AFRL-RI-RS-TR-2018-229

SOUND OVER- & UNDER-APPROXIMATIONS OF COMPLEXITY & INFORMATION SECURITY (SOUCIS)

THE UNIVERSITY OF MARYLAND

SEPTEMBER 2018

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

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-2018-229 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
SCOTT F. ADAMS
Work Unit Manager

/ S /
JAMES S. PERRETTA
Acting Deputy Chief, Information
Exploitation and 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

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
SEPTEMBER 2018	FINAL TECHNICAL REPORT	Apr 2015 – Apr 2018

4. TITLE AND SUBTITLE

SOUND OVER- & UNDER-APPROXIMATIONS OF COMPLEXITY & INFORMATION SECURITY (SOUCIS)

5a. CONTRACT NUMBER
FA8750-15-2-0104

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Michael Hicks, David Van Horn, Jeff Foster, Eric Koskinen, Dawn Song, Timos Antopoulos

5d. PROJECT NUMBER
STAC

5e. TASK NUMBER
MA

5f. WORK UNIT NUMBER
RY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The University of Maryland
3112 Lee Bldg 7809 Regents Drive
College Park, MD 20742-0001

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2018-229

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

The technical keystones of this initiative were the use of sound over-approximating static analysis in conjunction with precise under-approximating analysis. For the former, new static analysis techniques for inferring program invariants in conjunction with a new technique for revealing side channels and complexity attacks in Java programs were developed. For the latter, new randomized, fuzz testing and machine learning techniques for vulnerability identification were developed. The state of the art in both areas was systematically surveyed and results were found that challenged previously published conclusions. A collaborative workbench application was developed to organize an analyst's task in using the tools.

15. SUBJECT TERMS
Software analysis, code analysis, vulnerability detection, static analysis, dynamic analysis, fuzzing, fuzzer, fuzz testing, side channel attacks, complexity attacks, Java, machine learning, information security, timing channels, taint analysis

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			SCOTT F. ADAMS
U	U	U	UU	75	19b. TELEPHONE NUMBER (*Include area code*) 315-330-4894

TABLE OF CONTENTS

List of Figures

List of Tables

Foreword

> In computer security, a side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself (e.g. cryptanalysis and software bugs). Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited.

–Wikipedia (accessed June 13, 2018)

> An algorithmic complexity attack is a form of computer attack that exploits known cases in which an algorithm used in a piece of software will exhibit worst case behavior. This type of attack can be used to achieve a denial-of-service.

–Wikipedia (accessed June 13, 2018)

Acknowledgements

This work was made possible by not just the efforts of PIs, but also their graduate students and post-doctoral researchers. We especially want to acknowledge the leadership and contributions of Andrew Ruef, Shiyi Wei, Paul Gazzillo, and Chang Liu.

This report resulted from research sponsored by the Defense Advanced Research Projects Agency (DARPA) for the Space/Time Analysis for Cybersecurity (STAC) program, under Air Force Research Laboratory (AFRL) agreement number FA8750-15-2-0104.

Disclaimer

1 SUMMARY

Critical components of national cyberinfrastructure have been hardened to withstand traditional exploit-based attacks that target erroneous program behavior to compromise security. Looking forward, such systems will be subject to a new class of attacks that do not rely on errors but rather exploit the inherent resource usage of the services the infrastructure provides. Such attacks have the potential to cripple critical services that are nonetheless correct. Program analysis has proved to be a powerful technique for detecting erroneous software behavior. However, termination/liveness, resource usage bounds, and side-channel capacity are increasingly difficult refinements of the basic program analysis problem. Prior work failed to meet the requirements of speed and scale for the systematic analysis of large scale software systems and has not focused on the domain of exploitable resource usage patterns.

This report presents the technical accomplishments of the project **SOUCIS: Sound Over- & Under-Approximations of Complexity & Information Security**. The technical keystones of the project are the use of *sound over-approximating static analysis* in conjunction with *precise under-approximating analysis*. For the first, we employed a program *verification* perspective: over approximation techniques that attempt to guarantee the absence of vulnerabilities. We have developed sound *static analyses* that abstractly characterize all of the possible behaviors of the program and then attempt to mathematically prove that this abstraction is free of vulnerabilities. However, the competing needs of both scalability and precision are known to be problematic for such analyses. This problem motivated our development of more precise, scalable techniques that may fall short of characterizing all possible executions. These include randomized, *fuzz testing* and *machine learning*. We also developed *hybrid techniques* providing the benefits of both thrusts: tests over some runs induce hypotheses that can be verified correct for all of them. To ease the task of using these techniques together, we developed a collaborative *workbench* that helps invoke various tools, visualize the results, and store and share them for analyst collaboration.

The results of the SOUCIS project include code artifacts, technical reports, and published papers appearing in peer-reviewed scientific venues.

2 INTRODUCTION

Critical components of national cyberinfrastructure have been hardened to withstand traditional exploit-based attacks that target erroneous program behavior to compromise security. Looking forward, such systems will be subject to a new class of attacks that do not rely on errors but rather exploit the inherent resource usage of the services the infrastructure provides. Such attacks have the potential to cripple critical services that are nonetheless correct. Program analysis has proved to be a powerful technique for detecting erroneous software behavior. However, termination/liveness, resource usage bounds, and side-channel capacity are increasingly difficult refinements of the basic program analysis problem.

The overall setup of the Sound Over- and Under-Approximations of Complexity and Information Security (SOUCIS) project was to enable sound, highly automated program analysis for the elimination of complexity and side-channel vulnerabilities in applications for

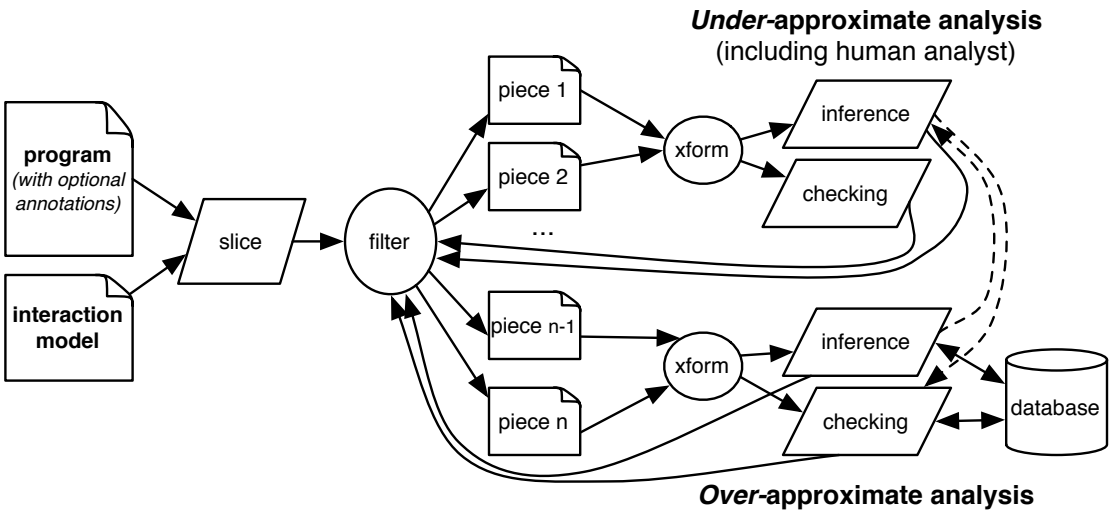


Figure 1: Analysis Overview

which only bytecode source is available. Importantly, the approach aimed to combine several analysis techniques, taking the best features of each to create a better whole.

Over its three years, the project advanced the state of the art of several techniques. Through several systematic, empirically-informed studies, we uncovered key features lacking in existing work, and pointed to new directions for exploration. The project also improved the current practice of code review by providing new tools for analysts. Technical papers derived from the SOUCIS project were published in top venues, and important code artifacts have been publicly released.

The next section of this report presents a technical overview of the SOUCIS project, organizing its various lines of inquiry. The following section details the actual research results while going into greater technical detail.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

The workflow of the SOUCIS approach to finding side channels and complexity attacks is presented in Figure 1.

At the start, the analyst provides the program to analyze along with an interaction model. The analyst provides annotations to the input program, e.g., to identify which data is secret. The interaction model describes how we expect the potential adversary to interact with the program; this is sometimes called the harness. The next step is slicing. The input program could be large, and yet the attacker-controlled input and/or the secrets to protect may have a relatively meager influence on program execution. This can be performed with the assistance of a taint analysis or by hand. Then there are two broad kinds of analysis: an over-approximating (e.g., static) analysis and an under-approximating (e.g., dynamic) analysis.

