channels in the presence of noise. Quantitative observations of a program's runtime behavior are often noisy. For instance, running a program with the same inputs twice on the same machine result in different measurements of execution time. Observing the program remotely adds a further level of noise. Classical definitions of confidentiality properties, therefore, need to be adapted to noisy environments. In the noisy environment, no two observations are equal and our definition needs to include ϵ tolerance:

$$\exists s_1, s_2 : d(\lambda p. T(s_1, p), \lambda p. T(s_2, p)) > \epsilon. \quad (3)$$

In this definition, d is a distance between two functions. The distance is suitably chosen, typically based on the noise expected for a particular use case. A straightforward extension of classical noninterference with ϵ tolerance to our functional setting is to use the ∞ -norm for the distance function:

$$d_\infty(f_{s_1}, f_{s_2}) = \sup_p |f_{s_1}(p) - f_{s_2}(p)|, \quad (4)$$

where $f_s(p) = \lambda p. T(s, p)$. However, we now demonstrate that the point-wise distance d_∞ is not the only option, and that depending on the type of noise, different distances are needed. In particular, we show that if we use the point-wise distance, we could certify a side-channel vulnerable program.

Gaussian noise (pointwise independent, mean 0). Consider the two functional observations (red and black) of a program in Figure 19. The red function corresponds to the secret value s_1 and the black function corresponds to the secret value s_2 . The eavesdropper can produce this graph easily by trying possible inputs on their machine beforehand. At runtime, the eavesdropper collects the public inputs and the execution time, and tries to learn the secret by matching the observed data to the red or black functions. In this example, we assume that the noise for each pair of public-secret inputs is independent and identically distributed. Furthermore, we assume that it is distributed according to a Gaussian distribution with mean 0. Let us consider ϵ of 3ms, and then apply our definition with distance d_∞ . We see that the two functional observations are ϵ -close for this distance, so the attacker cannot infer the secret value (s_1 or s_2). However, the functional observations are clearly very different, and an eavesdropper can reliably learn the secret. This can be captured using the L_1 -norm as the distance function. This example shows the point-wise distance d_∞ may not be appropriate to detect certain side channels.

Gaussian noise (pointwise independent, mean C). Let us consider the case where the noise is Gaussian, but with a non-zero mean. The non-zero mean is fixed, but is unknown to the attacker. This case arises if, for instance, the eavesdropper is remote and cannot determine the delay

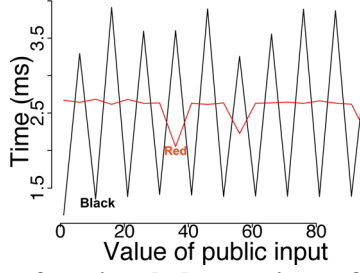


Figure 19: Two functional observations of execution time

introduced by the network and separate it from the noise of the remote machine. Consider a program with two functional behaviors (red and black) pictured in Figure 20, where the red and black behaviors correspond to secret s_1 and secret s_2 , respectively.

At runtime, the attacker interacts with the remote server running the same instance of the application with a fixed secret value. The green timing function in Figure 20 shows the execution-time function of public inputs obtained from observing the remote server. The green function looks far apart from both local observations (black and red functions). However, due to the effects of remote observations, the attacker knows that the execution time is off by an unknown constant. The attacker is in effect observing only the shapes of functions, i.e., their first derivatives. So, the distance is over the derivatives of functions. The attacker can use this distance and calculate that the green function is closer to the black function than the red function. Note that if the L_1 distance between the first derivative of two timing functions is greater than ϵ , the corresponding secret values can leak to a remote attacker.

The number of distinct timing observations (or clusters) over secret inputs is an important measure to evaluate the resulting threats of side channels. Here, we illustrate that it is critical to analyze functions to obtain clusters, especially in dynamic analysis. We consider three side-channel vulnerable programs. Program P_1 , a variant of square and multiply algorithm, leaks the number of set bits in the secret. Program P_2 , a vulnerable Jetty password matcher, leaks the length of secret passwords. Program P_3 , a vulnerable google Keyczar password matcher, leaks the value of secret passwords. Let public values be the sequence “0”, “1”, “00”, “01”, ..., “1111”, and secret values be the set “0”, “1”, “00”, “01”, ..., “1111”.

Figure 21 shows six plots about the execution times of three programs: Figure 21 (a-c) are point-wise depictions and Figure 21 (d-f) are functional presentations. The x-axis is the index of 30 public values (see corresponding values in the table in Figure 21). The y-axis is the cost of executing a pair of secret and public values in the number of bytecode executed.

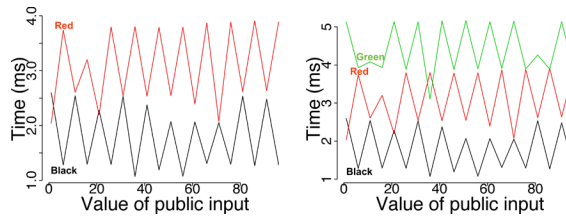


Figure 20: Attacker observations: (a) local (b) remote

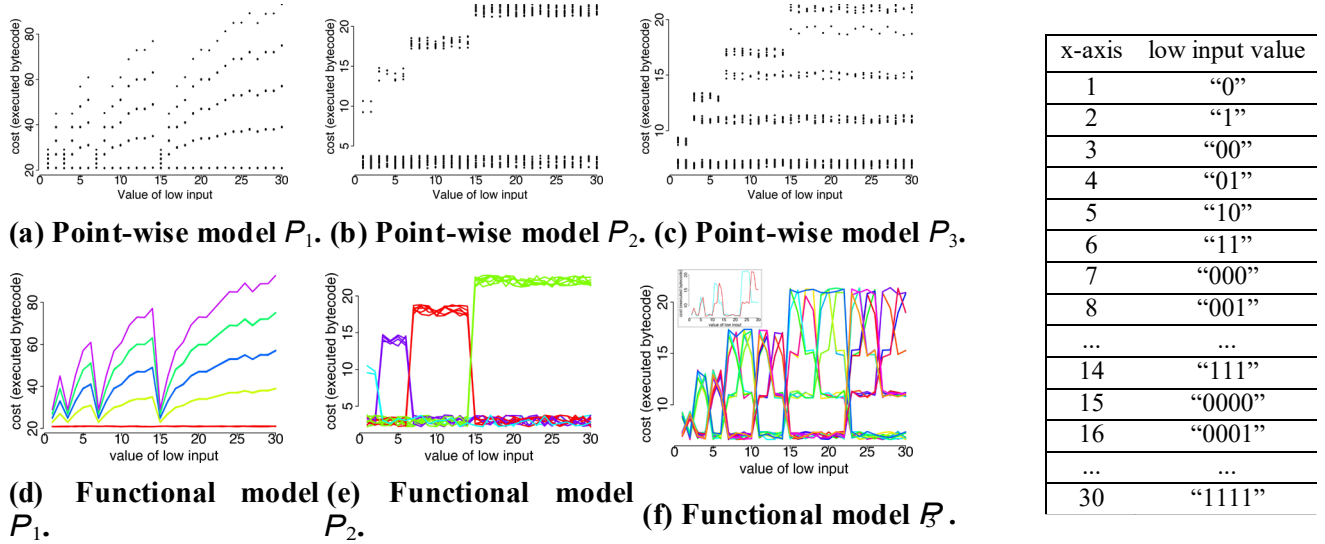


Figure 21: Programs P_1 , P_2 , P_3 are leaking the number of set bits, the length of secret, and the secret values, respectively. In programs P_2 and P_3 , the clustering over functions are required to find the correct number of classes of observations.

According to the point-wise definition, the number of clusters can be obtained by fixing the public input and finding the number of distinguishable classes of observations (different cost values) over all possible secrets. Let's consider a public input value that gives the largest number of clusters for each example. Let's pick the index 30 on the x-axis for all examples in Figure 21 (a-c). This choice results in 5, 2, and 5 clusters for programs P_1 , P_2 , and P_3 , respectively. According to the functional definition, we model the execution times of each secret value as a function from the public input to the cost of execution. Figure 21 (d-f) show 30 functions in each plot, colored based on their cluster labels. Any two functions that are $\epsilon = 0.1$ close to each other belong to the same cluster.

For P_1 , there are 5 clusters in Figure 21 (a), and also 5 functional clusters in Figure 21 (d). This means that both the point-wise and functional definitions agree that there are five clusters (0 to 4 possible set bits). However, the results are different for programs P_2 and P_3 . Figure 21 (b) shows that in each point on the public value, there are two classes of observations (either the lengths of secret and public inputs matched or not). On the other hand, the functional model in Figure 21 (e) shows that there are 4 different functional clusters that correspond to four possible lengths in the secret values. Specifically, let's look at Figure 21 (e) from the indices 7 to 14. These are the low (public) values with the length of three (from "000" to "111"). Secret values with length three are checked against these public values, while all other secret values are rejected immediately. As a result, the timing functions of secret values with the length three goes up from 2 to 17 at index 7 and goes down from 17 to 2 at index of 15. Observe that the clustering helps to group timing functions of secret values with length three in the same class. Similarly, there are unique indices for secrets with the lengths 1, 2, and 4 where the timing functions jump up/down and reveal the length of secrets.

Similarly, the point-wise model for the program P_3 in Figure 21 (c) underestimates that the (whole) secret values are leaking (found 5 clusters). The functional model in Figure 21 (f) finds 30 clusters: there is a unique function for each of the 30 secret values (two samples are shown on top of Figure 21(f)).

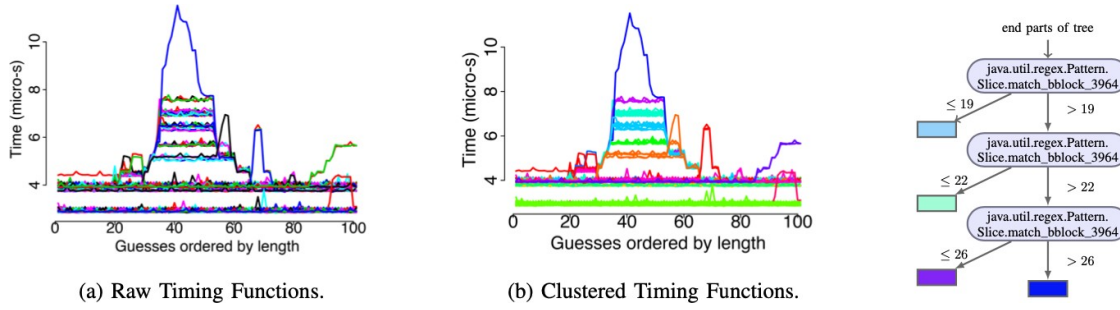


Figure 22: (a) Regex timing functions. Execution times of 435 secret values are modeled as 435 timing functions. How are these functions related? (b) 435 timing functions are clustered into 12 distinguishable classes of observations (clusters) using L_1 -norm distances. The presence of different clusters indicate some properties of secret patterns are leaking. What properties of secret patterns are leaking? (c) Regex decision trees. The number of calls to the basic block at line 3964 of `Pattern.match()` method discriminates different clusters. The code region shows the value of secret patterns is leaking

Debugging with FUCHSIA. We illustrate how our tool can be used for discovering and explaining a zero-day timing side channel. We analyze the `java.util.regex`² package of Open JDK 8. The package includes 620 methods and over 8,000 lines of code. The secret input is the regular expression compiled as a pattern, and the public input is the guess that matches against the pattern. The attacker’s goal is to infer a (fixed) secret (or its properties) by observing the response time for multiple public inputs. The key problem is to help the *defender* discover the existence of such side channels. The defender starts choosing a finite set of secret and public values. In this example, the defender uses AUDITR to generate the set of secret and public input values. The fuzzing of AUDITR is an extension of AFL and Kelinci where it generates multiple public input values for each secret value to discover the functional dependencies of response time over public inputs. The defender obtains 1,154 different secret patterns and 6,365 different matching guesses during 4 hours of fuzzing. The lengths of guesses are at most 52 characters. For each secret value, AUDITR varies 6,365 different guesses and measures the execution time of regex. Then, it fits B-spline to model the timing functions for each secret. Figure 22 (a) shows 435 different timing functions over the public inputs from the guesses with the prefix “bb...” to “gg...”. We choose the subsets of secret and public values for simpler presentations. Next, the defender wants to know how these timing functions are related and if there are timing side channels.

The defender provides the notion of a distance and the tolerance ϵ . In this case, he considers L_1 -norm distance between functions and the tolerance $\epsilon = 0.2$. Given the distance norm and the tolerance, AUDITR uses a non-parametric functional clustering algorithm to discover classes of observations. The clustering algorithm finds 162 clusters over the 1,154 secret values. The existence of 162 distinct classes of observations indicates the presence of a functional side channel in the regex package. Figure 22 (b) shows 12 clusters for the subset of 435 functions presented in Figure 22 (a).

Side channel explanation. Now, the defender wants to know what properties of secret are leaking

²<https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>

```

1  boolean match(Matcher matcher, int i, CharSequence seq)
2  {
3      int[] buf = buffer;
4      int len = buf.length;
5      for (int j=0; j<len; j++) { (line.3964)
6          if ((i+j) >= matcher.to) {
7              matcher.hitEnd = true;
8              return false;
9          }
10         if (buf[j] != seq.charAt(i+j))
11             return false;
12     }
13     return next.match(matcher, i+len, seq);
14 }
15

```

Figure 23: Pattern matching using regex (buf secret, seq public)

through the timing side channels. The task is to learn the discriminant [24]. It helps the defender localize the code regions causing to observe different clusters and use the information to establish facts about the leaks. Specifically, it shows which features are common for secrets in the same cluster and which features separate different clusters. AUDITR uses program internal features such as methods called or basic blocks invoked. These are obtained by running the same set of secret and public inputs on the instrumented regex. The instrumentation provides 203 features about the internals of regex. AUDITR applies an extension of the decision tree learning algorithm from [20]. It produces a decision tree whose nodes are labeled by program internal features and whose leaves represent sets of secret values inside a cluster.

Figure 22 (c) shows (parts of) the decision tree model learned for regex. Using this model, the defender realizes that the executions of the basic block at line 3964 of `Pattern` class in `Slice.match()` method is what distinguishes the clusters. This basic block represents the loop body of the `for` statement in the method shown in Figure 23. For instance, the purple cluster, the functions with the second highest execution times around the index 40 of Figure 22 (b), corresponds to the case where `match_block_3964` calls more than 22 times but less than 26 times. Note that the edge values in the decision tree are also B-spline functions, but we map them to their max values for the illustration. Inspecting the relevant code, the defender realizes that the matching prefix of secret patterns is leaking through the side channels. Hence, an attacker can obtain the (whole) secret patterns, one part in each step of observations. We reported this vulnerability, and the OpenJDK security team has confirmed it. Since fixing this vulnerability requires substantial modifications of library, the developers suggested to add artificial extra delays when the stored pattern is secret in order to mitigate it.