3.1 Overapproximate Analysis

On the static analysis side, SOUCIS advanced the use of static analysis techniques for identifying timing side channels. The core technique is found in our work on the Blazer tool [4]. Blazer establishes a novel *decomposition* methodology. The key insight is that we can break down a program into pieces and prove freedom from timing channels in each piece, making the problem more tractible. Blazer brings together control-flow analysis, information-flow analysis, static bounds analysis, and abstract interpretation. It combines the information from these analyses in a novel algorithm that partitions and checks subprograms to prove freedom from vulnerable timing channels or find where a potential leak may be found. This work is discussed in Section 4.1.

Information flow analysis is a crucial component of the Blazer approach (and that of other, related projects), as it identifies key portions of the program that are subject to attack. Unfortunately, we were never able to arrive at a satisfactory solution. Our assumption was that we could use any one of several existing, publicly available tools. Many such tools exist, with published papers about them showing impressive results. However, our experience with three tools was that each had significant limitations, thus pointing to the need for further research. We detail our observations in this regard in Section 4.1.3.

Another crucial piece of Blazer is its use of *abstract interpretation* [5, 6] to infer numeric invariants about a target program. These invariants feed into the determination of resource usage which could either depend on secret values or be manipulated by an attacker. A key challenge for abstract interpretation in this context is the need to provide precise invariants for real-world Java programs. While abstract interpretation is well studied, and existing tools exist for Java programs, we were surprised to discover that no existing tool handled full Java. Real-world Java programs are large, have many interacting methods, and make heavy use of heap-allocated objects. To support such programs, we developed one of the few fully automated numeric static analyses for Java. Developing this tool was a significant technical challenge, in part because there are many design decisions whose interaction is not well understood. For example, while the precision/performance tradeoff of different implementations of numeric domains is understood, the interaction of that domain with different ways of implementing a heap model is not. To put our tool on firm footing, we implemented a variety of options and systematically evaluated them on real-world programs [7]. In total, we studied 162 analysis configurations to assess both how individual configuration options performed overall and to study interactions between different options. Details about the different configurations and the results of our study are given in Section 4.2.

A precise, scalable static analysis is hard to develop, and our study confirmed that. As such, we also explored a hybrid analysis, with static and dynamic components, for inferring numeric invariants [8]. This analysis, called NumInv, dynamically infers expressive polynomial equality invariants and linear inequality relations from traces at specified program locations. It also infers candidate equality invariants. Such invariants are then confirmed as correct for all inputs using a static verifier. If a candidate invariant is incorrect the verifier returns counterexample traces, which the dynamic inference engine can use to infer more accurate invariants. This iterative process is called *CounterExample Guided Invariant geneRation*. The final, correct invariants can be fed into Blazer. We also found that by instrumenting the program with a resource counter, NumInv could also infer an invariant that

reflects running time, directly. Section 4.3 describes NumInv's algorithm in greater depth, and our empirical results.

3.2 Underapproximate Analysis

The underapproximate analyses in SOUCIS aim to be more precise and scalable, but risk missing bugs as they may not apply to all program runs. As such, they are best for identifying the possibility of bugs, rather than proving their absence. We considered two basic approaches: *fuzz testing* and *machine learning*.

SOUCIS brought fuzz-testing approaches to the problems of algorithmic complexity and side-channel vulnerabilities. While testing approaches are underapproximate, i.e., they are not guaranteed to find a vulnerability if one exists, such approaches have the benefit of finding individual program executions that lead to security violations. Moreover, fuzz-testing can handle complicated control flow that may be difficult for static analysis tools.

There were several challenges to enabling a fuzzing approach to side-channel and complexity attack analysis. The vulnerable portion of a large program may be very small, so the fuzzer needs to be guided towards potential vulnerabilities, rather than randomly selecting inputs. As such, our fuzzer (following the lead of libfuzzer [9]) supports the use of *harnesses* to focus fuzzing effort at internal methods. Client-server applications require new techniques for fuzzing server code. In particular, servers are interactive not batch oriented, and so may communicate via sockets or a User Interface (UI), rather than files. We built support for *Input/Output (I/O) mocking* to allow this sort of interaction. In addition, servers are often *stateful*: one interaction may change the state of the server, causing a subsequent interaction to behave differently than it might have otherwise. To ensure repeatability and systematic exploration we implemented efficient support for *memory checkpointing*. Finally, we organized our fuzzing architecture to support multiple runs and checks of relational properties across runs. Doing so permits determining the presence of side channels. Section 4.4 discusses the work on fuzzing.

Fuzz testing has enjoyed great success at discovering security critical bugs in real software. Recently, researchers have devoted significant effort to devising new fuzzing techniques, strategies, and algorithms. Such new ideas are primarily evaluated experimentally so an important question is: What experimental setup is needed to produce trustworthy results? We surveyed the recent research literature and assessed the experimental evaluations carried out by 32 fuzzing papers. We found problems in every evaluation we considered. We then performed our own extensive experimental evaluation using an existing fuzzer. Our results showed that the general problems we found in existing experimental evaluations can indeed translate to actual wrong or misleading assessments. We developed some guidelines that we hope will help improve experimental evaluations of fuzz testing algorithms, making reported results more robust [10]. We summarize our results, and recommendations for improving future evaluations, in Section 4.5.

SOUCIS also developed a machine learning-based approach to identify vulnerabilities. The general idea is to develop a code-similarity functionality approximated by a neural network. In doing so, we can create a database of vulnerable code samples, and examine whether a new binary function is similar to any existing vulnerable code sample to detect potential vulnerabilities. We showed that our approach is both more effective and more

efficient than previous approaches on the vulnerability detection task [11]. More details can be found in Section 4.6.

3.3 Workbench

To streamline the analyst's workflow when considering whether a target program has a side channel or complexity-attack vulnerability, we developed a tool called the *Workbench*. This browser-based tool permitted the analyst to organize the target programs in a workflow, and to kick off analysis tools (SOUCIS tools, and others from off the shelf) and organize the results. Initially, the static analysis tools generated and displayed web pages for an analyst to inspect the results of the static analysis. But there was little interaction, and no ability to invoke the tools. Ultimately, we developed a distributed analysis Workbench comprising the following components: (i) a single Controller node, acting as the user interface and task dispatcher; (ii) a single, though potentially distributable, database that holds all inputs and outputs, including intermediate results; and (iii) multiple Worker nodes that perform the actual tasks. The Workbench is discussed in Section 4.7. During the engagements, we used a combination of command-line tools, the graphical output of the tools, as well as the workbench to help run and collect data on the results of analyzing the challenge programs.

4 RESULTS AND DISCUSSION

4.1 Static Analysis for Timing Channels

We are interested in verifying that programs are free of timing channels. A timing channel is a 2-safety property: to prove its absence we have to show that for any pair of executions involving the same public inputs but different secret inputs, the publicly observed running time is the same. It seems appealing to leverage the success of abstract interpretation [5, 6]. Abstract interpretation tools enjoy rigorous guarantees and provide formal proofs of various safety and liveness properties. They also are efficient. Implementations such as ASTRÉE [12, 13] are able to validate properties of large C programs.

Abstract interpretation-based techniques focus on single executions, but to prove 2-safety properties (or indeed, k-safety, for any k) we need to relate multiple executions. A clever way to do this is to employ self-composition [14, 15]: To reason about k runs of a program, we can concatenate k copies of it (with variables suitably renamed) and then assert a property that relates variables in different copies. For our timing channel property, we could concatenate the program with itself, require that public inputs to both copies be the same, and then assert that execution counters inserted for each copy are (approximately) equal at the conclusion, despite allowed variation of secret inputs. Self-composition can be expensive due to the explosion of the cross-product state space. As such, several works have looked to improve the basic idea by exploiting structural similarities in the program (e.g., using *interleaving composition* [16, 17, 18]). Other works applied similar ideas to improve relational verifiers directly [19, 20, 21, 22]. These approaches (and others) nevertheless rely, at least implicitly, on making k copies of the program, which means that invariants are split across the product program. The result can be either poor performance or key information being lost due to abstraction during fixpoint computation.

With a focus on verifying the absence of timing channels, we depart from the composition-based strategies and instead establish a novel *decomposition* methodology. Our key insight is that rather than prove a relational property about all pairs of execution traces, we can prove a non-relational property about each trace in a certain trace partition, computed iteratively.