4.0 RESULTS AND DISCUSSION

We will next discuss our results for various components of the AUDITR tool chain. Again, for convenience, we have arranged this section along the four major components (LOOPR, PATHR, OBSERV, EXPLOITR) of our tool chain.

```

> python chj_analyze.py blogger
...
Methodswithbytecode:299
Methods translated 299
...
=====
Completed numerical analysis in 128.994906187 secs
=====
> python chj_analyze_taint.py blogger
...
=====
Completed taint analysis in 7.75347995758 secs
=====

```

Figure 24: Commands to analyze the E2 Blogger application

loops report

```

> python chj_report_loops.py blogger
....
#loops max-depth bounds taints method name (id)
-----
4 3 ?,?,MAX,? +,+,+ NanoHTTPD$HTTPSession.decodeMultipartFormData (43)
3 3 ?,?,? +,+,+ NanoHTTPD$HTTPSession.getBoundaryPositions (47)
5 3 ?,?,?,?,? +,+,+,+ SimpleWebServer.main (170)
3 2 ?,?,? +,+,+ URIVerifier.verify (309)
...

```

Figure 25: Loops report (partial) for the E2 Blogger application

4.1 LOOPR: CodeHawk based conservative and highly scalable static analysis

We will discuss selected challenge problems from Engagement 2.

Blogger. Blogger is a java web server which serves Plain Old Java Objects (POJO), and POJOs that collectively implement a simple public blogging application. Users can browse to the homepage at <http://localhost:8080>. On the homepage users can sign in, create a new blog post and view posts by other users.

Question 40 Is there an algorithmic complexity vulnerability in time that would cause the servers' response time (on the reference platform) to an HTTP request to exceed a runtime of 300 seconds between sending the last byte of the attack and the server sending the last byte of the response, when the maximum sum of the PDU sizes of the TCP requests sent from the user to the server is less than 5000 bytes?

CodeHawk Analysis Blogger has 299 methods. Numerical and taint analysis together take about 135 seconds, see Figure 24 for the commands to run the analysis.

The loops report, shown in Figure 25, presents four high-complexity loops, whose bounds are potentially tainted. The first three entries in the list did not lead to the discovery of any vulnerability, but inspection of the control flow graph of the fourth entry, the function `fi.iki.elonen.URIVerifier.verify()`, revealed two `push()` operations to a `LinkedList` inside a 2-deep nested loop, illustrated in Figure 26. The figure shows the loop nesting level in different shades of red; purple nodes indicate basic blocks that contain push operations.

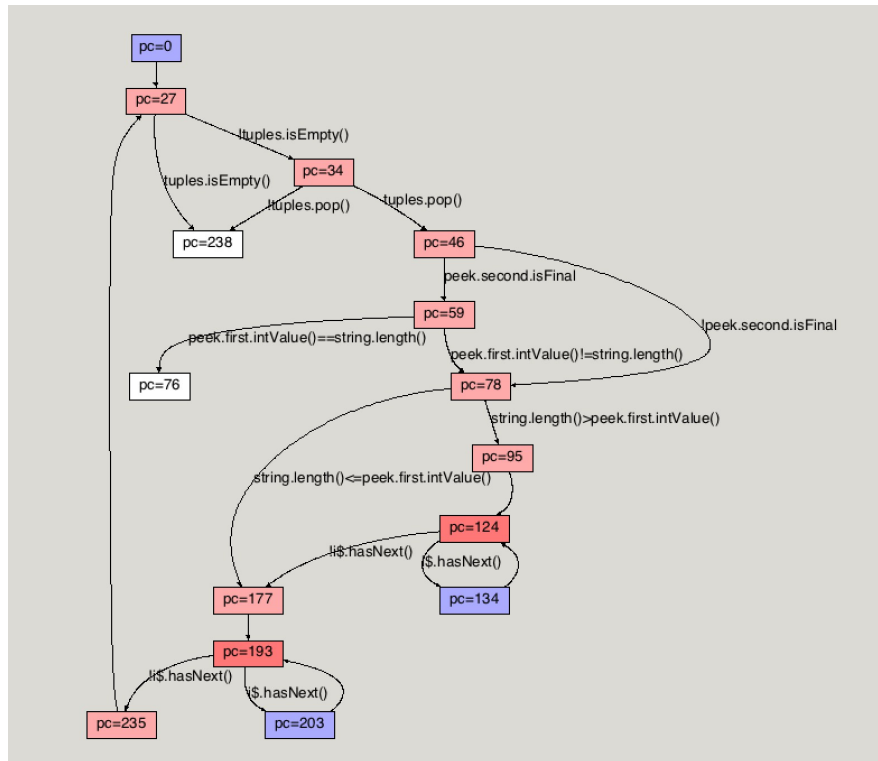


Figure 26: CodeHawk Annotated Control Flow Graph for UriVerifier.verify

Inspection of the CodeHawk reverse call graph started from this function, as shown in Figure 27, establishes that this method is reachable from the ClientHandler thread's run method.

Gabfeed. GabFeed is web forum software. It allows community members to post messages, search posted messages, and provides additional information on selected ('special') topics. Users can sign in to the login page: <https://localhost:8080/login> and log in with valid credentials. Once a user is signed in the user can proceed to visit GabFeed Rooms (forums), start a private chat with another user, view posts by other users on various topics (threads), create new messages on existing topics, create a new thread, or search for a single term in all posted public messages (note this is a case sensitive search). Each day GabFeed offers 'special' search terms to users. When users search for these terms additional information is provided to the user in the search results. Optionally users can authenticate the GabFeed server by browsing to <https://localhost:8080>.

Gabfeed_1: Question 7 Is there an algorithmic complexity vulnerability in time that would cause the challenge program's real runtime to be more than 300 seconds, measured as the time interval between sending the last byte of the attack and the server sending the last byte of the response, if the sum of the PDU sizes of the HTTP requests from the user to the server is not more than 400000 bytes.

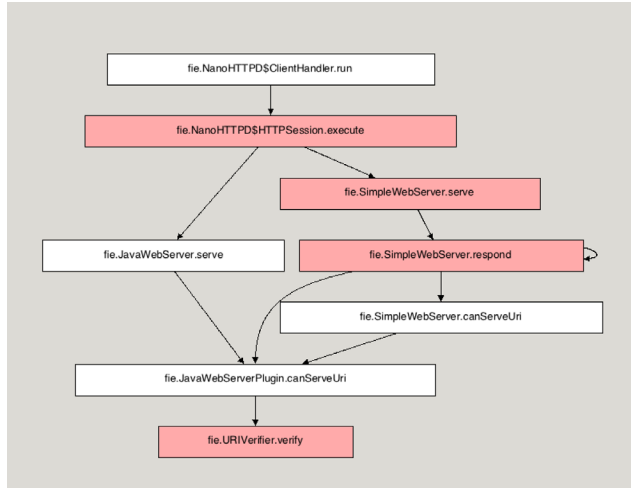


Figure 27: CodeHawk Reverse Call Graph starting from UriVerifier.verify()

```

> python chj_analyze.py gabfeed_1
...
Methods with bytecode : 573
Methods translated    573
...
=====
Completed numerical analysis in 313.647972822 secs
=====
> python chj_analyze_taint.py gabfeed_1
...
=====
Completed taint analysis in 19.294230938 secs
=====

```

Figure 28: Commands to analyze the E2 Gabfeed_1 application

CodeHawk Analysis Gabfeed_1 has 573 methods; numerical and taint analysis take about 330 seconds. In the loops report, shown in Figure 29, the `breakLines` method stands out with 11 loops with maximum depth 4. The maximum loop depth of 4, however, is somewhat misleading in that artificial loops have been added to create a distribution of running times by increasing the loop count with conditional iterations based on a random variable.

All loop bounds in the `breakLines` method relate to the number of words in the line presented. When setting these bounds in the userdata we obtain the costmodel for the `breakLines` method shown in Figure 30, which can be expressed more concisely as

$$\text{LB: } 1956 + a + (142 + b)n + 2653n^2 + 2708n^3 + 98n^4$$

$$\text{UB: } 1956 + a + (2731 + b)n + 5242n^2 + 2708n^3 + 98n^4$$

where n denotes the number of words in the line, and a and b are unknown costs of other methods. To account for the artificially introduced randomization loops we can replace these loop bounds with a fixed value, $1/p$, where p is the probability that the loop returns without advancing the count of words, which results in the costmodel shown in Figure 35, more concisely represented as:

$$\text{LB: } 5580 + a^t + (9142 + b^t)n + 3055n^2$$

```
gabfeed_1: loops report
```

```
> python chj_report_loops.py gabfeed_1
...
#loops max-depth      bounds      taints      method name (id)
-----
11      4      ?,?,?,?,?,?,?,?,?,? +,+,+,+,+,+,+,+,+ LineBreak.breakLines (542)
3      3      MAX,?,?      +,+,+      ModPow.modPow (546)
3      3      ?,MAX,?      +,+,+      DES.getLongFromBytes (40)
3      3      ?,?,?      +,+,+      DES.createSubkeys (43)
3      3      ?,MAX,?      +,+,+      DES.encryptBlock (44)
3      3      ?,?,?      +,+,+      DES.encrypt (46)
3      3      ?,?,?      +,+,+      DES.parseBytes (50)
3      3      ?,?,?      +,+,+      DES.hex (51)
3      3      ?,?,?      +,+,+      DES.passwordToKeyHelper (56)
3      3      ?,MAX,?      +,+,+      TemplateEngine.replaceTagsBuilderHelper (445)
5      3      MAX,MAX,?,?,MAX +,+,+,+,+ Sorter.mergeHelper (575)
6      3      ?,MAX,?,MAX,?,? +,+,+,+,+ ChatSerializer.deserialize (186)
3      3      ?,?,?      +,+,+      TreeNode.comparableClassFor (518)
3      3      MAX,?,?      +,+,+      OptimizedMultiplier.standardMultiply (552)
...
```

Figure 29: Loops report (partial) for the E2 Gabfeed_1 application

$$\text{UB: } 10758 + a^t + (12594 + b^t)n + 4138n^2$$

which shows a quadratic dependency on the number of words with a high coefficient that better reflects the actual complexity.

Gabfeed_1: Question 10 Is there a side channel in time in the challenge program from which a third party can discover the server's private key used in the server's authentication?

CodeHawk Analysis The loops report shown before (Figure 29) lists `modPow` as another high-complexity loop. The costmodel for this method, with adjustments for the randomization loop (see Figure 32), shown in Figure 35, indicates a large difference between the lower and upper bound for the cost. In particular, the term

$$2w^2 \text{ cost}(\text{fastMultiply})$$

stands out, which is a potentially cpu-intensive method. with high-complexity and recursive methods descendants.

The ascendant call graph in Figure 34 shows that the `modPow` methods is reachable from `Http` handler method that accepts the user input, and the taint transmission graph shows that the user input from that method, represented by the first method argument is propagated to the first argument of the (static) method `modPow`. The commands to produce the taint graph are shown in Figure 33.

Gabfeed_3: Question 36 Is there an algorithmic complexity vulnerability in time that would cause the challenge program's real runtime to exceed 300 seconds measured as the time interval between sending the last byte of the attack and the server sending the last byte of the response, with an input of less than 400,000 bytes in HTTP requests.

gabfeed_1: cost model results

```
> python chj_analyze_cost.pygabfeed_1
> ....
> python chj_show_costmodel.py gabfeed_1
...
ccs.linebreak.LineBreak.breakLines:(String)List (542)
lb:((((((((1956
+ (3 * (symc:count * symc:cost_java.util.Random.<init>:()))))
+ (2 * symc:cost_java.util.ArrayList.add:(Object)Z))
+ (2 * symc:cost_org.apache.commons.lang3.StringUtils.join:(Object,String)String))
+ symc:cost_org.apache.commons.lang3.StringUtils.split:(String)String)
+ symc:cost_java.util.Collections.reverse:(List))
+ (2 * symc:cost_org.apache.commons.lang3.ArrayUtils.subarray:(Object,I,I)Object))
+ (98 * (((symc:count * symc:count) * symc:count) * symc:count)))
+ (2708 * (((symc:count * symc:count) * symc:count)))
+ (2653 * (symc:count * symc:count)))
+ (142 * symc:count))

ub:((((((((1956
+ (3 * (symc:count * symc:cost_java.util.Random.<init>:()))))
+ (2 * symc:cost_java.util.ArrayList.add:(Object)Z))
+ (2 * symc:cost_org.apache.commons.lang3.StringUtils.join:(Object,String)String))
+ symc:cost_org.apache.commons.lang3.StringUtils.split:(String)String)
+ symc:cost_java.util.Collections.reverse:(List))
+ (2 * symc:cost_org.apache.commons.lang3.ArrayUtils.subarray:(Object,I,I)Object))
+ (153 * (((symc:count * symc:count) * symc:count) * symc:count)))
+ (2708 * (((symc:count * symc:count) * symc:count)))
+ (5242 * (symc:count * symc:count)))
+ (2731 * symc:count))
...
```

Figure 30: Cost model report (partial) for the E2 Gabfeed_1 application

CodeHawk Analysis CodeHawk provides a report that lists all conditional branch expressions. Inspection of this report revealed the unusual condition

```
(list.size() % 8) != conditionObj0.getValue() in ccss.Sorter.changingSort
```

Inspection of the control flow graph of `Sorter.changingSort`, depicted in Figure 37, shows that an extra basic block (starting at `pc = 179`) is executed if `list.size() % 8` is equal to zero. Within this basic block a call is made to `Sorter.changingSortHelper` which pushes an extra block on the stack as shown in the call graph in Figure 38. The redundant stack block significantly increases the runtime.

Graph Analyzer. The analyst graph visualization and modification tool is a mock-up of an analyst's notebook (TM). This is similar to a tool often used by law enforcement officers and intelligence analysts to chart their investigations and organize their thoughts. Several analysts will often be working on the same investigation, and these investigation files will be passed around the team via e-mail or file sharing. Often time's external, public data is also used for these investigations. To make integration with other tools easier, the application parses and exports a modified version of the GraphViz DOT graph specification.

Graph_analyzer: Question 20 Is there an algorithmic complexity vulnerability in time that would cause the runtime to exceed 1 hour with an input file of less than 5000 bytes?

Inspection of the reverse call graph on all of these methods shows that only the methods `MinimizerBarnesHut.minimizeEnergy`, `otgb.CmdLoadDot.doIt`, and `CmdForceLayout.runCmd` are reachable from the `CommandProcessor.main` method (see Figure 41). `MinimizerClassic.minimizeEnergy` is not reachable by any method; most others are only reachable from

gabfeed_1: cost model results

```
> python chj_analyze_cost.pygabfeed_1
> ....
> python chj_show_costmodel.py gabfeed_1
...

ccs.linebreak.LineBreak.breakLines:(String)List (542)
lb:(((((((5580
+ (2 * (symc:count * symc:cost_java.util.Random.<init>: ()))
+ (2 * symc:cost_java.util.Random.<init>: ()))
+ (2 * symc:cost_java.util.ArrayList.add: (Object)Z))
+ (2 * symc:cost_org.apache.commons.lang3.StringUtils.join: (Object,String)String))
+ symc:cost_org.apache.commons.lang3.StringUtils.split: (String)String)
+ symc:cost_java.util.Collections.reverse: (List))
+ (2 * symc:cost_org.apache.commons.lang3.ArrayUtils.subarray: (Object,I,I)Object))
+ (3055 * (symc:count * symc:count)))
+ (9142 * symc:count))

ub:(((((((10758
+ (2 * (symc:count * symc:cost_java.util.Random.<init>: ()))
+ (2 * symc:cost_java.util.Random.<init>: ()))
+ (2 * symc:cost_java.util.ArrayList.add: (Object)Z))
+ (2 * symc:cost_org.apache.commons.lang3.StringUtils.join: (Object,String)String))
+ symc:cost_org.apache.commons.lang3.StringUtils.split: (String)String)
+ symc:cost_java.util.Collections.reverse: (List))
+ (2 * symc:cost_org.apache.commons.lang3.ArrayUtils.subarray: (Object,I,I)Object))
+ (4138 * (symc:count * symc:count)))
+ (12594 * symc:count))
...
```

Figure 31: Cost model report (partial) for the E2 Gabfeed_1 application, accounting for randomization loops

editor commands. There does not seem to be any command to activate the editor from the CmdProcessor provided. We were not able to establish a taint graph between CommandProcessor.main method and minimizeEnergy.

SnapBuddy. SnapBuddy is a web application for image sharing. Users can login to SnapBuddy to upload photos to share with their friends. Photos can be marked public in which case any user of the system can view them (all profile photos are public). Photos that aren't marked public should only be viewable by the owner and their friends. In addition to uploading photos, users can apply filters to their photos to change how they look. Finally, users are able to have the server tell them their current location (similar to how observed WiFi access points can be used to determine a user's location). Once their location is set, they can see the people around them in that location (location changes can only occur a certain number of times per day), and their neighbors can see them.

snappbuddy_3: Q 22 Is there an algorithmic complexity vulnerability in time that causes the challenge program's runtime to exceed 300 sec. measured as the time interval between sending the last byte of the attack and the server sending the last byte of the response.

CodeHawk Analysis. Commands to analyse Snapbuddy_3 are shown in Figure 42.

TextCrunchr. TextCrunchr is a text analysis program. It can perform some useful analysis (word frequency, word length, etc...) and includes some fun stuff too (automatic enigma encryption of input). TextCrunchr can process plain text files and can handle compressed files as

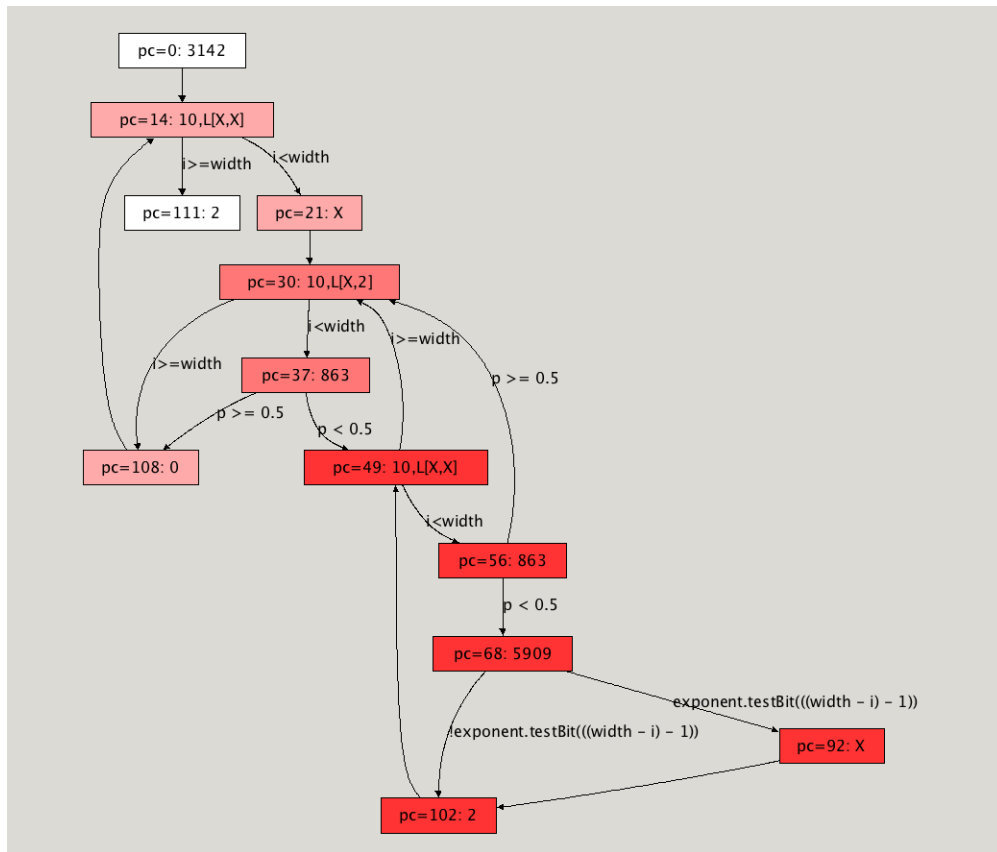


Figure 32: gabfeed_1: Control flow graph for ModPow.modPow

```

>python chj_report_taint_origins.py gabfeed_1 --source Http
index taint origin site
-----
68 STUB:AbstractHandler.getUrlParam @8: URLEncodedUtils.parse
96 VAR:com.cyberpointllc.stac.webserver.handler.AbstractHandler.handle:(com.sun.net.httpserver.HttpExchange):r0
97 VAR:com.cyberpointllc.stac.webserver.handler.AbstractHandler.handle:(com.sun.net.httpserver.HttpExchange):r1
-----
....
> python chj_analyze_taint_propagation gabfeed_1 97
....
Completed taint propagation analysis in 18.1891429424 secs
=====
....
> python chj_report_taint_trail.py gabfeed_1 97 --sink modPow
Generating paths from VAR(AbstractHandler.handle:r1@0) to
- VAR(ModPow.modPow: r0 @0)

```

Figure 33: Commands to produce a taint trail for the `modPow` method in `E2:gabfeed_1`

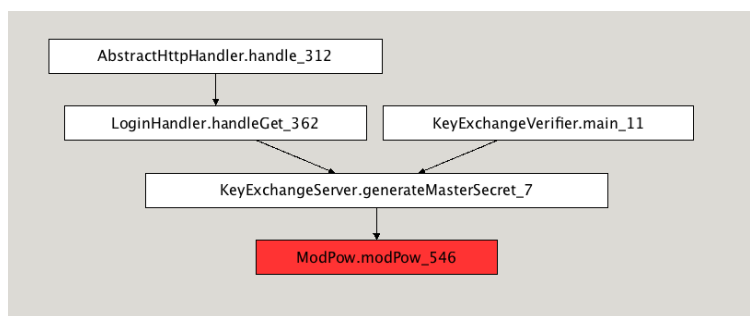


Figure 34: `gabfeed_1`: Ascendant call graph for `ModPow.modPow`

well (which it will uncompress to analyze the contents). Researchers at Zooomblr (a startup search company) are using it in the back-end to analyze a large corpus of data from the Internet. These researchers have set up a web crawler to download pages and compressed files from various websites. The files from those websites are automatically fed into TextCrunchr which emits the results of the analysis (which can be console, xml or graphical).

textcrunchr_6: Question 8 Is there an algorithmic complexity vulnerability in time that would cause the challenge program's real runtime to exceed 300 seconds for an input of at most 400,000 bytes?

CodeHawk Analysis. The control flow and the reverse control flow graphs for `textcrunchr_6` are shown in Figure 44 and Figure 45.

gabfeed_1: cost model results

```
...
ccs.math.ModPow.modPow:(BigInteger, BigInteger, BigInteger) BigInteger (546)
lb: (((3154
+ (symc:width * symc:cost_java.util.Random.<init>: ()))
+ (13568 * (symc:width * symc:width)))
+ (1812 * symc:width))

ub: (((3154
+ (symc:width * symc:cost_java.util.Random.<init>: ()))
+ (2 * ((symc:width * symc:width) * symc:cost_fastMultiply)))
+ (20724 * (symc:width * symc:width)))
+ (4401 * symc:width))
...
```

Figure 35: Cost model report (partial) for the E2 Gabfeed_1 application, accounting for randomization loops

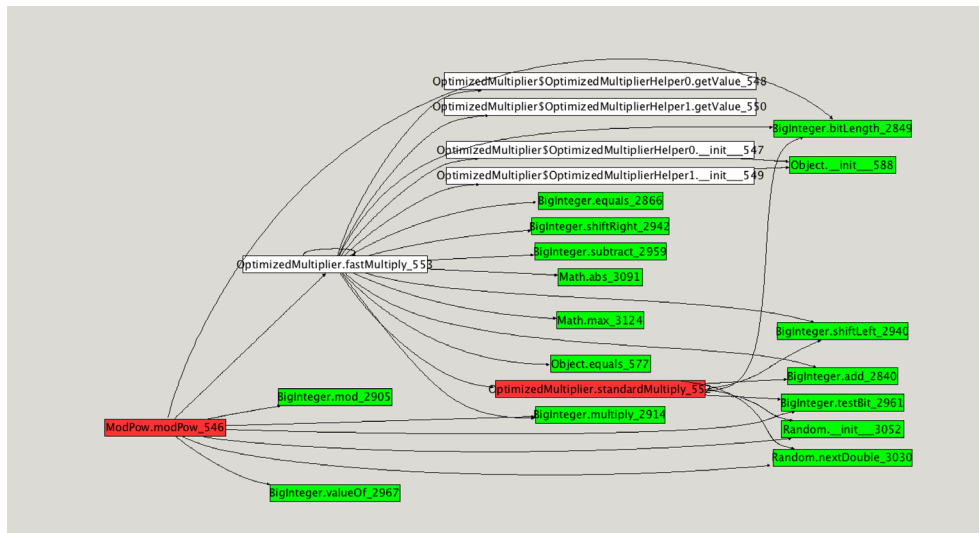


Figure 36: gabfeed_1: Descendant call graph for ModPow.modPow

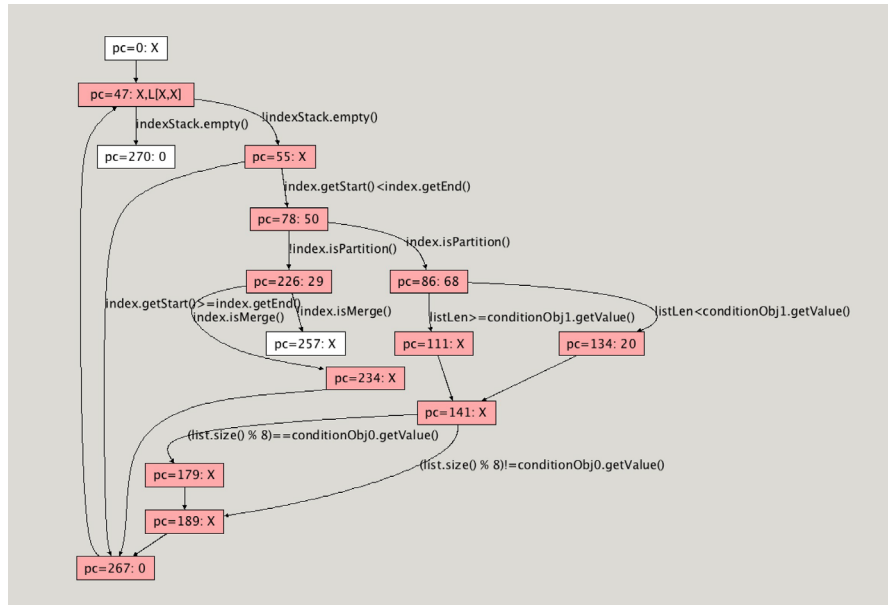


Figure 37: gabfeed_3: Control flow graph for `Sorter.changingSort`

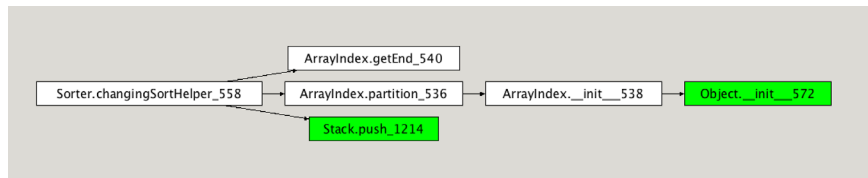


Figure 38: gabfeed_3: Descendant call graph for `Sorter.changingSortHelper`

```
> python chj_analyze.py graph_analyzer
....
Methods with bytecode : 4380
Methods translated   4380
....
Completed numerical analysis in 422.310921907 secs
=====
....
> python chj_analyze_taint.py graph_analyzer
....
Completed taint analysis in 359.280421972 secs
=====
```

Figure 39: Commands to analyze E2:graph_analyzer

graph_analyzer:loop bounds

```
> python chj_report_loops.py graph_analyzer
....
```

#loops	max-depth	bounds	taints	method name (id)
10	4	1,?,102,?,	+...+	MinimizerBarnesHut.minimizeEnergy (498)
12	4	?,1,?,1,1,?,?	+...+	MinimizerClassic.minimizeEnergy (513)
3	3	?,?,?	+,+,+	LevelLayout.addDummies (433)
10	3	1,1,1,?,1,1,?,MAX	+...+	OptimizerModularity.refine (533)
5	3	?,?,?,?,1	+,+,+,+,+	CmdDisplayGUI.loadlayers (883)
3	3	1,?,?	+,+,+	CmdLoadDot.doIt (899)
3	3	MAX,?,1	+,+,+	ModeModify.legal (1763)
3	3	7,7,7	+,+,+	ColorTilePanel.netscapeColors (4141)
3	3	1,?,?	+,+,+	CmdForceLayout.runCmd (4606)
....				