We implemented our approach in a tool called *Blazer*. It works by equipping a standard abstract interpreter with the ability to consult an oracle to decide which Control-Flow Graph (CFG) arcs to follow, thus deriving partition-specific invariants. *Blazer* initially considers a partition with all paths in the program. If *Blazer* proves that all partition elements T_i have tight bounds on running times, the property is proved. If not, *Blazer* further sub-partitions T_i into smaller groups of paths, where the difference in paths between the partitions is not dependent on a secret's value (as determined by taint analysis). It then iterates until either the property is proved or no further subpartition is possible. In the latter case, *Blazer* attempts to synthesize possible attacks. In particular, it generates sub-partitions and running times based on secret information (i.e., is tainted); if a difference in secret values results in observable differences in running time, then there is a possible attack.

We evaluated *Blazer* on a collection of 25 benchmarks, including 12 tricky hand-crafted benchmarks, 7 programs from the literature [1, 2, 3], and 6 fragments of STAC challenge problems [23]. We find that *Blazer* is able to prove the absence of timing channels when the program is safe or else synthesize an attack specification in all but two cases.

Full details of the formalism, algorithm, and evaluation can be found in the published paper [4]. In the remainder of this section we present our approach (§4.1.1), experimental results (§4.1.2), and the challenges we had with using off-the-shelf taint analysis with *Blazer* (§4.1.3).

4.1.1 Approach. We will use simple examples to present the key idea of using a partition of executions to prove timing channel freedom. We then describe our algorithm in more detail, explaining how partitioning is interleaved with running time computation. Finally, we explain how the same basic approach can be used to synthesize a possible attack when timing channel freedom cannot be proved.

Consider the following example program.

Example 1.

```
1   void foo(int high, uint low) {
2     if (high == 0) {
3       i = 0;
4       while(i < low) i++;
5     }
6     else {
7       i = low;
8       while(i > 0) i−−;
9     }
10  }
```

This program takes a secret input high, and an attacker-controlled (tainted) input, low. A program is said to have a *timing channel* if the attacker, assumed to know the program

code, can infer information about the high by observing the program's running time (perhaps iteratively varying the input low). The execution of this particular program branches on secret variable high so we may wonder whether there is a timing channel. Proving there is not one requires, in principle, that we relate *all pairs of execution traces*. Doing so directly (*e.g.* by constructing a self-composition [14, 16] or a relational program analysis [24, 25, 19, 21, 22]) can magnify the overall state space to consider.

We show that we can instead prove that all executions (later: execution partitions) *share the same property*. For timing channels, we want to prove that each execution of the above program has a running time that is a function of (only) low. In this case we can be more specific: the running time is *linear* in low (for some fixed linear function). We might write this as a property $P^{\text{lin}}_{\mathsf{low}}$, and then write:

$$\forall \pi \in [\![C]\!].\ P^{\text{lin}}_{\mathsf{low}}(\pi)$$

where $[\![C]\!]$ is the set of all execution traces of the program C. An obvious consequence of this is that *every pair* of executions π_1, π_2 share $P^{\text{lin}}_{\mathsf{low}}$. As such, it is clear that there can be no timing channel.

The key idea of our approach is to break down the executions of the program into various cases, depending on high-independent branching, and in each case discover a running time property P that describes all traces in that case. To do this, we symbolically (and automatically) discover a partitioning of the execution traces $\mathfrak{T} = T_1, ..., T_n$ such that $[\![C]\!] = \bigcup_{i\in[1,n]} T_i$ so that we can find some P_i to characterize the running time for all $\pi \in T_i$. As long as each P_i is independently acceptable (*i.e.*, running time does not depend (much) on high), then the overall program satisfies the desired property. In Example 1, we only needed one partition component. Here is another example.

Example 2.

```
1   void bar(int high, int low) {
2     int i;
3     if (low > 0) { // O(2*low)
4       i = 0;
5       while(i<low) i++;
6       while(i>0) i−−;
7     } else { // O(1)
8       if (high == 0) { i = 5; } else { i = 0; i++; }
9     }
10  }
```

In this program, not all executions have the same symbolic running time. If low is positive, the execution will be linear in low, otherwise it will be either one instruction or two, depending on the value of high. To discover this, we can form a partition of the execution traces as follows:

$$T_{>} \triangleq \{\pi \mid in(\pi)[\mathsf{low}] > 0\} \quad \text{and} \quad T_{\leqslant} \triangleq \{\pi \mid in(\pi)[\mathsf{low}] \leqslant 0\}$$

where $[\![C]\!] \subseteq T_{>} \cup T_{\leqslant}$ and $in(\pi)[\mathsf{low}]$ means the value of the low input variable of trace π. For now, let us assume this partitionis given to us (we describe how we get the partitionshortly).

With some help (discussed below), we can coerce an off-the-shelf abstract interpreter to now perform two analyses proving, respectively, that:

$$\forall \pi \in T_{>}.\ P^{\text{lin}}_{\mathsf{low}}(\pi) \quad \text{and} \quad \forall \pi \in T_{\leqslant}.\ P^{\text{const}}_{\text{c}}(\pi).$$

Here, property $P^{\text{const}}_{\text{c}}(\pi)$ means that the running time of the trace is within some (fixed) small attacker-unobservable bound (say, ε) of the constant c. For this example, $c = 1$ with $\varepsilon = 1$.

These two proofs establish relationships between any two *copartitional* traces (two traces in $T_{>}$ or two traces in $T_{\leqslant}$). But what about some $\pi_0 \in T_{>}$ and some $\pi_1 \in T_{\leqslant}$? These two traces have different symbolic running times. However, notice that the path condition (whether low is above 0) depends only on variable low and not on secret variable high. Consequently, we can immediately conclude that, although traces from two different partition components have different running times, these differences cannot be correlated with high's value.

Synthesizing Partitions with Trails. Our trace partition in Example 2 considered different input values: those for which $\mathsf{low} > 0$, or not. This distinction corresponds to whether we branch at line 3, or not. Our algorithm likewise develops a partition by considering sets of paths, particularly those that take a branch one way vs. the other. Paths are specified using annotated regular expressions tr that we call *trails*, which for the purposes of this example have the following grammar:

$$tr ::= \ ij \ \parallel \ i{\downarrow} \ \parallel \ tr_1 \cdot tr_2 \ \parallel \ tr_1 \mid_\alpha tr_2 \ \parallel \ tr^*_\alpha$$

Trails are defined over edges ij in a control-flow graph, where i and j represent CFG blocks in the program. We also permit edge $i{\downarrow}$, meaning a block i that subsequently moves to the exit block. We use $\cdot$ for sequential composition (often dropped when clear from context), vertical bar for branching, and star for looping. Both branching and looping regex operators are annotated with $\alpha \in \{l,h,(l,h)\}$. Symbol l is used to mean low, h to mean high, or l,h to mean both. Here is an example:[1]

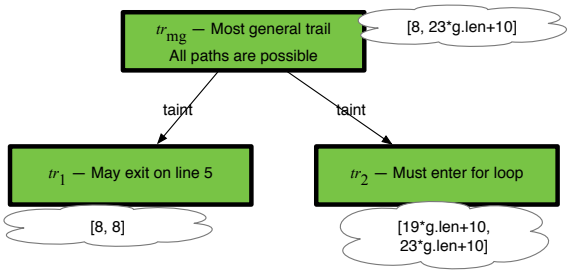
$$23 \cdot (34 \cdot 45 \cdot 5^*_l \cdots) \mid_l (38 \cdots)$$

This represents executions of Example 2. These executions start on line 2 and proceed to line 3, characterized by edge 23. Then they follow the branch at line 3, which depends on low input, as indicated by the annotation l. According to the left-hand side of the branch, this trail considers executions that go from line 3 to 4 and then 4 to 5 and then 0 or more times through line 5, due to the low-dependent loop, etc. According to the right-hand side of the branch $|_l$, the trail also considers executions that go from 3 to 8 (etc.).

The trail-based partition is generated iteratively. We start with a trail that characterizes all possible executions of the program; we call this the *most general trail*, tr_{mg}. We attempt to compute the running time of all paths characterized by this trail. If the running time is primarily a function of low input, with constant-bounded effect from high, then we are done. If this is not possible, we further break down the trail at low-dependent (only) branching

[1] For this example we refer to line numbers, rather than CFG blocks (as used by our actual algorithm), for clarity.

```
1   loginSafe(String username, byte[] guess) {
2     boolean dummy, matches = true;
3     byte[] user_pw = retrievePassword(username);
4     if(user_pw == null)
5       return false;
6     for(int i = 0; i < guess.length; i++) {
7       if (i < user_pw.length) {
8         if(guess[i] != user_pw[i]) matches = false;
9         else dummy = true;
10      } else {
11        dummy = true; matches = false;
12      }
13    }
14    return matches; }
```




```
1   loginBad(String username, byte[] guess) {
2     byte[] user_pw = retrievePassword(
            username);
3     if(user_pw == null)
4       return false;
5     for(int i = 0; i < guess.length; i++) {
6       if (i < user_pw.length) {
7         if(guess[i] != user_pw[i])
8           return false;
9       } else {
10        return false;
11      }
12    }
13    return true; }
```

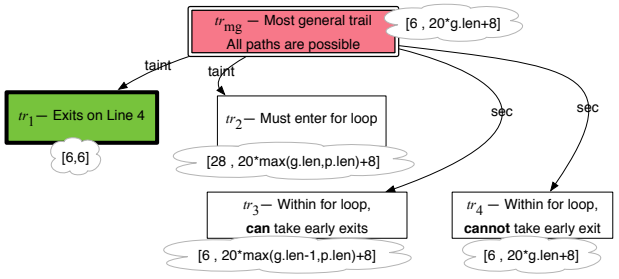


Figure 2: Two Versions of a Program That Validates a Password (Left Column) and the Corresponding Output of our Tool *Blazer* (Right Column). Each Rectangle Represents a Trail, and Each Downarrow from a Trail Represents a Subtrail; the taint and sec Annotations Indicate on What Sort of Data the Subtrail Was Created; the Balloons Contain Ranges $[x, y]$ that Indicate the Lower/Upper Bound on the Trail's Running Time in Terms of Number of Instructions (g.len is Short for guess.length and p.len is Short for user_pw.length); Bolded (Green) Nodes Indicate Areas Where the Program is Safe and Double-Line (Red) Nodes Indicate an Attack Specification

points. For Example 2, this results in a trail describing all paths for all true paths through line 3, and another for all false paths.

Ultimately, the final partition is a collection of trails $tr_1, ..., tr_n$ such that:

1. The union of all of the trails' languages covers the language of tr_{mg}. That is, $\bigcup_{i \in [1,n]} L(tr_i) \supseteq L(tr_{mg})$.
2. Trails correlate to the branching decisions that depend only on low security variables.
3. Each trail tr_i is such that every execution's running time can be described by a single function P_i, *e.g.* $P_{\mathsf{low}}^{\mathrm{lin}}$. Hence it can be proved that $\forall \pi \in L(tr_i) \cap [\![C]\!]$, that $\pi \models P_i$.

Thanks to the formal guarantees of our decomposition this suffices to entail that the overall program is free of timing channels.

More elaborate example. Consider the program loginSafe at the top of Figure 2. This function looks up the given username (a non-secret) and if the user is known, checks that the given guess matches the user's password. We return true if so, and false otherwise.[2]

To the right of the code is a visual depiction of the a tree of trail specifications. The top box represents the most general trail, which is our starting point. With it, we compute the running time of executions that follow this trail. This is embodied in a component called BoundAnalysis. Technically, to implement BoundAnalysis we equip an off-the-self abstract interpreter with the ability to be restricted to a given trail, leverage the seeding technique [26] to compute transition invariants [27], and match these invariants against a database of complexity bound lemmas [28, 29]. The result of applying BoundAnalysis to $\not{tr}_{mg}$ is that it computes a lower bound of 8 and an upper bound of $23 \times \mathsf{g.len} + 10$ (depicted as a cloud), where g.len is shorthand for guess.length.

With the range of running times in hand, we now ask whether the range is *narrow*: that the running time of the given executions is a function of low variables plus up to a maximum constant value c, where c is a limit on the observability of the attacker, *i.e.*, the difference between the longest and shortest running time is c. If we find that the range between a given lower/upper bound is narrow then we know we mark the code as free from timing channels. For $\not{tr}_{mg}$, we find that this bound is *not* narrow. This means we need to partition $\not{tr}_{mg}$ according to (only) tainted data, and then try again, for each partition.

We do this using a component called RefinePartition. In this case it splits the most general trail into two based on the branch at line 4. The parent-child relationship in the tree shown in the figure tracks this subtrail relationship. Edges are annotated as to whether the subtrail was chosen based on branching on low data (when trying to prove safety) or based on branching on high data (when trying to synthesize an attack, discussed shortly). For legibility, we have replaced each trail's regular expression with an intuitive description of the trail it specifies.

Now we compute the running times for these two trails separately. The left-hand trail has running time 8—this is the trail that exits on line 4. The right side has a running time that is a direct function of g.len. These two running times individually satisfy our narrowness criterion for timing channel freedom. In particular, the constant one does trivially, and the range on the right side does assuming that $\mathsf{g.len} \leqslant n$ for some fixed size that does not exceed the power of the attacker (which can be specified); e.g., if $n = 100$, the running time is equivalent to $21 \times \mathsf{g.len} \pm 210$, which is safe assuming the adversary's observational power is bounded by the constant 210. As such, because each one is acceptable, the two together are as well, and the program is sure to be free of timing channels.

Synthesizing Attacks. Our partitioning strategy is also useful for finding possible attacks if a program is not free of timing channels. To see how this works, consider the bottom of Figure 10. For this example, there is a timing channel (based on a bug in the Tenex password checker [30])—if the program exits on either lines 8 or 10, then the running time reveals the length of the prefix of guess that matches the real password.

When trying to prove this program is timing-channel free, the bound analysis will discover

[2]We do not consider the presence or non-presence of a username in the database to be secret information, for this example.

running time bounds similar to the correct example: a lower bound of 6 and an upper bound of $20 \times \mathsf{g.len} + 8$. As such, it will partition again according to the tainted branch on line 3. This time, though, the running times of the two partition trails do not meet our narrowness criterion: while the trail on the left hand side is narrow (due to the exit on line 4), the trail on the right hand side is not narrow: it has a constant lower bound and an upper bound that relies on both g.len and the length of the actual password. Unfortunately, we cannot easily sub-partition the right side further, since branches in the loop also depend on secret information—partitioning is only permitted on low data. As such we have found a potential problem.

At this point, the tool changes gears and attempts to discover a vulnerability. To this end, RefinePartition generates the next two trails (tr_3 and tr_4). These trails differ based on whether or not the shortcut **return** statement can be taken on Line 7 (or 10). For both tr_3 and tr_4, our toolchain computes a lower bound of constant running time, corresponding to the early exit on line 4. For tr_4, our toolchain computes a *linear* running time of $20 \times \mathsf{g.len} + 8$ for the upper bound. Meanwhile, tr_3 forces the program to return early after entering the loop, either by taking the **return** statement on Line 8 or the **return** statement on Line 10. For tr_3, our toolchain computes a range of running times, different from the ones computed for tr_4. With this, the tool reports a vulnerability: there are two trails (tr_3 and tr_4), the choice between them depends on high data (arcs labeled sec), and yet they have different running times. Therefore, for the same low input there can be two different possible executions that yield different running times as the branching depends on the value of the secret.

We call this output an *attack specification*. Because we are working with a static analysis, the result of our tool is not immediately two concrete traces. However, it provides a specification for two traces that witness the attack. All that remains is to ensure that these traces are feasible by finding justifying inputs. This can be done manually by a programmer or via an under-approximate analysis (*e.g.*, a symbolic execution [3]).

4.1.2 Evaluation. We implemented our decomposition-based approach in a tool called *Blazer* and it on 24 benchmarks, including 6 examples drawn from the STAC challenge problems, and 6 real-world programs in which timing attacks were exploited and reported in cryptography papers [1, 2, 3]. Benchmarks are paired up so that there are two versions: the "unsafe" version is expected to be vulnerable to timing-channel attacks while the "safe" one is not. For third-party benchmarks, we created safe versions by hand (except for User). Our experiment harness executes *Blazer* on both the safe and unsafe versions of each benchmark.

Benchmarks. Our benchmarks, which reflect a broad range of code patterns, are up to 100 basic blocks in size (details reported later in this section). Figure 3 illustrates some selected examples.

- *MicroBench.* These are hand-crafted to exercise the various aspects of *Blazer*. We start with simple examples; `nosecret_safe` tests the basics of side-channel detection, which can only occur when there is a secret. The others are more intricate. The `loopAndBranch` benchmark, for instance, is shown in Figure 3. At first this seems to have a vulnerability, but the potentially vulnerable trail is infeasible, which is caught by the abstract interpreter. Also shown in Figure 3 is the classic Unix login vulnerability

```
1 void loopAndbranch_safe(int high, int low) {
2   int i = high;
3   if(low < 0) { while(i > 0) i−−; }
4   else {
5     low = low + 10; // low is above 0
6     if(low >= 10) { int j = high; while(j>0) j−−; }
7     else { if(high<0) { int k = high; while (k>0) k
          −−; } }
8   } }
```

```
1 boolean login_safe(String u, String p) {
2   boolean outcome = false;
3   if (map.containsKey(u)) {
4     if (map.get(u).equals(md5(p)))
5       outcome = true;
6   } else {
7     if (map.get(u).equals(md5(p))) { } // remove
          for unsafe
8   }
9   return outcome; }
```

```
1 BigInteger modPow1_safe(BigInteger base,
      BigInteger exponent, BigInteger modulus) {
2   BigInteger s = BigInteger.valueOf(1);
3   int width = exponent.bitLength();
4   for (int i = 0; i < width; i++) {
5     s = s.multiply(s).mod(modulus);
6     if (exponent.testBit(width − i − 1))
7       s = s.multiply(base).mod(modulus);
8     else s.multiply(base).mod(modulus); //
          remove for unsafe
9   }
10  return s; }
```

Figure 3: Some Examples Selected from the Benchmarks. For Lack of Space, only the Main Methods are Shown

that leaks usernames. When line 7 is removed, the program takes longer when a username exists because it hashes the input password via `md5`.