Figure 40: Loops report for the E2:Graph analyzer application

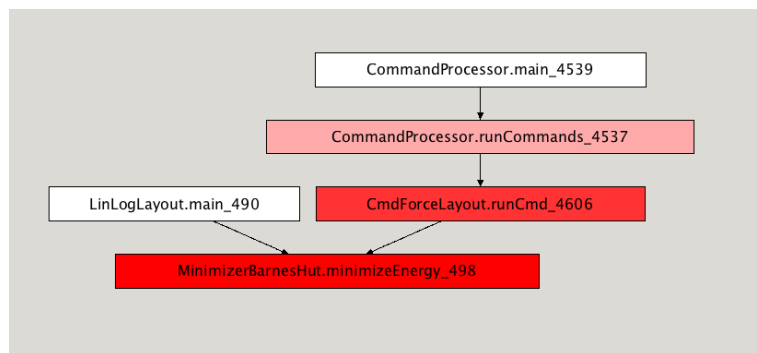


Figure 41: graph_analyzer: Ascendant call graph for MinimizerBarnesHut.minimizeEnergy

```
> python chj_analyze.py snapbuddy_3
....
Methods with bytecode : 5043
Methods translated : 5043
....
Completed numerical analysis in 3050.25512099 secs
=====
....
> python chj_analyze_taint.py snapbuddy_3
....
Completed taint analysis in 131.139262915 secs
=====
....
```

Figure 42: Commands to analyze E2:snapbuddy_3

```
graph_analyzer:loop bounds
```

```
> python chj_report_loops.py snapbuddy_3
.....

6      6      ?,?,?,?,? +,+,+,+,+ DilateFilter$ClassfilterPixels.doIt0 (2062)
5      5      ?,?,?,?,?,? +,+,+,+,+,+ OilFilter.filterPixels (3432)
5      5      ?,?,?,?,? +,+,+,+,+ EdgeFilter.filterPixels (2157)
7      5      ?,?,?,?,?,? +,+,+,+,+,+ BlockFilter.filter (1274)
6      5      ?,?,?,?,?,? +,+,+,+,+,+ ColorHalfToneFilter.filter (1823)
7      5      ?,257,?,?,?,MAX+,+,+,+,+ DiffusionFilter.filterPixels (2052)
5      5      ?,?,?,?,? +,+,+,+,+ ErodeFilter.filterPixels (2215)
5      5      ?,?,?,?,? +,+,+,+,+ SCNoise.evaluate (4935)
15     5      ?,?,257,?,?,?,?,?,?,130,?,?,?,?,?,?+,+,+,+,+,+,+,+,+,+,+LensBlurFilter.filter (3052)
5      5      ?,?,?,?,? +,+,+,+,+ MaximumFilter.filterPixels (3199)
6      5      ?,?,?,?,? +,+,+,+,+,+ MedianFilter.filterPixels (3216)
7      5      MAX,?,?,?,?,?+,+,+,+,+,+ QuantizeFilter.quantize (3641)
6      5      ?,?,?,?,?,? +,+,+,+,+,+ ReduceNoiseFilter.filterPixels (3727)
5      5      ?,?,?,?,? +,+,+,+,+ SkeletonFilter.filterPixels (4061)
4      4      ?,?,?,?,? +,+,+,+ MinimumFilter.filterPixels (3219)
4      4      ?,?,?,?,? +,+,+,+ GaussianFilter.convolveAndTranspose (2491)
4      4      ?,?,?,?,? +,+,+,+ ConvolveFilter.convolveHV (1235)
4      4      ?,?,?,?,? +,+,+,+ ConvolveFilter.convolveV (1237)
4      4      ?,?,?,?,? +,+,+,+ CausticsFilter$ClassfilterPixels.doIt7 (1403)
4      4      ?,?,?,?,? +,+,+,+ SCNoise.evaluate (4934)
14     4      ?,?,?,?,?,?,?,?,?,?,?+,+,+,+,+,+,+,+,+,+,+GlintFilter$Classfilter.doIt10 (2509)
5      4      ?,?,?,?,?,? +,+,+,+,+ FFT.butterflies (4809)
4      4      ?,?,?,?,? +,+,+,+ LifeFilter.filterPixels (3091)
4      4      ?,?,?,?,? +,+,+,+ OctTreeQuantizer$ClassreduceTree.doIt1 (3362)
4      4      ?,?,?,?,? +,+,+,+ OutlineFilter.filterPixels (3455)
4      4      ?,?,?,?,? +,+,+,+ SmartBlurFilter$ClassthresholdBlur.doIt2 (4196)
11     4      ?,?,?,?,?,?,?,?,?+,+,+,+,+,+,+,+,+,+,+SmearFilter$ClassfilterPixels.doIt4 (4248)
3      3      ?,?,?,? +,+,+ DES.passwordToKey (61)
4      3      ?,?,?,?,? +,+,+,+ LocationServiceImpl.<init> (305)
3      3      ?,?,?,? +,+,+,+ CurlFilter.filter (1949)
6      3      ?,?,?,?,?,? +,+,+,+,+,+ DespeckleFilter.filterPixels (2008)
3      3      MAX,MAX,? +,+,+ ApplyMaskFilter$ClasscomposeThroughMask.doIt1 (1167)
3      3      ?,?,?,? +,+,+,+ ConvolveFilter.convolveH (1236)
5      3      ?,?,MAX,?,MAX +,+,+,+ BoxBlurFilter.blur (1324)
3      3      ?,?,MAX,MAX +,+,+ CausticsFilter$ClassfilterPixels.doIt3 (1399)
10     3      ?,?,?,?,?,?,?,?,?+,+,+,+,+,+,+,+,+,+,+Noise.init (4911)
3      3      ?,?,?,? +,+,+ Noise.findRange (4913)
3      3      ?,?,?,? +,+,+,+ Gradient.sortKnots (2601)
4      3      MAX,MAX,MAX,? +,+,+,+ PointFilter.filter (1541)
3      3      ?,MAX,? +,+,+,+ ChromaKeyFilter.filter (1597)
3      3      ?,?,?,? +,+,+,+ TransformFilter.filter (1801)
7      3      ?,?,?,MAX,?,?,258+,+,+,+,+,+ ContourFilter.filterPixels (1865)
5      3      ?,?,?,?,9,? +,+,+,+,+ ImageUtils.paintCheckedBackground (2914)
6      3      ?,?,?,?,?,MAX +,+,+,+,+ DeinterlaceFilter.filter (2001)
5      3      ?,MAX,MAX,MAX,MAX +,+,+,+,+ DisplaceFilter.filter (2078)
3      3      ?,?,?,? +,+,+,+ EmbossFilter$ClassfilterPixels.doIt4 (2165)
8      3      ?,?,MAX,4,MAX,?,257,MAX+,+,+,+,+,+Histogram.<init> (2785)
6      3      257,MAX,?,4,MAX,256 +,+,+,+,+ EqualizeFilter.filterPixels (2185)
3      3      MAX,MAX,? +,+,+ GlowFilter.filter (2536)
3      3      MAX,?,?,? +,+,+,+ GradientFilter$ClassrepeatGradient.doIt0 (2628)
3      3      ?,?,?,? +,+,+,+ HalfToneFilter$Classfilter.doIt0 (2730)
5      3      ?,MAX,MAX +,+,+,+ ImageCombiningFilter.filter (2806)
5      3      ?,?,?,?,? +,+,+,+,+ LaplaceFilter.filter (3024)
6      3      ?,?,?,MAX,?,? +,+,+,+,+,+ FFT.transform2D (4811)
3      3      ?,?,?,? +,+,+,+ MotionBlurFilter.filter (3277)
5      3      MAX,?,21,?,?,? +,+,+,+,+ QuiltFilter.filterPixels (3676)
3      3      MAX,MAX,? +,+,+,+ ShapeFilter.makeMap (3971)
4      3      MAX,?,?,MAX +,+,+,+ ShatterFilter.filter (4019)
3      3      MAX,MAX,? +,+,+,+ SmearFilter$ClassfilterPixels.doIt3 (4247)
3      3      ?,?,?,? +,+,+,+ UnsharpFilter.filter (4590)
5      3      ?,?,?,?,? +,+,+,+,+ VariableBlurFilter$Classblur.doIt3 (4607)
3      3      ?,MAX,MAX +,+,+,+ WarpGrid.addCol (4674)
14     3      ?,?,?,?,?,?,?,MAX,MAX,?,MAX,?,?,?+,+,+,+,+,+,+,+,+,+,+WarpGrid.warp (4678)
3      3      MAX,MAX,? +,+,+,+ WarpGrid$ClassremoveRow.doIt2 (4671)
.....
```

Figure 43: Loops report for the E2:SnapBuddy_3 application

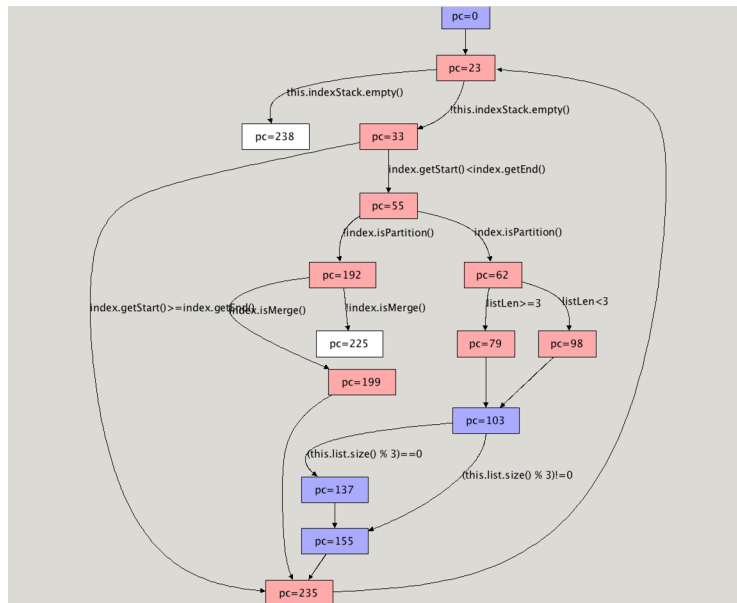


Figure 44: textcrunchr_6: Control flow graph for `Sorter$ClasschangingSort.doIt1`

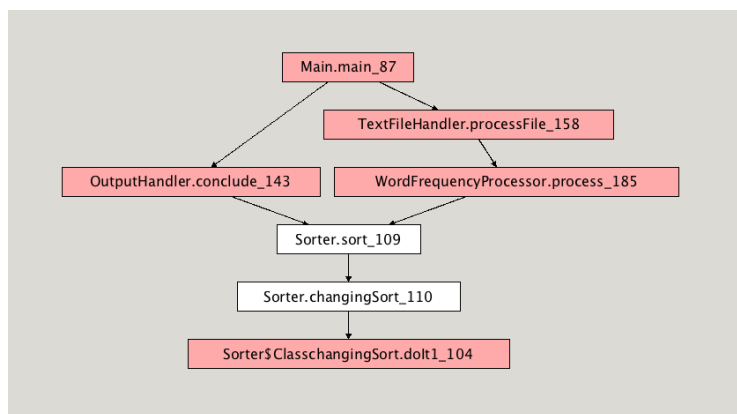


Figure 45: textcrunchr_6: Reverse callgraph for `Sorter$ClasschangingSort.doIt1`

4.2 PATHR: A precise path- and context-sensitive static analysis

4.2.1 QUANTM: Type-directed Bounding of Memory. We implemented Quantm in Scala using the Checker Framework [40, 41], Microsoft Z3 [33] and Scala SMT-LIB [42]. Quantm is implemented as a Java annotation processor, relying on the Checker Framework to extract type annotations and perform type checking. Microsoft Z3 served as an off-the-shelf SSMTMT solver. We also used Scala SMT-LIB for parsing string-typed annotations. We chose several web applications as benchmarks (180k lines of code in total), each of which supports various functionalities. Benchmarks were collected from different sources, including GitHub (jforum3 with 218 stars and SpringPetClinic with 2325 stars), Google Code Archive (jRecruiter³), and DARPA STAC project (TextCrunchr, Braidit, WithMi, Calculator, Battleboats, Image_processor, Smartmail, Powerbroker, and Snapbuddy). Braidit is a peer-to-peer 2-player game that tests the players’ ability to recognize topologically equivalent braids, based on the word equivalence problem in the Artin braid group. Textcrunchr is a text analysis program, performing text analysis (word frequency, word length, etc...) and including other utilities (automatic enigma encryption of input). SprintPetClinic is a very popular sample Spring-based application. jforum3 is a powerful and robust discussion board system implemented in Java. jRecruiter is a Spring-based show-case application for a variety of interesting Java related technologies. It is actively used by the Atlanta Java Users Group (AJUG) for their job posting service.

To set up the experiments, we created drivers invoking callback methods in patterns that imitate standard usage. To support the Object-Oriented feature (which is orthogonal to the problem and approach in this thrust), we not only annotate collection-typed local variables, but also annotate collection-typed object fields that are reachable from local variables. Then we gave bounds to each method as tight as possible and used Microsoft Z3 to verify the bounds.

We evaluated our technique by answering the following research questions

research question (RQ)1. **Bound verification.** How effective is *AST Counter Instrumentation*?

That is, what percentage of methods and collection variables were verified w.r.t. their specifications.

RQ2. **Analysis speed.** How fast/scalable is our verification technique?

Our findings are summarized below.

RQ1: Bound verification. We verified 16453 methods in total, 239 of which are non-vacuously verified (who declares at least one collection variable) and the rest are vacuously verified (who declares no collection variable). If not considering vacuously verified methods, then we verified 239 out of 290 (82.4%) methods. In order to verify method boundedness, we also wrote and verified global invariants on 381 collection variables out of a total of 426 (89.4%), as well as provided 82 method summaries. We invoked 78 callbacks from drivers. We believe this result demonstrates that our technique is effective at verifying method and variable specifications. Our technique works very well when there is no statement reading a list-typed element from a collection, which if it happens, constitutes the vast majority of the causes of the 51 unverified methods and 45 unverified collections, because to ensure soundness we had to enforce no constraint on these list-typed variables read from collections, leading to unboundedness. We currently do not support this feature in the type system and we will leave it for future work. Note that in the table we did

³<https://code.google.com/archive/p/jrecruiter/>

Table 3: Benchmark Results for QUANTM

Benchmarks	Lines of Code	Verified/ Unverified methods	Verified/ Unverified collections	Call- backs	Sum- maries	Analysis time (s)
TextCrunchr	2,150	(13+190)/5	23/5	4	9	14.7
Braidit	20,835	(8+2114)/0	50/0	8	8	84.2
jforum3	22,813	(35+1675)/8	54/10	24	27	69.8
jRecruiter	13,936	(29+933)/5	40/4	10	7	45.1
SpringPetClinic	1,429	(6+98)/0	11/1	9	12	15.8
WithMi	24,927	(30+2515)/5	35/2	5	4	82.0
Calculator	5,378	(20+316)/2	25/6	5	3	18.2
Battleboats	21,525	(8+2171)/6	12/2	5	2	75.6
Image_processor	1,365	(4+110)/0	5/0	0	0	7.8
Smartmail	1,977	(7+137)/4	11/3	0	0	10.9
Powerbroker	29,374	(22+3015)/8	27/3	3	8	91.6
Snapbuddy	34,797	(57+2940)/8	88/9	5	2	107.0
Total	180,506	(239+16214)/51	381/45	78	82	622.6

not include unverified methods and variables caused by orthogonal problems such as Java features (e.g. dynamic dispatch).

We attribute the effectiveness of *AST Counter Instrumentation* to the scalable type checking approach and our AST counter-base approach. Also note that without *AST Counter Instrumentation*, it would have not been possible to **flow-insensitively** verify the desired properties. The detailed results from each benchmark are in Table 3. Below is the legend for the table.

- “Lines of code” counts the total lines of code in projects.
- “Verified methods” gives number of verified methods and unverified methods.
- Number of verified methods is split into *non-vacuously* and *vacuously* verified, where being vacuously verified means not declaring any local collection variables.
- “Verified/Unverified collections” gives the number of collection variables that are verified versus unverified.
- “Callbacks” gives the number of invoked callbacks in drivers.
- “Method summaries” gives the number of method summaries supporting verifying collection variables that are inter-procedurally mutated.
- “Analysis time” indicates the speed of our analysis on each benchmark. Experiments were conducted on a 4-core 2.9 GHz Intel Core i7 MacBook with 16GB of RAM running OS X 10.13.6.

We present an interesting loop that we found and simplified from the `jforum3` benchmark. In the while loop at line 4-17, line 5 reads in a String with `readLine` and line 14 adds an element to list `comments`. Although the while loop may not terminate, the inductive invariant $\text{len}(\text{comments}) - \text{idx}(\text{reader}) = c5 - c14$ is preserved before and every execution step. Here we consider variable `reader` as an iterator, respecting the semantics of the `readLine` API.

```

1 @Inv("len(comments)-idx(reader)=c5-c14") List<String> comments = new ArrayList<>();
2 // ...
3 String s;
4 while (true) {
5   c5: s = reader.readLine();
6   if (s != null) {
7     s = s.trim();
8   }
9   if (s == null || s.length() < 1) {
10    continue;
11  }
12  if (s.charAt(0) == '/') // comment if (collectComments and s.length() > 1) c14:
    comments.add(s.substring(1)); continue;

```

We also discovered a Denial-of-Service bug from benchmark `TextCrunchr` that is caused by a collection variable with an excessively large length. `TextCrunchr` is a text analysis program that is able to perform some useful analysis (word frequency, word length, etc.), as well as process plain text files and compressed files, which it will uncompress to analyze the contents. The vulnerability is in the decompressing functionality where it uses a collection variable `queue` to store files to be decompressed. Our tool `Quantm` correctly did not verify the boundedness of variable `queue` and we believe this leads to `TextCrunchr`'s being vulnerable to a Zip bomb attack, because variable `queue` may store an exponential number of files that is caused by a carefully crafted zip file, which contains other carefully crafted zip files inside, thus leading to an exponential number of files to be stored in variable `queue` and to be decompressed.

We also discovered a Denial-of-Service bug from benchmark `calculator`. `Calculator` is a server composed of a suite of six different calculators that can be used to evaluate various arithmetic expressions, supporting addition, subtraction, multiplication and etc. It also supports bracket expressions such as $(5 + 6) \cdot 2$. We discovered method `processOutcome` is vulnerable, which uses a collection variable `operandStack` to store intermediate computation results when processing bracket expressions. At the beginning we failed to verify the boundedness of variable `operandStack`. However later we realized that the usage of variable `operandStack` may lead to an out-of-memory error: to trigger excessive memory usage, we could feed as input a sequence of bracket expressions and then cause calculator to waste lots of expensive computation (e.g. multiplication of two very large numbers), whose temporary results all reside in memory at the same time because of being reference by variable `operandStack`, thus leading to out-of-memory error. Specifically, calculator will evaluate all bracket expressions but only return the evaluation of the right-most bracket expression as the result. For example, for input expression $(1 + 2)(3 \cdot 4)(2^5)$, calculator will first evaluate expression $(1 + 2)$ but then will immediately discard the result before evaluating expression $(3 \cdot 4)$. Similarly, it will discard the value of $(3 \cdot 4)$ before evaluating expression 2^5 . The final result will be 32, the evaluation of 2^5 . `snappybuddy` is a web application for image sharing. It has a Denial-of-Service vulnerability in method `changingSortHelper`, performing merge sort by using a collection variable `indexStack`, which stores merge or partition computation that should happen in future. The vulnerability is that method `changingSortHelper` pushes redundant merge computation into stack variable `indexStack`, wasting computation.

RQ2: Analysis speed. On average, it takes 51.9 second to analyze a 15k lines of code benchmark program (including vacuously verified methods) with DPDEBUGGER. The detailed results from each benchmark are in Table 3. Given the lines of code of our benchmarks, we believe this result indicates that the speed of our analysis benefits from being type-based and flow-insensitive, exhibiting the potential of scaling to even larger programs.

In experiments, we encountered collection variables that could not be annotated, leaving Quantm unable to verify boundedness of methods that declare them. We next categorize the reasons and discuss future work for improvements:

- To ensure soundness, we enforce no constraint on a collection-typed variable’s length (i.e. allow it to be infinitely long), when it is the result of reading a list-typed element from a collection. The reason is that the type system does not yet support annotating lengths of inner lists. This extension of our type system is left for future work.
- Not discovering the right global invariants. In the future we plan to automate the invariant discovery process with abstract interpretation, which will hopefully help uncover more invariants.
- Imprecision and “soundness” caused by Java features such as aliasing, dynamic dispatch, inner class, class inheritance and multi-threading. We regard these as orthogonal problems to our problem statement and we could extend our type system to support them.

Integration with building tools. To evaluate how user-friendly Quantm is for a developer, we also evaluated how Quantm integrates with open source repositories (i.e. jforum3, jRecruiter and SpringPetClinic) that use popular building tools (e.g. Maven). We discovered that the configuration is reasonably easy: Developers only need to add several Maven dependencies (including Quantm, Checker framework, Scala library, Scala SMT-LIB and Microsoft Z3’s Java bindings) into `pom.xml` and specify Quantm as an additional annotation processor. After that, a developer could immediately start using our tool!

Annotation workflow. The typical annotation workflow of a user is to first configure Quantm as an annotation processor, and then compile the target code/project without any annotations. Note that errors and warnings are expected if Quantm cannot prove boundedness of a procedure, which is intrinsically caused by insufficient annotations (i.e. type refinements). In the end, a user will fix the errors and warnings by annotating collection variables. In the case of a method returning a locally allocated collection variable, we inlined the method into its caller to ensure soundness. Additionally, to perform inter-procedural analysis, we introduce method summaries to describe changes in lengths of collection-typed variables caused by method invocation. Method summaries are automatically applied when type checking a method invocation statement. The annotation burden for method summaries was light (6.8 methods on average) in the experiments.

4.2.2 Themis: Precise Detection of Side-Channel Vulnerabilities. We describe our evaluation of Themis on a set of security-critical Java applications. Our evaluation is designed to answer the following research questions:

RQ1 How does Themis compare with state-of-the-art tools for side channel detection in terms of accuracy and scalability?

RQ2 Is Themis able to detect known vulnerabilities in real-world Java applications, and can Themis verify their repaired versions?

RQ3 Is Themis useful for detecting zero-day vulnerabilities from the real world?

In what follows, we describe a series of experiments that are designed to answer the above questions. All experiments are conducted on an Intel Xeon(R) computer with an *E5 1620 v3* CPU and 64G of memory running on Ubuntu 16.04.

Detection of Known Vulnerabilities. To demonstrate that Themis can be used to detect non-trivial vulnerabilities in real-world Java programs, we further evaluate Themis on security-sensitive Java frameworks. The benchmarks we collect come from the following sources:

1. Response-size side-channel benchmarks from existing 4 publications;
2. Benchmarks that contains a response-size side channel from the STAC engagements;
3. A well-known timing side channel in the MessageDigest class from JDK6; and
4. Seven other benchmarks with known vulnerabilities collected from Github.

Benchmarks that fall in the first two categories contain response-size side-channel vulnerabilities, and all other benchmarks contain timing side-channels. All benchmarks except for those in category (1) also come with a repaired version that does not exhibit the original vulnerability. Before running Themis, we need to specify the entry points of each application. Since most applications come with test cases, we use these test harnesses as entry points. For those applications for which we do not have suitable drivers, we manually construct a harness and specify it as the entry point.

The table in Figure 46 shows the accuracy and running time of Themis on these benchmarks. Using a value of $\epsilon = 64$, Themis successfully finds vulnerabilities in the original vulnerable versions of these frameworks and is able to verify that the original vulnerability is no longer present in the repaired versions. The running time of Themis is also quite reasonable, taking an average 8.81 seconds to analyze each benchmark.

Benefit of taint analysis. Themis performs taint analysis to identify hot spots, which overapproximate program fragments that may contain a side-channel vulnerability. The QCHL verifier only analyzes such hot spots rather than the entire program. To demonstrate the usefulness of taint analysis, we compare the lines of code (in Soot IR) in the original application (reported in the LOC column) with the lines of code (also in Soot IR) with those analyzed by the QCHL verifier (reported in the LOC' column). As we can see from Figure 46, taint analysis significantly prunes security-irrelevant parts of the application in terms of lines of codes. This pruning effect can also be observed using other statistics. For example, the number of reachable methods ranges from 15 to 1487, with an average of 479, before taint analysis, whereas the number of reachable methods after taint analysis ranges from 6 to 35, with an average of 15, after taint analysis. Thus, pruning using taint information makes the job of the QCHL verifier significantly easier.

Benefit of ϵ . To justify the need for our relaxed notion of non-interference, Figure 46 also shows the results of the same experiment using an ϵ value of 0. Hence, the $\epsilon = 0$ column from Figure 46

Benchmark	Version	LOC	LOC'	$\epsilon = 64$	$\epsilon = 0$	Time (s)
Spring-Security	Safe	1630	41	✓	✓	1.70
Spring-Security	Unsafe	1602	32	✓	✓	1.09
JDK7-MsgDigest	Safe	633	30	✓	✓	1.27
JDK6-MsgDigest	Unsafe	619	27	✓	✓	1.33
Picketbox	Safe	208	73	✓	✗	1.79
Picketbox	Unsafe	180	65	✓	✓	1.55
Tomcat	Safe	12221	100	✓	✗	9.93
Tomcat	Unsafe	12173	96	✓	✓	8.64
Jetty	Safe	2667	77	✓	✓	2.50
Jetty	Unsafe	2619	76	✓	✓	2.07
orientdb	Safe	19564	134	✓	✗	37.99
orientdb	Unsafe	19413	131	✓	✓	38.09
pac4j	Safe	1978	104	✓	✗	3.97
pac4j	Unsafe	1900	105	✓	✓	1.85
boot-auth	Safe	7106	74	✓	✗	9.12
boot-auth	Unsafe	6977	69	✓	✓	8.31
tourPlanner	Safe	7735	46	✓	✓	22.22
tourPlanner	Unsafe	7660	34	✓	✓	22.01
Dyna_table	Unsafe	175	40	✓	✓	1.165
Advanced_table	Unsafe	232	55	✓	✓	2.01

Figure 46: Evaluation on existing vulnerabilities: a checkmark indicates that Themis gives the correct result, while cross indicates a false positive

Benchmark	LOC	Category	#Reports	Time (s)
Jetty	2619	Server	4	10.17
Tomcat	12173	Server	1	5.86
OpenMRS	10721	Healthcare	1	9.71
OACC	78	Authentication	1	1.83
Apache Shiro	4043	Authentication	0	6.54
Apache Crypto	4505	Crypto	0	4.33
bc-java	5759	Crypto	0	6.89

Figure 47: Evaluating Themis on identifying zero-day vulnerabilities from popular Java applications

corresponds to the standard notion of non-interference. As we can see from the table, Themis reports several false positives using an ϵ value of 0. In particular, the repaired versions of some programs still exhibit a minor resource usage imbalance but this difference is practically infeasible to exploit, so the developers consider these versions to be side-channel-free. However, these programs are deemed unsafe using standard non-interference. We believe this comparison shows that our relaxed policy of ϵ -bounded non-interference is useful in practice and allows security analysts to understand the severity of the side channel.

Benefit of relational analysis. To investigate the benefit of relational invariants, we analyze the safe versions of the 20 benchmarks from Figures 46 and 47 with relational invariant generation disabled. In this case, Themis can only verify the safety of 10 of the benchmarks. Although this number can potentially be increased by using a more sophisticated non-relational loop invariant generation algorithm, Themis circumvents this need, instead using simple relational invariants that are conjunctions of simple equality constraints. This experiment corroborates the hypothesis that QCHL makes verification easier by requiring simpler loop invariants compared to other techniques like self-composition.

Discovery of Zero-Day Vulnerabilities. To evaluate whether Themis can discover unknown vulnerabilities in real world Java applications, we conducted an experiment on seven popular Java frameworks. Our data set covers a wide range of Java applications from different domains such as HTTP servers, health care platforms, authentication frameworks, etc. For example, Eclipse Jetty is a well-known web server that is embedded in products such as Apache Spark, Google App Engine, and Twitter’s Streaming API. OpenMRS is the world’s leading open source enterprise electronic medical record system platform; OACC is a well-known Java application security framework, and bc-java is an implementation of the Bounty Castle crypto API in Java.

As in our previous experiment, we first manually annotate each application to indicate the sources of confidential information. We then use Themis to find timing side channels in these applications using an ϵ value of 10. The results of this experiment are summarized in Figure 47. As we can see from this figure, Themis reports a total of seven vulnerabilities in four of the analyzed applications. We manually inspected each report and confirmed that the detected vulnerabilities are indeed true positives. We also reported the vulnerabilities detected by Themis to the developers, and the majority of these vulnerabilities were confirmed and fixed by the developers in less than 24 hours. However, the vulnerability that we reported for OpenMRS was rejected by the developers. The reason for this false positive is that the leaked password is actually hashed and salted in the database, but, because the logic for hashing and salting is not part of the Java implementation, Themis was not able to reason about this aspect.

4.3 OBSERVER: Solver-Guided Static Analysis

4.3.1 REXPLOITER: Availability Vulnerabilities in Programs Using Regular Expressions.

To assess the usefulness of the techniques presented in this work, we performed an evaluation in which our goal is to answer the following questions:

RQ1. Do real-world Java web applications use vulnerable regular expressions?

RQ2. Can REXPLOITER detect ReDOS vulnerabilities in web applications and how serious are these vulnerabilities?

Results for RQ1. In order to assess if real-world Java programs contain vulnerabilities, we scraped the top 150 Java web applications (by number of stars) that contain at least one regular expression from GitHub repositories (all projects have between 10 and 2, 000 stars and at least 50 commits) and collected a total of 2, 864 regular expressions. In this pool of regular expressions, Rexploiter found 37 that have worst-case exponential complexity and 522 that have super-linear (but not exponential) complexity. Thus, we observe that approximately 20% of the regular expressions in the analyzed programs are vulnerable. We believe this statistic highlights the need for REXPLOITER that can help programmers reason about the complexity of regular expression matching.