- *STAC.* Several benchmarks were extracted from the STAC challenge problems. `modPow1` (shown in Figure 3) and `modPow2` perform cryptographic arithmetic using the Java `BigInteger` library.
- *Literature.* We have also crafted examples taken from papers that demonstrate timing attacks on real-world cryptographic methods. These include Genkin *et al.* [1] (*GPT*), Kocher [2] (*K96*), and Pasareanu *et al.* [3] (*PPM16*). The example from *PPM16* is discussed in Section 4.3.1 and shown in Figure 10.

Blazer supports manually-specified summaries of running times so we specify running times for library calls such as those to the Java `BigInteger` library (in `modPow#` and `Cryptography` benchmarks).

Observer modeling. As discussed in the implementation section, observable running time differences are modeled in several ways. We designed the *MicroBench* so that distinguishing between safe and unsafe is possible by evaluating computational complexity, *e.g.*, linear vs. quadratic. The variables are assumed to be unbounded, while a safe program is assumed to be one where the symbolic running times have the same polynomial degree. While sufficient for these hand-crafted micro-benchmarks, this model of observability is too simplistic for real-world code.

For the real-world examples from *STAC* and *Literature*, we use a model of observable running time based on concrete differences in bytecode instructions between partitions. We assume some reasonable maximum for the input variables, *e.g.*, 4096 bits for the cryptographic benchmarks. Then we plug these values into the symbolic bound expressions to get a concrete estimate of the maximum number of bytecode instructions. Using this method, an observable difference is defined by some minimum threshold in the difference between the number of instructions. For these benchmarks, we use a low number of instructions (25,000) to define the observable difference in running time. In real-world applications of this verification, observability depends on many factors, including hardware, operating system, network latency, etc, and would need to have application-specific calibration.

Results. The benchmarks were run sequentially on a single commodity Personal Computer with a quad-core 3.07 gigahertz processor and 12 gigabytes of Random Access Memory (RAM). Running time is collected for both safety verification alone as well as safety verification plus the search for an attack specification. The latter running time only applies to the unsafe benchmark, since the tool halts if it proves safety. Running time is measured by performing five runs and taking the median.

Table 1 shows the results. The **Benchmark** column identifies the benchmark's method and alternates between the safe and unsafe versions. **Size** indicates the number of basic blocks in the method's control-flow graph. The **Safety Time** column shows the tool's median running time in seconds for safety verification alone, while the **w/Attack Time** column is the median running for safety verification and the subsequent search for an attack specification.

Our tool is sound: it either determines the program is safe, finds an attack specification, or gives up. For every safe benchmark, *Blazer* verified the safety of the benchmark. In all unsafe benchmarks, the tool found an attack specification, i.e., two candidate subtrails with differing running times, except for `gpt14_unsafe`.

For most benchmarks, the safety verification takes only a few seconds, save some notable outliers. The running time for generating an attack specification, which includes the safety verification, often takes longer, because it is a computationally more intensive step. It takes the trails tree output from safety verification and further decomposes it into subtrails.

As for the outliers, the running time appears loosely related to the number of basic blocks in the program, as shown by the very high running times of the outliers `modPow1_unsafe`,

Table 1: The Results of Applying Our Tool *Blazer* to a Variety of Benchmarks: Hand-Made Examples, Examples from the Literature [1, 2, 3], and Extracted from STAC Challenge Problems. Median Running Times Are Shown for Safety Verification Alone as Well as Safety Verification Plus the Search for an Attack Specification. The Safe Benchmarks Need no Search for Attack Specification, so No Running Time is Shown. (Size = number of basic blocks in CFG, s = seconds)

Benchmark	Size	Safety Time (s)	w/Attack Time (s)
MicroBench			
array_safe	16	1.60	–
array_unsafe	14	0.16	0.70
loopBranch_safe	15	0.23	–
loopBranch_unsafe	15	0.65	1.54
nosecret_safe	7	0.35	–
notaint_unsafe	9	0.28	1.77
sanity_safe	10	0.63	–
sanity_unsafe	9	0.30	0.58
straightline_safe	7	0.21	–
straightline_unsafe	7	22.20	28.49
unixlogin_safe	16	0.86	–
unixlogin_unsafe	11	0.77	1.27
STAC			
modPow1_safe	18	1.47	–
modPow1_unsafe	58	218.54	464.52
modPow2_safe	20	1.62	–
modPow2_unsafe	106	7813.68	31758.92
pwdEqual_safe	16	2.70	–
pwdEqual_unsafe	15	1.30	2.90
Literature			
gpt14_safe	15	1.43	–
gpt14_unsafe	26	219.30	1554.64
k96_safe	17	0.70	–
k96_unsafe	15	1.29	3.14
login_safe	18	6.54	–
login_unsafe	17	4.40	9.10

`modPow2_unsafe`, and `gpt14_unsafe`. This is due to a combinatorial explosion of subtrails, superlinear with respect to the number of conditional branches, as well as the increased memory pressure of storing and processing the tree of decomposed subtrails. The running time for `straightline_unsafe` is an exception to this relationship: it has few basic blocks but a long running time. This is likely due to a particularly large basic block that has 90 instructions, increasing the processing time of the subtrails that contain it.

4.1.3 Experience report: Taint Analysis. Taint analysis (aka information flow analysis) is an important component of SOUCIS analysis, especially in the Blazer approach just described. In particular, taint analysis enables tracking of the flow of secret values through a program to focus on the vulnerable parts of the program that interact with secrets. Doing so makes the subsequent analysis task smaller, and more tractable. In the course of the project, we tried three different off-the-shelf taint analysis tools, JOANA [31], Pidgin [32], and Soot info-flow [33]. Despite the strongly encouraging published results, we found that no tool could consistently handle the size and complexity of target programs we considered. Worse, when tools did produce an answer, we found they could differ on the answer produced. Our experience leads us to believe that Java-based taint analysis is far from a solved problem and more serious work is needed.

The JOANA tool was an easy choice, because it operates on T. J. Watson Library for Analysis (WALA), the same underlying analysis framework that on which our own analysis tools, including the abstract interpreter and *Blazer*, operate. After working with the first engagement's challenge problems, we found that it could not efficiently produce results for STAC challenge programs that use 3rd party libraries. We profiled the taint analysis tool and found that the reason for this lack of scalability is that JOANA cannot generate system dependence graphs for these programs. The system dependence graph is the first step to tracking information flow through the programs. Without it, we would not be able to track the flow of secret values, a necessity for our static analysis approach.

We then experimented with the Pidgin information flow tool, which was capable of generating dependence graphs more quickly. Pidgin uses a multi-threaded pointer analysis engine, which significantly outperformed the pointer analysis used by JOANA. We enhanced Pidgin's dependence graph construction to help reduce the number of false positives on challenge programs, but by the third set of challenge programs, Pidgin's weaknesses were showing. Pidgin could produce a program dependence graph for most of the problems, but many were incomplete and queries on them did not correctly track the flow of the secret. Ultimately, only we were only able to get correct results on one out of the eight groups of challenge programs. This translated to only five of the 25 challenge questions.

Due to the limitations of Pidgin, we started exploring Soot infoflow, a taint analysis tool with a strong track record of analyzing Android Java programs with FlowDroid [34]. We also discovered that a team also working on STAC had used Soot infoflow and reported good results with Soot on two of the STAC challenge problems. This tool fared better than the previous tools, but still had trouble on many of the challenge problems.

This experience suggests that a revolutionary step forward is needed to improve taint analysis to the point that it can work for large programs.