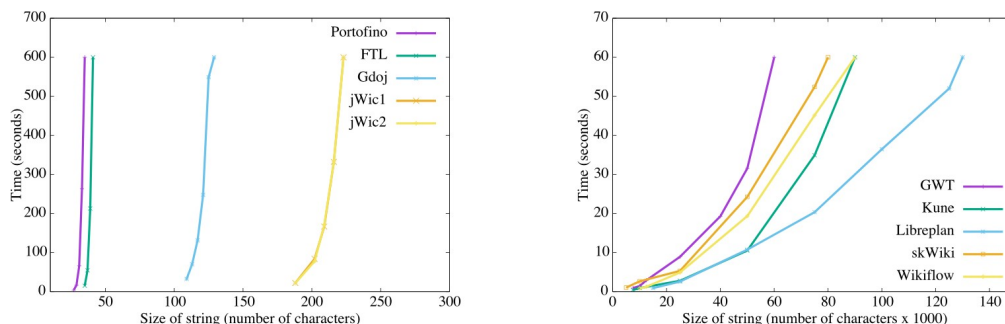


Fig. 8: Running times for exponential vulnerabilities (left) and super-linear vulnerabilities (right) for different input sizes.

Figure 48: Running times for exponential vulnerabilities (left) and super-linear vulnerabilities (right) for different input sizes

Results for RQ2. To evaluate the effectiveness of Rexploiter in finding ReDOS vulnerabilities, we used Rexploiter to statically analyze all Java applications that contain at least one vulnerable regular expression. These programs include both web applications and frameworks, and cover a broad range of application domains. The average running time of Rexploiter is approximately 14 minutes per program, including the time to dynamically analyze regular expressions. The average size of analyzed programs is about 58,000 lines of code.

Our main result is that Rexploiter found exploitable vulnerabilities in 27 applications (including from popular projects, such as the Google Web Toolkit and Apache Wicket) and reported a total of 46 warnings. We manually inspected each warning and confirmed that 41 out of the 46 vulnerabilities are exploitable, with 5 of the exploitable vulnerabilities involving hyper-vulnerable regular expressions and the rest being super-linear ones. Furthermore, for each of these 41 vulnerabilities (including super-linear ones), we were able to come up with a full, end-to-end exploit that causes the server to hang for more than 10 minutes. In Fig. 48, we explore a subset of the vulnerabilities uncovered by Rexploiter in more detail. Specifically, Fig. 48 (left) plots input size against running time for the exponential vulnerabilities, and Fig. 48 (right) shows the same information for a subset of the super-linear vulnerabilities.

Possible fixes. We now briefly discuss some possible ways to fix the vulnerabilities uncovered by Rexploiter. The most direct fix is to rewrite the regular expression so that it no longer exhibits super-linear complexity. Alternatively, the problem can also be fixed by ensuring that the user input cannot contain instances of the attack core. Since our technique provides the full attack automation, we believe Rexploiter can be helpful for implementing suitable sanitizers. Another possible fix (which typically only works for super-linear regular expressions) is to bound input size. However, for most vulnerabilities found by Rexploiter, the input string can legitimately be very large (e.g., review). Hence, there may not be an obvious upper bound, or the bound may still be too large to prevent a ReDoS attack. For example, Amazon imposes an upper bound of 5000 words ($\approx 25,000$ characters) on product reviews, but matching a super-linear regular expression against a string of that size may still take significant time.

4.3.2 SCANNER: Asymptotic Resource Side-Channel Vulnerabilities. We evaluate SCANNER by analyzing 10 widely-used open-source PHP applications. These applications include Word-Press and Joomla (content management systems), OpenClinic (a medical records system), Gallery (a web-based photo album organizer), OpenCart, ZeusCart, and osCommerce (e-commerce), WeBid (an online auction software), and HotCRP and OpenConf (conference management systems). Some of these applications contain information that is clearly security-sensitive, such as medical records, account balances, and purchase histories. Some of the applications we analyzed also provide well-documented privacy guarantees for information that may be considered security-sensitive. For example, Word-press explicitly states that email addresses of users will not be disclosed. We run our experiments on a server laptop with Ubuntu 14.04, a dual-core 2 GHz processor, 8 GB of RAM, and the Apache 2.4.18 web server, connected to a campus wireless network. The client is a desktop machine running Ubuntu 14.04, with a dual-core 3 GHz processor, 8 GB of RAM, and the Firefox browser.

ID	Positive Query		Negative Query	
	Avg Time	Std Dev	Avg Time	Std Dev
1	1,523.71	64.85	2,020.32	137.60
2	1,655.69	29.84	2,358.24	119.24
3	96.27	10.23	513.35	22.67
4	2,593.63	113.80	1,225.34	18.47
5	265.13	16.21	29.27	4.44
6	329.26	14.68	32.61	8.85
7	4,241.29	821.32	630.82	80.96
8	4,580.50	121.90	668.72	62.65
9	4,616.64	130.10	676.79	99.84
10	3,740.37	468.63	798.55	64.92
11	1,612.79	204.89	343.49	167.43
12	1,758.41	93.46	271.01	17.87
13	2,000.76	318.38	317.93	64.06
14	839.20	56.08	502.07	43.62
15	801.15	15.35	483.67	64.28
16	796.30	37.07	467.50	30.85
17	1,172.84	243.14	486.17	27.92
18	1,090.41	135.98	488.25	47.94

Figure 49: Summary of timing results for positive and negative queries

Overview of Results. Figure 50 gives statistics about the analyzed programs and summarizes

Application	Lines of Code	Files	Functions	Number of Callsites	Timing Vulnerabilities	Response Size Vulnerabilities	False Positives	Lines of Annotations	Analysis Time (min:sec)
OpenConf	24,581	133	152	4,493	0	0	0	39	1:52
OpenClinic	30,849	180	526	6,766	0	2	0	27	1:09
WeBid	48,753	336	652	8,042	1	1	0	53	4:01
HotCRP	57,799	125	2,540	18,367	0	3	1	32	3:47
Joomla	59,820	2,563	14,259	124,894	0	2	0	22	5:11
Gallery	62,699	505	1,959	16,027	0	0	0	13	8:26
osCommerce	86,663	702	195	22,273	0	0	0	53	10:13
OpenCart	156,322	1,014	3,717	53,010	1	0	0	98	27:48
ZeusCart	166,400	612	8,798	23,354	5	0	0	44	31:06
Wordpress	298,643	577	5,866	60,717	0	3	1	15	49:57
Total	992,529	6,747	38,664	337,943	7	11	2	396	143:30

Figure 50: Summary of our Experimental Results

the results of our evaluation. As shown in Figure 50, the analyzed programs are quite large and contain between 24K and 298K lines of code. SCANNER’s running time on these applications is quite reasonable, with the largest application taking 49 minutes to analyze. Most importantly, SCANNER reports a total of 20 vulnerabilities, 18 of which are indeed exploitable. Among the vulnerabilities uncovered by SCANNER, 7 are timing side-channel vulnerabilities, and the rest are response-size side channels.

Table 51 summarizes the information that the attacker can learn by exploiting the underlying vulnerabilities uncovered by SCANNER. The leaked information is in the form of yes/no queries. However, we emphasize that the underlying vulnerabilities actually allow an attacker to infer large parts of a database table by using these binary conditions as an oracle. For example, the vulnerability in OpenClinic can be exploited to infer all medications prescribed to a user. Similarly, an attacker can use the vulnerability in OpenCart to learn all e-books purchased by a user or the vulnerability in WeBid to infer all bids of a user. Hence, these programs violate multi-run security.

Application	ID	Information Leaked
OpenClinic	1	Does X have a medical record?
OpenClinic	2	Has X been prescribed medication Y?
WeBid	3	Does X have product Y on their watchlist?
WeBid	4	Is X bidding on product Y?
OpenCart	5	Has X bought downloadable product Y?
ZeusCart	6	Has X bought downloadable product Y?
ZeusCart	7	Does X have an order with amount Y?
ZeusCart	8	Has X bought anything between dates X and Y?
ZeusCart	9	Does X's order Y have processing status Z ?
ZeusCart	10	Is Y the account status for user X?
HotCRP	11	Is X an author of a submitted paper?
HotCRP	12	Is X the title of a submitted paper?
HotCRP	13	Is Z the abstract of a submitted paper?
Wordpress	14	Has X sent a private comment to the admin?
Wordpress	15	Is X the e-mail of a commentator in a post?
Wordpress	16	Is X the private comment sent to the admin?
Joomla	17	Has X authored a private article?
Joomla	18	Is X the title of a private article?

Figure 51: Information Inferred by the Attacker

Response-size Side-Channel in OpenClinic. Our first example is a response-size side-channel vulnerability found in OpenClinic, which is an open-source medical records system. This vulnerability reveals information about whether a patient has a medical record in the system. The OpenClinic application contains a medical record search page, which allows authorized users to search for medical records matching a given query string. The important point is that logical connectives are allowed to be included in the query string, and if the connective is set to “OR”, then the database query will return all rows for which the patient name matches any of the keywords in the search string. If the table does not contain patients matching the query, then the application prints No results found for X, where X is the search string provided by the user. On the other hand, if there is a record matching the query, then the application displays the patient’s record, whose length is independent of the search string. Therefore, one can learn whether the patient has a medical record or not by simply observing whether the response size is linear or constant with respect to the input string.

Note that the attacker cannot directly see the patient’s record since (a) he does not have access to the user’s credentials, and (b) the same-origin policy (SOP) prevents scripts contained in the attacker’s website from directly accessing data in OpenClinic. However, since the attacker can perform time measurements in the victim’s browser, he can observe the differences in time to infer the differences in response size.

Timing Side-Channel in ZeusCart. Our second example is a timing side-channel vulnerability that SCANNER found in the ZeusCart e-commerce system. Description of vulnerability. At a high level, ZeusCart allows a user with the right credentials to search order histories, which may include information like customer name, order ID, and the corresponding order status. If there is no matching purchase, the user quickly gets an empty results table in constant time. Otherwise, the application performs additional sanitization over the input whose running time is linear in the

size of the query string. Since there is a significant difference in response times depending on whether a user has a specific order status, an attacker who does not have login credentials can infer the existence of specific purchases by specific users by performing timing measurements in the victim's browser. One possible fix for this vulnerability is to sanitize the query string by ensuring that it does not exceed a certain number of characters. This modification would ensure that the response time is bounded by a constant.

Threats to Validity. In our evaluation, we use a specific machine and network environment when conducting timing measurements for positive and negative queries. Of course,

these measurements could be substantially different if the exploits were executed on other platforms. However, because the resource usage imbalances detected by SCANNER are always asymptotic, it is possible for the attacker to further inflate the resource imbalance by providing even larger inputs.

Another threat to validity is that we evaluate our technique on 10 PHP applications; hence, our benchmarks may not be representative of many other real-world PHP applications. Nevertheless, we believe these applications constitute a good test-bed for our approach: Many of the projects we picked (e.g. Wordpress, Joomla) are well-known popular web applications with thousands of stars on Github, and their security has been widely audited by the open-source community.

4.4 EXPLOITR: Explaining Potential Witnesses and Exploits

4.4.1 DPDEBUGGER: Debugging Differential Performance. Table 5 summarizes the results over eight case studies involving large Java applications, while Table 4 shows clustering and classification steps in DPDEBUGGER for seven of them (the eighth is in Figure 18). We next explain how DPDEBUGGER explained differences in performance for different classes of inputs for these seven applications.

1. **Charts4j.** Charts4j is a Java chart library that enables developers to generate different charts available in the Google Chart API. The input data set for our experiments consists of asking for different plots of the same data. Our tool applied spectral clustering with the number of clusters set to 3. It finds three functional clusters between execution time and the number of data items. The DRT for this experiment (shown in Table 4) gives two key insights: First, we notice that green cluster consists of plots that call *scale* function. Upon further investigation, the scale function is indeed expensive since it computes minimum and maximum for the input array and normalized values for each element of the input array. Next, the class initialization for `PlotImpl` distinguishes blue and red clusters. This happens for plots with one dimension where it needs to generate another dimension. It also needs to convert the new data set to a double array. This part is a performance bug because it can build a double array structure for the new dimension instead of double list.
2. **Snapbuddy.** SnapBuddy is a mock social network application where users can make their public profiles. As inputs for our experiments, we passively monitored users' interaction with public profile pages. We applied *K*-linear clustering algorithm where we set the number of clusters to 5. As a result, our tool finds five linear relationships between the size of public profile image and download time. The DRT learned by our tool is shown in Table 4. It reports that filter combinations applied in profile pictures are the key discriminants.

Table 4: Java applications studied using DPDEBUGGER

Charts-4j	Cluster	Discriminant Regression Tree
Name		
Snapshot		
JFreeChart		

Name	Cluster	Discriminant Regression Tree
Apache POI		
Odedj		
Collab		
Tweeter		

3. **JFreeChart.** JFreeChart is a free Java chart library that helps developers to plot different charts in their applications. The input data set is the set of Open-High-Low-Close (OHLC) items. We use this library to plot OHLC items with different renders. Our tool applied K -linear clustering algorithm with $K = 2$. The clustering step finds two linear relationships between time of execution and the number of OHLC items. The DRT produced by our algorithm is shown in Table 4. We observe that whether the plot applies `CandlestickRenderer` and calls `drawItem` distinguishes between red and blue clusters. In particular, the number of calls to `drawItem` is equal to the number of data points in OHLC dataset, and each call executes the loop statement when a candlestick renderer is applied. This performance bug was also reported in [2] using static analysis.
4. **Apache POI.** The Apache POI Project's mission is to read and write MS Excel, MS Word, and PowerPoint files using Java. As inputs to our experiments, we used different slides that randomly include text, table, images, shapes, and smartArts. We applied the spectral clustering algorithm with number of clusters set to 3. The DRT produced by our algorithm is shown in Table 4. The DRT shows that all program traces labeled with the blue cluster do not have any image data source (like PNG, smartArts, and so on). Green and red clusters are distinguished only based on their item sizes.
5. **Ode4j.** Ode4j is an open source, high performance library for simulating rigid body dynamics. The data set includes the different number of geom objects that interact with different APIs. We applied the spectral clustering algorithm with the number of clusters set to 2. The clustering finds two functional relationships between the time of execution and number of geom objects. The DRT (Table 4) reports `collide` function in `DxSAPSpace` class to distinguish between blue and red clusters. Looking into the source code, we can see the quadratic behavior in `collide` function. As there is a linear cost function named `collide2` in `DxSAPSpace` class, we suspect that quadratic behavior can be mitigated.
6. **Collab.** Collab is a scheduling application that allows users to create new events and modify existing events. The data set consists of different operations to add and modify events using `ADD`, `UNDO`, and `DONE` commands. We applied K -linear clustering algorithm with K equals to 2. It finds two linear relationships between the time of execution and the number of add operations. The DRT produced by our algorithm is shown in Table 4. The discriminant model shows that the traces in red cluster call `split` function more times than traces in blue cluster. The average number of calls to `split` function for blue cluster is 1,678 and for red cluster is 88,507. The temporary store data structure of Collab is a B-tree. The number of times that the B-tree needs to split a parent node is the main explanation to distinguish red and blue clusters.
7. **Tweeter.** Tweeter application is a mock of Twitter application. Users can post tweets and see tweets posted by other users. The data set consists of a maximum 20 words for a tweet. We applied K -linear clustering algorithm where we set K to 2. The clustering step finds two relationships between time of execution and number of words in a tweet. The DRT produced by our algorithm is shown in Table 4. The discriminant model produces `getAlternatives` function as a feature to distinguish blue and red clusters. This function is called more times when the input tweet includes mistakes, and it will never be called if the tweet is correct.