4.2 Design Tradeoffs in Numeric Static Analysis for Java

Static analysis of numeric program properties has a broad range of useful applications. Such analyses can potentially detect array bounds errors [35], analyze a program's resource usage [29, 28], detect side channels [36, 37], and discover vectors for denial of service attacks [38, 39]. One of the major approaches to numeric static analysis is abstract interpretation [5], in which program statements are evaluated over an abstract domain until a fixed point is reached. Indeed, the first paper on abstract interpretation [5] used numeric intervals as one example abstract domain, and many subsequent researchers have explored abstract interpretation-based numeric static analysis [40, 41, 42, 43, 44, 45].

Abstract interpretation is an important part of the SOUCIS approach. Blazer uses an abstract interpreter to obtain program numeric invariants (Section 4.1). The performance and precision of the abstract interpreter significantly affect its ability to effectively detect timing-channel attacks.

Despite the long history, applying abstract interpretation to real-world Java programs remains a challenge. Such programs are large, have many interacting methods, and make heavy use of heap-allocated objects. In considering how to build an analysis that aims to be sound but also precise, prior work has explored some of these challenges, but not all of them together. For example, several works have considered the impact of the choice of numeric domain (e.g., intervals vs. convex polyhedra) in trading off precision for performance but not considered other tradeoffs [42, 46]. Other works have considered how to integrate a numeric domain with analysis of the heap, but unsoundly model method calls [40] and/or focus on very precise properties that do not scale beyond small programs [41, 42]. Some scalability can be recovered by using programmer-specified pre- and post-conditions [43]. In all of these cases, there is a lack of consideration of the broader design space in which many implementation choices interact. Understanding the design tradeoffs is important to produce a practical abstract interpreter to be used for detecting timing-channel attacks.

To bridge this gap, we implemented and systematically evaluated a large design space of fully automated, abstract interpretation-based numeric static analyses for Java. Each analysis is identified by a choice of five configurable options—the numeric domain, the heap abstraction, the object representation, the interprocedural analysis order, and the level of context sensitivity. In total, we study 162 analysis configurations to assess both how individual configuration options perform overall and to study interactions between different options. To our knowledge, our basic analysis is one of the few fully automated numeric static analyses for Java, and we do not know of any prior work that has studied such a large static analysis design space.

Our tool is publicly available at `https://github.com/plum-umd/JANA`. See [7] for more details of our approach and evaluation.

4.2.1 Analysis Configurations options. We selected analysis configuration options that are well-known in the static analysis literature and that are key choices in designing a Java static analysis. Table 2 summarizes the key choices we study.

For the numeric domain, we considered both intervals (INT) [47] and convex polyhedra (POL) [6], as these are popular and bookend the precision/performance spectrum.

Modeling the flow of data through the heap requires handling pointers and aliasing. We

Table 2: Analysis Configuration Options, and their Possible Settings

Config. Option	Setting	Description
Numeric domain (ND)	INT	Intervals
	POL	Polyhedra
Heap abstraction (HA)	SO	Only summary objects
	AP	Only access paths
	AP+SO	Both access paths and summary objects
Abstract object representation (OR)	ALLO	Alloc-site abstraction
	CLAS	Class-based abstraction
	SMUS	Alloc-site except Strings
Inter-procedural analysis order (AO)	TD	Top-down
	BU	Bottom-up
	TD+BU	Hybrid top-down and bottom-up
Context sensitivity (CS)	CI	Context-insensitive
	1CFA	1-CFA
	1TYP	Type-sensitive

consider three different choices of *heap abstraction*: using *summary objects* (SO) [40, 48], which are *weakly updated*, to summarize multiple heap locations; *access paths* (AP) [49, 50], which are *strongly updated*; and a combination of the two (AP+SO).

To implement these abstractions, we use an ahead-of-time, global *points-to analysis* [51], which maps static/local variables and heap-allocated fields to abstract objects. We explore three variants of *abstract object representation*: the standard *allocation-site abstraction* (the most precise) in which each syntactic `new` in the program represents an abstract object (ALLO); *class-based abstraction* (the least precise) in which each class represents all instances of that class (CLAS); and a *smushed string abstraction* (intermediate precision) which is the same as allocation-site abstraction except strings are modeled using a class-based abstraction [52] (SMUS).

We compare three choices in the *interprocedural analysis order* we use to model method calls: *top-down analysis* (TD), which starts with `main` and analyzes callees as they are encountered; and *bottom-up analysis* (BU), which starts at the leaves of the call tree and instantiates method summaries at call sites; and a hybrid analysis that is bottom-up for library methods and top-down for application code (TD+BU). In general, top-down analysis explores fewer methods, but it may analyze callees multiple times. Bottom-up analysis explores each method once but needs to create summaries, which can be expensive.

Finally, we compare three kinds of *context-sensitivity* in the points-to analysis: *context-insensitive* analysis (CI), *1-CFA analysis* [53] in which one level of calling context is used to discriminate pointers (1CFA), and *type-sensitive analysis* [54] in which the type of the receiver is the context (1YP).

4.2.2 Evaluation. We implemented our analysis using WALA [55] for its intermediate representation and points-to analyses and either APRON numerical abstract domain library [56, 57] or ETH Library for Numerical Analysis (ELINA) [58, 59] for the interval or polyhedral, respectively, numeric domain. We then applied all 162 analysis configurations to the DaCapo benchmark suite [60] (see Table 4), using the numeric analysis to try to prove array accesses are within bounds. We measured the analyses' performance and the number

of array bounds checks they discharged. We analyzed our results by using a multiple linear regression over analysis features and outcomes, and by performing data visualizations.

Since many analysis configurations are time-intensive, we set a limit of 1 hour for running a benchmark under a particular configuration. All performance results reported are the median of the three runs. We also use the median precision result, though note the analyses are deterministic, so the precision does not vary except in the case of timeouts. Thus, we treat an analysis as not timing out as long as either two or three of the three runs completed, and otherwise it is a timeout. Among the 1782 median results (11 benchmarks, 162 configurations), 667 of them (37%) timed out. The percentage of the configurations that timed out analyzing a program ranged from 0% (xalan) to 90% (chart).

We studied three research questions.

Research Question (RQ) 1: we examined how analysis configuration affects performance. Table 3 summarizes our regression model for performance. We measure performance as the time to run both the core analysis and perform array index out-of-bounds checking. If a configuration timed out while analyzing a program, we set its running time as one hour, the time limit (characterizing a lower bound on the configuration's performance impact). Another option would have been to leave the configuration out of the regression, but doing so would underrepresent the important negative contribution to performance.

In the top part of the table, the first column shows the independent variables and the second column shows a setting. One of the settings, identified by dashes in the remaining columns, is the baseline in the regression. We use the following settings as baselines: TD, AP+SO, 1TYP, ALLO, and POL. We chose the baseline according to what we expected to be the most precise settings. For the other settings, the third column shows the estimated effect of that setting with all other settings (including the choice of program, each an independent variable) held fixed. For example, the fifth row of the table shows that AP (only) decreases overall analysis time by 37.6 minutes compared to AP+SO (and the other baseline settings). The fourth column shows the 95% confidence interval around the estimate (the performance difference of an analysis option to the baseline is contained in the interval between the two times provided). The last column shows the p-value: the probability that, when the null hypothesis is true, the statistical summary is the same or of greater magnitude than the observed results. As is standard, we consider p-values less than 0.05 (5%) significant; such rows are highlighted green. The bottom part of the table shows the additional effects of two-way combinations of options compared to the baseline effects of each option. Any interactions not included were deemed not to have meaningful effect and thus were dropped by the model generation process [61].

Table 3 presents several interesting performance trends. We found that using summary objects causes significant slowdowns, e.g., the vast majority of the analysis runs that timed out used summary objects. We also found that polyhedral analysis incurs a significant slowdown, but only half as much as summary objects. Surprisingly, bottom-up analysis provided little performance advantage generally, though it did provide some benefit for particular object representations. Finally, context-insensitive analysis is faster than context-sensitive analysis, as might be expected, but the difference is not great when combined with more approximate (class-based and smushed string) abstract object representations.

RQ2: we examined how analysis configuration affects precision. We found that using access paths is critical to precision. We also found that the bottom-up analysis has worse

Table 3: Model of Run-Time Performance in Terms of Analysis Configuration Options (Table 2), Including Two-Way Interactions. Independent Variables for Individual Programs not Shown. R^2 of 0.72. (Explained or total variation – how close data are to the fitted regression line) [Est. (min) = estimated effect in minutes, CI = confidence interval (units in minutes)].