Table 5: Java applications studied using DPDEBUGGER. Legend: $\#M_T$: total number of functions in application, $\#M_R$: total number of observed functions, $\#N$: number of collected traces, C_T : Clustering algorithm (K for K -linear and S for spectral), T : computation time of DPDEBUGGER in seconds, A : accuracy of classification model, H : decision-tree height, $\#C$: Number of clusters, $\varepsilon < 0.1$ sec

Application	$\#M_T$	$\#M_R$	$\#N$	C_T	T	A	H	$\#C$
Apache FOP	39,694	2,765	1,988	S	14.4	96.4%	3	2
Charts4j	715	71	2,000	S	11.7	87.1%	2	3
SnapBuddy	3,071	150	616	K	1.6	88.6%	5	4
JFreeChart	9,162	527	1,000	K	1.6	99.4%	1	2
Apache POI	10,396	199	661	S	0.44	86.4%	2	3
Ode4j	4,564	114	577	S	7.8	85.0%	1	2
Collab	185	53	530	K	0.7	96.6%	1	2
Tweeter	947	31	320	K	0.6	100%	1	2

Table 6: Case Studies. Here $\#M$ is the number of methods in applications, $\varepsilon_{0,1}$ is the tolerance for L_1 -norm of the timing model, $\varepsilon_{1,1}$ is the tolerance for L_1 -norm of the first derivative of the timing model, A is the accuracy of the tree model, H is the height of the tree, $\#L$ is the number of leaf nodes in the tree, and T is the computation time for decision tree learning (s)

Benchmark	$\#M$	$\#R$	$\#S$	$\#P$	$\varepsilon_{0,1}$	$\#K_{0,1}$	$T_{0,1}$	$\varepsilon_{1,1}$	$\#K_{1,1}$	$T_{1,1}$	A	H	$\#L$	T
Regex	620	203	1,154	6,365	2e-1	162	1,801	2e-1	49	4,812	89.7%	14	120	2,084.0
Jetty	63	56	800	635	1e-1	20	49.7	1e-2	15	82.6	99.4%	12	20	52.1
iControl (SOAP)	41,541	127	342	1,164	1e-1	33	19.7	1e-1	19	43.2	98.2%	10	9	10.5
Javax (crypto)	612	56	1,533	1,045	1e-1	54	174.2	1e-1	32	253.0	88.6%	6	36	7.2
GabFeed	573	43	1,105	65	1e-1	34	58.5	1e-2	34	70.5	99.6%	31	34	41.7
Stegosaurus	237	96	512	60	2e-1	5	3.6	1e-1	3	3.6	100.0%	4	5	12.6
SnapBuddy	3,071	65	477	14	2e-1	13	2.8	2e-1	8	3.0	96.2%	14	13	3.1
ShareValue	13	7	164	41	6e-2	29	0.7	1e-2	14	0.7	99.3%	17	29	3.4
MST(Kruskal)	5	6	120	40	3e-1	20	0.4	3e-1	5	0.4	80%	7	20	3.0
Collab	185	53	176	11	1e-2	1	0.3	1e-2	1	0.3	N/A	N/A	N/A	N/A

4.4.2 FUCHSIA: Data-Driven Debugging for Functional Side Channels. Table 6 summarizes 10 Java applications used as case studies. We consider L_1 -norm distance between timing functions ($\varepsilon_{0,1}$) and their first derivatives ($\varepsilon_{1,1}$). The main research questions are “Do functional clustering and decision tree learning (a) scale well and (b) provide useful information about leaks?”

A) Regex. This example was already introduced and discussed in Figure 22. To answer the research question. The decision tree pinpoints a location in the regex package that leaks the value of secret patterns and the overall computation time of clustering and decision tree learning is about 65 mins.

B) Jetty. We analyze the `util.security` package of Eclipse Jetty web server. The package has a `Credential` class which had a timing side channel. This vulnerability was analyzed in [11] and fixed initially in [43]. Then, the developers noticed that the implementation in [43] can still leak information and fixed this issue with a new implementation in [44]. We consider this new implementation shown below and apply FUCHSIA to check its security. The final fix was done a

few months later [45], but before we reported our finding to the developers.

```
1  boolean stringEquals(String s1, String s2) {
2      if (s1 == s2) return true;
3      if (s1 == null || s2 == null) return false;
4      boolean result = true;
5      int l1 = s1.length(), l2 = s2.length();
6      if (l1 != l2) result = false;
7      int l = Math.min(l1, l2);
8      for (int i = 0; i < l; ++i)
9          result &= (s1.charAt(i) == s2.charAt(i));
10     return result;
11 }
```

C) iControl (SOAP). iControl (SOAP) is an open source API that uses SOAP/XML to establish communications between dissimilar systems. The library has 41,541 methods. One key confidentiality-related functionality of the library is to store credentials of various users and to validate their credentials against a given guess. The defender’s goal is to find out whether there exist timing side channels in the library, and if so, identify the code regions potentially responsible for creating the side channels.

The natural candidate for the secret input in this application is the stored credential at the server, while the public input is a given guess against a stored credential. The defender considers the lexicographic ordering over public inputs to generate functional data. With a timeout of two hours on fuzzing, the defender obtains 342 unique secret credentials and 1,164 unique guesses. The credentials include secret passwords with the length of at most 16 characters. FUCHSIA begins by varying 1,164 guesses for each secret input and uses B-spline to model 342 timing functions. With the default parameters (L_1 -norm as the distance norm and $\epsilon = 0.1$) for the clustering, FUCHSIA identifies 33 classes of observations. The presence of multiple clusters points towards the existence of timing side channels.

D) Javax Crypto. Javax library provides the classes and interfaces for cryptographic operations in Java. The `crypto` package in the library has 612 methods and provides functionalities such as creating and modifying symmetric secret keys. We analyzed the `crypto` package of javax library against timing side-channel vulnerabilities. The secret input is the symmetric secret key of encryption algorithms (such as “DES”), and the public input is a guess key to be compared against the secret key. During two hours of fuzzing, the defender generates 1,533 secret keys and 1,045 guess keys. The length of a key is at most 16 bytes. FUCHSIA identified 1,533 timing functions using B-spline basis and returned 54 clusters. The presence of 54 classes of observations indicates the existence of timing side channels.

E) GabFeed. GabFeed is a STAC application with 573 methods implementing a chat server. The server takes users’ public key and its own private key to generate a common key. The defender uses FUCHSIA to obtain 1,105 server’s private keys and 65 public keys where the public keys are ordered by their number of set bits. In total, there are 71,825 test cases. For each secret key, FUCHSIA varies public keys and measures the execution time to generate the common key. Next, FUCHSIA uses B-spline and creates timing functions for each secret. The next step is to find the equivalence relations over the secret input using the functional clustering. The defender provides L_1 -norm and $\epsilon_{0,1} = 0.1$ as parameters to the hierarchical clustering, and FUCHSIA discovers 34 clusters. This step partitions 1,105 secret values into 34 distinguishable classes.

F) Stegosaurus. Stegosaurus with 237 methods is a messaging service that uses steganographic algorithms to hide secret messages. The application takes the secret message with a 128-bit key and embeds the message in a random image. The secrets are the message and the key. The defender

uses FUCHSIA to generate 512 secret messages with a size of at most 8 letters. We assume that the secret key is a fixed random value chosen by the service. The public input is an image chosen by the defender and ordered by their scales. For each secret message, FUCHSIA varies the scale of images from 80 to 1,024. Then, it measures the execution time of the application to encode the message in different images. In total, FUCHSIA models 512 timing functions. The next step is to find the equivalence classes over these functions. The defender provides L_1 -norm and $\epsilon_{0,1} = 0.2$ as parameters to the clustering, and FUCHSIA finds 5 classes of observations.

G) SnapBuddy. SnapBuddy with 3,071 methods is a mock social network where each user has their own page with a photograph. The size of profiles is a public input (observable through the generated packets), and the identity of users actively interacting with the server is a secret input. The defender considers the identities of 477 users currently in the network as the secret inputs and varies the size of public profiles from 13KB to 350KB. FUCHSIA uses B-spline to model the profile retrieval times for each user as a function of profile sizes. FUCHSIA models 477 timing functions, one for each user. The next step is to find out the relationships between the timing functions of different users and determine if there are timing side channels. For this aim, FUCHSIA applies the clustering algorithm to identify different classes of observations. FUCHSIA discovers 13 clusters ($\epsilon_{0,1}=0.2$). The clustering partitions timing observations for 477 users into 13 equivalence classes.

H) Share Value. The application is an extension of classical share value program studied in [46, 47]. In this case, every user in the system has public and private shares. The program calculates useful statistics about shares. The program has 164 users each with maximum of 400 private shares. The user can have 1 to 400 public shares. FUCHSIA generates private and public shares in the given range. In particular, it fixes private shares for each user. Next, FUCHSIA varies the number of public shares for each user and measure the response times to calculate the statistics for each user. Then, it fits B-spline to the execution times and applies the functional clustering that discovers 29 clusters with $\epsilon_{0,1} = 0.06$.

I) Kruskal. We analyze Kruskal's algorithm and its implementation in [48]. Here, we assume that a graph data structure with Kruskal's algorithm is used in a security setting where the graph nodes are public and the structure of the graph (the connection of nodes) is secret. The input generation for Kruskal's algorithm is based on the domain knowledge of this problem. Given that the structure of graphs is secret, the defender constructs 4,800 graphs as the following. The defender considers 120 different graph structures from the interval between a spanning tree ($n-1$) and a complete graph ($n(n-1)/2$). For each structure, the number of nodes (n) varies from 2 to 200 and the number of edges is determined based on the structure of the graph. For example, if the structure of a graph is a spanning tree, the number of edges varies from 1 to 199. For each graph structure, FUCHSIA fits timing functions that are from the number of nodes to the execution time. Then, FUCHSIA applies the clustering algorithm and discovers 20 clusters with $\epsilon_{0,1} = 0.3$. The presence of the 20 clusters indicate the possibility of information leaks about the graph structure.

J) Collab. Collab is a scheduling application that allows users to create a new event and modify existing ones. Users can apply *add*, *commit*, and *search* operations on events. An audit event is a secret, while other events are public. The defender considers 176 users in the system, each with either zero or one audit events. The public inputs are the operations performed on the public events of users. For each user, FUCHSIA applies 1 to 11 operations randomly from the set of possible operations on their public events and measure the response times. FUCHSIA models 176 timing functions, one for each user in the system. The next step is to find out the classes of observations on these functions. FUCHSIA discovers only one cluster with a small tolerance value, and the

defender concludes that no information about the audit events of users is leaking through timing side channels. The clustering algorithm takes about 1(s).

5.0 CONCLUSIONS

This section summarizes key conclusions on the major thrusts of the four major components (LOOPR, PATHR, OBSERV, EXPLOITR) of our tool chain.

5.1 LOOPR: CodeHawk based conservative and highly scalable static analysis

The CodeHawk Java Analyzer was first developed as a new front end to the CodeHawk Abstract Interpretation Engine as part of the IARPA StoneSoup project, in which Kestrel Technology participated as a member of Kestrel Institute's VIBRANCE team. During that project the core infrastructure was implemented and analysis capabilities were developed for a select number of numeric and symbolic properties related to the CWE's targeted in that project. The role of the CodeHawk Java Analyzer was to establish the safety from a well-defined set of vulnerabilities in as many program locations as possible to minimize the need for runtime instrumentation and to minimize the incurred performance overhead of such instrumentation. The position of the analyzer in the VIBRANCE project was the first step in a pipeline of tools that produced an automatic hardening of the application of interest. Thus the workflow requirements on the analyzer were straightforward: parsing, followed by analysis for targeted properties, followed by the reporting of a list of safe locations in the application.

The target functionality of the analyzer in the AuditR project, in contrast, was much more open-ended, as denial-of-service and side-channel disclosure vulnerabilities do not have a straightforward translation into well-defined program properties, like absence of integer-overflow or null-pointer dereference. The goal in the AuditR project was to focus the attention on sections of the code most likely to contribute to availability or confidentiality weaknesses, to allow other tools to construct inputs to demonstrate the possibility of exploitation. These requirements required a redesign of the system architecture to allow for higher levels of interactivity and more flexibility in accessing and use of the analysis results. In particular, queries needed to be fast, precluding performing analysis as part of the query. Thus one of the major changes we implemented was to save all analysis results, including all invariants at all locations in the program, and to have queries operate on this data. To reduce the size of this data we constructed an encoding that applied aggressive data sharing and indexing, resulting in a reduction of about two orders of magnitude compared with an explicit data storage. To provide maximum accessibility and facilitate custom queries it was decided to construct a python infrastructure to provide general accessors to all of the data by means of objects mirroring the analysis artifacts, such that creating query scripts would not require an understanding of the serialization format, but only knowledge of the relevant analysis results.

The following efforts will improve scalability and usability of the cost profiling part of the LOOPR.

- **Cross-validation with model predictions.** Given the byte code timings we can construct small methods consisting of various combinations of byte codes and compare their execution times with the execution times predicted by our cost model. This would probably require construction of class files manually, rather than having them generated by the Java compiler

from source, to have sufficient control over the exact composition of the method. At present CodeHawk does not include facilities to construct or transform class files. Other tools such as ASM [49] or BCEL [50] could be used to accomplish this. Having a collection of such measurement / prediction pairs would enable to obtain context-dependent timing values or timing values for particular combinations of byte codes rather than individual byte code instructions. Achieving this would probably involve significant engineering effort.

- **Cross-validation with real-world code.** To validate our results it would be interesting to compare the profiling results with the corresponding model predictions. While we have done so for a few individual methods, we have not yet performed this systematically on scale.
- **Automatic regeneration of timings.** All our profiling results pertain to the STAC reference platform. It would be useful to have the ability to regenerate timings and integrate them with the analysis automatically. This would aid in the speed with which we can expand the set of profiled library methods as well as allow us to extend our cost analysis, for both library methods and byte codes, to platforms other than the STAC reference platform.
- **More complex timing cost expressions.** The majority of library methods are fairly simple and their timing cost can be adequately represented by either a constant or a linear expression in one or more of the argument values. Of course we still encounter a large number of methods whose timing behavior may have other relationships with respect to arguments or field values. Our techniques can be expanded to discover these relationships.

Timing behavior may also be conditional, as a result of the existence of different paths through a method depending on the arguments. At present we cover this case by representing the cost by a timing range. Cost analysis could be improved considerably if we allow timing expressions to be conditional. We have most of the infrastructure available to handle conditional expressions in other contexts, such as safety conditions for methods, but have not yet integrated this into the cost framework.