Option	Setting	Est. (min)	CI	p-value
AO	TD	-	-	-
	BU	-1.98	[-6.3, 1.76]	0.336
	TD+BU	1.97	[-1.78, 6.87]	0.364
HA	AP+SO	-	-	-
	AP	-37.6	[-42.36, -32.84]	<0.001
	SO	0.15	[-4.60, 4.91]	0.949
CS	1TYP	-	-	-
	CI	-7.09	[-10.89, -3.28]	<0.001
	1CFA	1.62	[-2.19, 5.42]	0.405
OR	ALLO	-	-	-
	CLAS	-11.00	[-15.44, -6.56]	<0.001
	SMUS	-7.15	[-11.59, -2.70]	0.002
ND	POL	-	-	-
	INT	-16.51	[-19.56, -13.46]	<0.001
AO:HA	TD:AP+SO	-	-	-
	BU:AP	-5.31	[-9.35, -1.27]	0.01
	TD+BU:AP	-3.13	[-7.38, 1.12]	0.15
	BU:SO	0.11	[-3.92, 4.15]	0.956
	TD+BU:SO	-0.08	[-4.33, 4.17]	0.97
AO:OR	TD:ALLO	-	-	-
	BU:CLAS	-8.87	[-12.91, -4.83]	<0.001
	BU:SMUS	-4.23	[-8.27, -0.19]	0.04
	TD+BU:CLAS	-4.07	[-8.32, 0.19]	0.06
	TD+BU:SMUS	-2.52	[-6.77, 1.74]	0.247
AO:ND	TD:POL	-	-	-
	BU:INT	8.04	[4.73, 11.33]	<0.001
	TD+BU:INT	2.35	[-1.12, 5.82]	0.185
HA:CS	AP+SO:1TYP	-	-	-
	AP:1CFA	7.01	[2.83, 11.17]	<0.001
	AP:CI	3.38	[-0.79, 7.54]	0.112
	SO:CI	-0.20	[-4.37, 3.96]	0.924
	SO:1CFA	-0.21	[-4.37, 3.95]	0.921
HA:OR	AP+SO:ALLO	-	-	-
	AP:CLAS	9.55	[5.37, 13.71]	<0.001
	AP:SMUS	6.25	[2.08, 10.42]	<0.001
	SO:SMUS	0.07	[-4.09, 4.24]	0.973
	SO:CLAS	-0.43	[-4.59, 3.73]	0.839
HA:ND	AP+SO:POL	-	-	-
	AP:INT	6.94	[3.53, 10.34]	<0.001
	SO:INT	0.08	[-3.32, 3.48]	0.964
CS:OR	1TYP:ALLO	-	-	-
	CI:CLAS	4.76	[0.59, 8.93]	0.025
	CI:SMUS	4.02	[-0.15, 8.18]	0.05
	1CFA:CLAS	-3.09	[-7.25, 1.08]	0.147
	1CFA:SMUS	-0.52	[-4.68, 3.64]	0.807

precision than top-down analysis, especially when using summary objects, and that using a more precise abstract object representation improves precision. But other traditional ways of improving precision do so only slightly (the polyhedral domain) or not significantly (context-sensitivity).

RQ3: we looked at the precision/performance tradeoff for all programs. To begin our discussion, Table 4 shows the fastest configuration and the most precise configuration for

Table 4: Benchmarks and Overall Results (Prog = program, min = minutes).

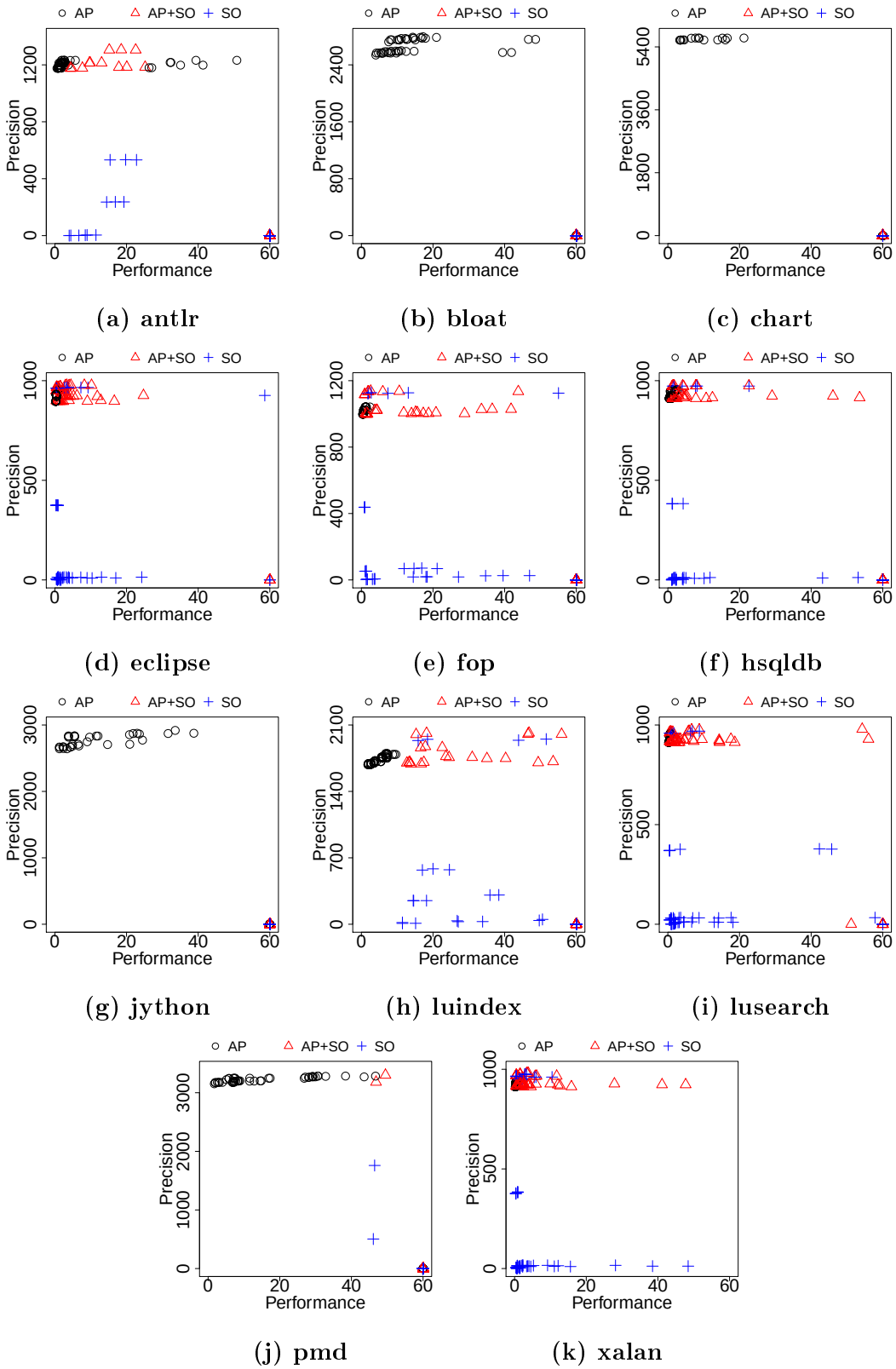
Prog	Size	# Checks	Best Performance			Best Precision		
			Time(min)	# Checks	Percent	Time(min)	# Checks	Percent
			BU-AP-CI-CLAS-INT			TD-AP+SO-1TYP-CLAS-INT		
antlr	55734	1526	0.6	1176	77.1%	18.5	1306	85.6%
			BU-AP-CI-CLAS-INT			TD-AP-1TYP-SMUS-POL		
bloat	150197	4621	4.0	2538	54.9%	17.2	2795	60.5%
			BU-AP-CI-CLAS-INT			TD-AP-1TYP-SMUS-INT		
chart	167621	7965	3.3	5593	70.2%	7.7	5654	71.0%
			BU-AP-CI-ALLO-INT			TD-AP+SO-1TYP-SMUS-POL		
eclipse	18938	1043	0.2	896	85.9%	3.3	977	93.7%
			BU-AP-CI-CLAS-INT			TD-AP+SO-1CFA-SMUS-INT		
fop	33243	1337	0.4	998	74.6%	2.6	1137	85.0%
			BU-AP-CI-SMUS-INT			TD-AP+SO-CI-SMUS-INT		
hsqldb	19497	1020	0.3	911	89.3%	1.4	975	95.6%
			BU-AP-CI-SMUS-INT			TD-AP-1CFA-CLAS-POL		
jython	127661	4232	1.3	2667	63.0%	33.6	2919	69.0%
			BU-AP-CI-SMUS-INT			TD-AP+SO-1TYP-ALLO-INT		
luindex	69027	2764	1.8	1682	60.9%	46.8	2015	72.9%
			BU-AP-CI-CLAS-INT			TD-AP+SO-1CFA-ALLO-POL		
lusearch	20242	1062	0.2	912	85.9%	54.2	979	92.2%
			BU-AP-CI-CLAS-INT			TD-AP+SO-CI-CLAS-INT		
pmd	116422	4402	1.7	3153	71.6%	49.5	3301	75.0%
			BU-AP-CI-CLAS-INT			TD-AP+SO-1CFA-SMUS-POL		
xalan	20315	1043	0.2	912	87.4%	3.8	981	94.1%

each benchmark. Further, the table shows the configurations' running time, number of checks discharged, and percentage of checks discharged.