- **Exceptions.** We have performed preliminary investigations of profiling techniques to capture the effect of exceptions, both setting up exception frames and throwing exceptions. Preliminary results indicate a need to take into account the depth of the call stack or the distance between where an exception is thrown and where it is caught. Similarly for try-finally blocks. More research is needed to be able to adequately incorporate these effects.

In the process of profiling we actually discovered some anomalies that could lead to Denial-of-service vulnerabilities. For example a call to `java.util.logging.Level.equals` takes orders of magnitude less time when called with an object that is another `Level` object than when it is not a `Level` object. The reason is that the call is optimized for the common case and throws and catches an exception when it receives an object of different type. It would be interesting to create an analysis targeted at identifying such constructs.

- **JIT Compilation.** We have not considered the effects of JIT compilation in our profiling studies. More research is needed to detect and represent these effects.

5.2 PATHR: A precise path- and context-sensitive static analysis

5.2.1 QUANTM: Type-directed Bounding of Collections in Reactive Programs. In this thrust, we proposed a technique that statically verifies the boundedness of total length of collection variables when given constraint(s) on input(s). Our technique is able to verify the above property for non-terminating reactive programs. To ensure scalability, we take a type-based approach and enforce using global inductive invariants, as opposed to different invariants at different program locations. To design global invariants for programs supporting mutation, we introduce AST counters, which track how many times an AST node was executed. We then add axioms on relations of the counter variables. Experimental results demonstrate that our technique is scalable, and effective at verifying bounds. We plan to build on this work in at least the following two directions: (i) extending from collection lengths to general memory usage, (ii) generalizing the AST counter technique and applying it in different contexts.

5.2.2 REXPLOITER: Availability Vulnerabilities in Programs Using Regular Expressions.

We have developed an end-to-end solution for statically detecting regular expression denial-of-service vulnerabilities in programs. Our key idea is to combine complexity analysis of regular expressions with safety analysis of programs. Specifically, our regular expression analysis constructs an attack automaton that recognizes all strings that trigger worst-case super-linear or exponential behavior. The program analysis component takes this information as input and performs a combination of taint and string analysis to determine whether an attack string could be matched against a vulnerable regular expression. We have used our tool to analyze thousands of regular expressions in the wild and we show that 20% of regular expressions in the analyzed programs are actually vulnerable. We also use Rexploiter to analyze Java web applications collected from Github repositories and find 41 exploitable security vulnerabilities in 27 applications. Each of these vulnerabilities can be exploited to make the web server unresponsive for more than 10 minutes. There are two main directions that we would like to explore in future work: First, we are interested in the problem of automatically repairing vulnerable regular expressions. Since it is often difficult for humans to reason about the complexity of regular expression matching, we believe there is a real need for techniques that can automatically synthesize equivalent regular expressions with linear complexity. Second, we also plan to investigate the problem of automatically generating sanitizers from the attack automata produced by our regular expression analysis.

5.3 OBSERV: Solver-Guided Static Analysis

5.3.1 THEMIS: Precise Detection of Side-Channel Vulnerabilities. We have proposed a new security policy called ϵ -bounded non-interference that can be used to verify the absence of resource side channels in programs. We have also proposed an automated verification algorithm, implemented in a tool called Themis, for proving this property. Our approach verifies the absence of side channels by combining lightweight static taint analysis with precise relational verification using a new program logic called QCHL [11]. We have evaluated our tool, Themis, in a number of ways and have shown that (a) it can find previously unknown vulnerabilities in widely used Java programs, (b) it can verify that the repaired versions of vulnerable programs do not exhibit the original vulnerability, and (c) it compares favorably against Blazer, a state-of-the-art tool for finding timing side channels in Java programs. There are a number of directions for future work. First,

our current implementation only provides cost models for timing and response size; however, we would like to broaden the applicability of Themis by providing cost models for other kinds of resources. Second, our current implementation can verify bounded non-interference for a given ϵ , but we would also like to automatically infer the smallest ϵ for which the program is safe. While this extension can be easily done by solving an optimization rather than a satisfiability problem, our current implementation does not provide this capability due to limitations in Z3's OCaml API.

5.3.2 SCANNER: Asymptotic Resource Side-Channel Vulnerabilities. In this foundational work, we introduced asymptotic resource side channels and described a static analysis for detecting them. Our method uses an effective, but lightweight program abstraction that combines taint information with a summary of resource usage. We have used our tool SCANNER to analyze 10 open-source widely-used PHP applications and found 18 exploitable security vulnerabilities. We believe that SCANNER can help web application developers by automatically uncovering easy-to-exploit vulnerabilities in their applications.

5.4 EXPLOITR: Explaining Potential Witnesses and Exploits

5.4.1 DPDEBUGGER: Debugging Differential Performance. We propose *discriminant regression trees* which are regression trees where the goal is to classify input data into a small number of clusters. We give a new algorithm for learning discriminant regression trees. It finds (functional) clusters first, which enables learning the tree using an efficient algorithm for learning decision trees. We present extensions to two classical clustering algorithms k -means and spectral clustering. These extensions allow us to obtain functional clusters. We implemented this approach in the tool DPDEBUGGER and evaluated it on benchmarks consisting of a suite of Java programs. Our experiments show that the approach is scalable and is able to explain the differences in performance between different classes of inputs.

5.4.2 FUCHSIA: Data-Driven Debugging for Functional Side Channels. We focused on the known-message setting under the assumption that secret inputs are less volatile than public inputs. In this setting, the observations appear as timing functions. We propose a notion of noninterference in the functional setting and show that it allows defenders to detect side channels using functional data clustering. We propose decision tree algorithms to pinpoint locations in the program that contribute to the side channels. Our tool FUCHSIA scales well for large real-world applications and aids debuggers to identify vulnerable fragments in such applications.

This work opens potential promising directions for future work. One direction is to combine the fuzzer with clustering that can directly estimate the number of distinguishable observations during the input generations. In this case, the objective is to find n secret values and m public values and maximize the number of distinguishable clusters in timing observations. Another direction is to study the potential timing side channels for machine learning applications. Given a learning problem with n samples and m features as public inputs, the feasibility of leaking hyper-parameters [51] of machine learning models via timing side channels is a relevant and challenging problem.

6.0 REFERENCES

- [1] Tianhan Lu, Pavol Cerný, Bor-Yuh Evan Chang, and Ashutosh Trivedi. Type-directed bounding of collections in reactive programs. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, volume 11388 of *Lecture Notes in Computer Science*, pages 275–296. Springer, 2019.
- [2] Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *ACM SIGPLAN Notices*, volume 50, pages 369–378. ACM, 2015.
- [3] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th international conference on Software engineering*, pages 151–160. ACM, 2008.
- [4] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM Sigplan Notices*, volume 45, pages 160–173. ACM, 2010.
- [5] Quentin Carbonneaux, Jan Hoffmann, and Zhong Shao. Compositional certified resource bounds. *ACM SIGPLAN Notices*, 50(6):467–478, 2015.
- [6] Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, et al. Proving termination of programs automatically with approve. In *International Joint Conference on Automated Reasoning*, pages 184–191. Springer, 2014.
- [7] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In *ACM Sigplan Notices*, volume 44, pages 375–385. ACM, 2009.
- [8] Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In *ACM Sigplan Notices*, volume 45, pages 292–304. ACM, 2010.
- [9] Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable approach to bound analysis and amortized complexity analysis. In *Computer Aided Verification-26th Int. Conf.(CAV'14)*, pages 743–759, 2014.
- [10] Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *International Static Analysis Symposium*, pages 280–297. Springer, 2011.
- [11] Jia Chen, Yu Feng, and Isil Dillig. Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, page 875–890, New York, NY, USA, 2017. Association for Computing Machinery.
- [12] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In *Tools and Algorithms for the Construction and Analysis of Systems -*

25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I, volume 11427 of *Lecture Notes in Computer Science*, pages 41–58. Springer, 2019.

- [13] Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. Local search for fast matrix multiplication. In *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 155–163. Springer, 2019.
- [14] Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Clausal proofs of mutilated chessboards. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 204–210. Springer, 2019.
- [15] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving very hard problems: Cube-and-conquer, a hybrid SAT solving method. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 4864–4868. ijcai.org, 2017.
- [16] Marijn Heule, Manuel Kauers, and Martina Seidl. A family of schemes for multiplying 3x3 matrices with 23 coefficient multiplications. *ACM Commun. Comput. Algebra*, 53(3):118–121, 2019.
- [17] Valentin Wüstholtz, Oswaldo Olivo, Marijn J. Heule, and Isil Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, page 3–20, Berlin, Heidelberg, 2017. Springer-Verlag. ISBN 9783662545799. URL https://doi.org/10.1007/978-3-662-54580-5_1.
- [18] J. Chen, O. Olivo, I. Dillig, and C. Lin. Static detection of asymptotic resource side-channel vulnerabilities in web applications. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 229–239, 2017.
- [19] Saeid Tizpaz-Niari, Pavol Cerný, and Ashutosh Trivedi. Data-driven debugging for functional side channels. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.
- [20] Saeid Tizpaz-Niari, Pavol Cerný, Bor-Yuh Evan Chang, and Ashutosh Trivedi. Differential performance debugging with discriminant regression trees. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 2468–2475. AAAI Press, 2018.
- [21] Saeid Tizpaz-Niari, Pavol Cerný, and Ashutosh Trivedi. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *ISSTA '20: 29th ACM*

SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020, pages 189–199. ACM, 2020.

- [22] Saeid Tizpaz-Niari, Pavol Cerný, and Ashutosh Trivedi. Quantitative mitigation of timing side channels. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 140–160. Springer, 2019.
- [23] Saeid Tizpaz-Niari, Pavol Cerný, Sriram Sankaranarayanan, and Ashutosh Trivedi. Efficient detection and quantification of timing leaks with neural networks. In *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, volume 11757 of *Lecture Notes in Computer Science*, pages 329–348. Springer, 2019.
- [24] Saeid Tizpaz-Niari, Pavol Cerný, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Ashutosh Trivedi. Discriminating traces with time. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, volume 10206 of *Lecture Notes in Computer Science*, pages 21–37, 2017.
- [25] Saeid Tizpaz-Niari. *Differential Performance Debugging and its application to side-channel analysis*. PhD thesis, University of Colorado Boulder, 2020.
- [26] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi: 10.1145/512950.512973. URL <http://doi.acm.org/10.1145/512950.512973>.
- [27] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.
- [28] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski, editors, *POPL*, pages 84–96. ACM Press, 1978.
- [29] Gogul Balakrishnan and Thomas W. Reps. DIVINE: Discovering variables in executables. In Byron Cook and Andreas Podelski, editors, *VMCAI*, volume 4349 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2007. ISBN 978-3-540-69735-0.
- [30] Henny B. Sipma. Sound Static Analysis of C Programs and x86 Executables using Abstract Interpretation. Final report, DHS-sponsored, AFRL-contract: FA8750-12-C-0277, Kestrel Technology, LLC, 2016.
- [31] Henny B. Sipma. Java Native Methods: A Need for Multi-Language Static Analysis. Final Report, IARPA-sponsored project on Java Native Methods FA8650-14-C-7425, Kestrel Technology, LLC, 2015.

- [32] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *ACM SIGPLAN Notices*, volume 43, pages 159–169. ACM, 2008.
- [33] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [34] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '99*, page 13. IBM Press, 1999.
- [35] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 661–667, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [36] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. In *OOPSLA*, pages 561–578, 2014. doi: 10.1145/2660193.2660234. URL <http://doi.acm.org/10.1145/2660193.2660234>.
- [37] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
- [38] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.
- [39] Wei-Yin Loh. Regression tress with unbiased variable selection and interaction detection. *Statistica Sinica*, pages 361–386, 2002.
- [40] Werner Dietl, Stephanie Dietzel, Michael D Ernst, Kivanç Muslu, and Todd W Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690. ACM, 2011.
- [41] Matthew M Papi, Mahmood Ali, Telmo Luis Correa Jr, Jeff H Perkins, and Michael D Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212. ACM, 2008.
- [42] Scala library for parsing and printing the smt-lib format. <https://github.com/regb/scala-smtlib>.
- [43] Timing side-channel on the password in eclipse jetty, May 2017. <https://github.com/eclipse/jetty.project/blob/f3751d70787fd8ab93932a51c60514c2eb37cb58/jetty-util/src/main/java/org/eclipse/jetty/util/security/Credential.java#L81>.
- [44] Timing side-channel on the length of password in eclipse jetty, May 2017. <https://github.com/eclipse/jetty.project/commit/2baa1abe4b1c380a30deaccaled367466a1a62ea>.

- [45] Fixed timing side-channel on the length of password in eclipse jetty, August 2017. <https://github.com/eclipse/jetty.project/commit/a7e8b4220a410b85c843bffc13f07d70f1b3fe8>.
- [46] Johan Agat. Transforming out timing leaks. In *POPL*, pages 40–53. ACM, 2000.
- [47] Heiko Mantel and Artem Starostin. Transforming out timing leaks, more or less. In *ESORICS*, pages 447–467. Springer, 2015.
- [48] Kruskal’s algorithm implementations. <https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>.
- [49] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.
- [50] BCEL. Bcel:bytecode engineering library. <https://commons.apache.org/proper/commons-bcel/>.
- [51] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. In *IEEE Symposium on Security and Privacy*, pages 36–52. IEEE, 2018.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

API Application Programming Interface 24, 25, 63, 67, 71, 73, 80

AST Abstract Syntax Tree 5–7, 30, 31, 79

CHIF CodeHawk Internal Form 10–13

CHL Cartesian Hoare Logic 7

DARPA Defense Advanced Research Projects Agency 12, 61

DoS Denial-of-Service 5

DRT Discriminant Regression Tree iii, 9, 40–42, 71, 73

IARPA The Intelligence Advanced Research Projects Activity 11, 12, 24, 77

OHLC Open-High-Low-Close 73

POJO Plain Old Java Objects 48

QCHL Quantitative Cartesian Hoare Logic 7, 31–34, 65, 67, 79

ReDOS Regular Expressions Denial-of-Service iii, 7, 35, 67, 68

RQ research question 61, 64, 65, 67, 68

SMT Satisfiability-Modulo-Theory 5, 29, 31, 33, 34, 39, 61, 64

STAC Space/Time Analysis for Cybersecurity 1, 7–9, 12, 24, 61, 65, 78

GLOSSARY OF TERMINOLOGY

Availability An availability problem arise when a user-provided input controls the asymptotic complexity or termination of a program component. 1

Confidentiality A confidentiality problem arise when two different user inputs result in substantially different resource usage behaviors depending on the value of some confidential data stored in the system. 1

false negative a test result which incorrectly indicates that a particular condition or attribute is absent. 1

false positive A test result which incorrectly indicates that a particular condition or attribute is present. iv, 1, 2, 4, 8, 66, 67