We see several interesting patterns in this table, though note the table shows just two data points and not the full distribution. First, the configurations in each column are remarkably consistent. The fastest configurations are all of the form BU-AP-CI-*-INT, only varying in the abstract object representation. The most precise configurations are more variable, but all include TD and some form of AP. The rest of the options differ somewhat, with different forms of precision benefiting different benchmarks. Finally, notice that, overall, the fastest configurations are much faster than the most precise configurations—often by an order of magnitude—but they are not that much less precise—typically by 5–10 percentage points.

To delve further into the tradeoff, we examine, for each program, the overall performance and precision distribution for the analysis configurations, focusing on particular options (HA, AO, etc.). As settings of option HA have come up prominently in our discussion so far, we start with it and then move through the other options. Figure 4 gives per-benchmark scatter plots of this data. Each plotted point corresponds to one configuration, with its performance on the x-axis and number of discharged array bounds checks on the y-axis. We regard a configuration that times out as discharging no checks, so it is plotted at (60, 0). The shape of a point indicates the HA setting of the corresponding configuration: black circle for AP, red triangle for AP+SO, and blue cross for SO. As a general trend, we see that *access paths improve precision and do little to harm performance; they should always be enabled.* More specifically, configurations using AP and AP+SO (when they do not time out) are always toward the top of the graph, meaning good precision. Moreover, the performance profile of SO and AP+SO is quite similar, as evidenced by related clusters in the graphs differing in the y-axis, but not the x-axis. In only one case did AP+SO time out when SO alone did not.[3]

[3]In particular, for eclipse, configuration TD+BU-SO-1CFA-ALLO-POL finished at 59 minutes, while



(a) antlr (b) bloat (c) chart

(d) eclipse (e) fop (f) hsqldb

(g) jython (h) luindex (i) lusearch

(j) pmd (k) xalan

Figure 4: Tradeoffs: **AP** vs. **SO** vs. **AP+SO**

In addition to the above observation, we found top-down analysis works better than bottom-up. While summary objects, originally proposed by Fu [40], do help precision for some programs, the benefits are often marginal when considered as a percentage of all checks, so they tend not to outweigh their large performance disadvantage. Lastly, we found that the precision gains for more precise object representations and polyhedra are modest, and performance costs can be magnified by other analysis features.

4.2.3 Conclusion and Future Work. Systematically implementing and evaluating a novel family of techniques to handle method calls, heap-allocated objects, and numeric analysis led to some interesting conclusions. Among others, we discovered that strongly updatable access paths are always a good idea, adding significant precision at very little performance cost. We also found that top-down analysis also tended to improve precision at little cost, compared to bottom-up analysis. On the other hand, while summary objects did add precision when combined with access paths, they also added significant performance overhead, often resulting in timeouts. The polyhedral numeric domain improved precision, but would time out when using a richer heap abstraction; intervals and a richer heap would work better.

The results of our study suggest several directions for future work that may improve the capability of SOUCIS approaches. A systematic study of *Blazer* (Section 4.1) using statistical analysis and data visualization will help discover better design tradeoffs, leading to better performance and precision in detecting timing-channel attacks. Another direction is to investigate more scalable and precise techniques to further improve the tool, e.g., a more sparse representation of summary objects that retains their modest precision benefits, but avoids the overall blowup. Finally, the results of our Java numeric analysis may be used to detect other important security bugs, e.g., algorithmic complexity vulnerabilities.

TD+BU-AP+SO-1CFA-ALLO-POL timed out.

4.3 Bound Analysis Using Dynamic Invariants

The automated discovery of *program invariants*—relations among variables that are guaranteed to hold at certain locations of a program—is an important research area in program analysis and verification. Generated invariants can be used to prove correctness assertions, reason about resource usage, establish security properties, provide formal documentation, and more [62, 63, 64, 65, 66, 67]. In the context of the SOUCIS approach, invariants are needed by *Blazer*, described in Section 4.1, in order to compute running times as part of the decomposition-based static analysis.

A particularly useful class of invariants, especially in the context of using them for inferring running times, are *numerical invariants*, which involve relations among numerical program variables. Within this class of invariants, *nonlinear polynomial* relations, e.g., $x \leqslant y^2, x = qy + r$, arise in many scientific, engineering, and safety- and security-critical applications.[4] For example, the commercial analyzer Astrée, which has been applied to verify the absence of errors in the Airbus A340/A380 avionic systems [13, 12], implements the ellipsoid abstract domain [68] to represent and analyze a class of quadratic inequality invariants. Complexity analysis, which aims to determine a program's performance characteristics [28, 69, 70], perhaps to identify possible security problems [71, 4], also makes use of polynomial invariants, e.g., $O(n^2 + 2m)$ where n, m are some program inputs. In addition, such polynomial invariants have been found useful in the analysis of hybrid systems [72, 73], and in fact are required for implementations of common mathematical functions such as mult, div, square, sqrt and mod.

A static analysis can reason about all program paths soundly, but doing so is often expensive and is only possible for relatively simple forms of invariants [74]. For *Blazer* in particular, we found our abstract interpretation-based approach had difficulty producing precise invariants at scale. Dynamic analyses limit their attention to only some of a program's paths, and as a result can often be more efficient and produce more expressive invariants, but provide no guarantee that those invariants are correct [67, 75]. Recently, several systems (such as the Precondition Inference Engine (PIE) [76], the Implication Counter-Examples (ICE) learning model [77] and *Guess-and-Check* [78]) have been developed that take a hybrid approach: use a dynamic analysis to infer *candidate invariants* but then confirm these invariants are correct for all inputs using a *static verifier*. When invariants are incorrect the verifier returns counterexample traces which the dynamic inference engine can use to infer more accurate invariants. This iterative process is called *CounterExample Guided Invariant geneRation* (CEGIR).

While the CEGIR approach is promising, existing tools have some practical limitations. One limitation is that they find invariants strong enough to prove a particular (programmer-provided) postcondition where the quality of the generated invariants depends on the strength of the postcondition. As such, they are not well suited for automated analyses on code that lacks such formal specifications. Another limitation is that these tools employ a *sound* static verifier, which aims to definitively prove that an invariant holds. While this is a good goal, it turns out to be a significant restriction on the quality of the invariants that can ultimately be inferred—it can be quite challenging to do when invariants are nonlinear polynomials and involve many program variables.

[4] We refer to nonlinear polynomial relations such as $x = qy + r, x \leqslant y^2$ simply as *polynomial relations*.

We developed a new CEGIR algorithm called NumInv that overcomes these limitations. It has two main components. First, it uses algorithms from Dynamic Invariant Generator (DIG) [79, 75] to dynamically infer expressive polynomial equality invariants and linear inequality relations from traces at specified program locations. Second, it uses KLEE [80], a symbolic executor, to check candidate invariants and produce counterexamples when they fail to hold. To check that an invariant p holds at location L, NumInv transforms the input program so that L is guarded by the conditional $\neg p$. If KLEE is able to reach L then p must not be an invariant, and so it outputs a counterexample consisting of the relevant input values at that location. On the other hand, if KLEE never reaches that location prior to timing out, then NumInv accepts the invariant as correct. Although this technique is unsound, KLEE, by its nature as a symbolic executor, turns out to be very effective in discovering counterexamples to refute invalid candidates.

We evaluated NumInv by using it to infer invariants on more than 90 benchmark programs taken from the Nonlinear Arithmetic (NLA) [79] and Hoare Logic with Abduction (HOLA) [81] suites for program verification and from examples in the literature on complexity bound analysis [28, 69, 82]. Our results show that NumInv generates sufficiently strong invariants to verify correctness and to understand the semantics of 23/27 NLA programs containing nontrivial arithmetic and polynomial relations. We also find that NumInv discovers highly precise invariants describing nontrivial complexity bounds for 18/19 programs used to benchmark static complexity analysis techniques (in fact, for 4 programs, NumInv obtains more informative bounds than what were given in the literature). We note that both ICE and PIE cannot find any of these invariants produced by NumInv, even when we explicitly tell these tools that they should attempt to verify these invariants. Finally, on the 46 HOLA programs, we compare NumInv directly with PIE. We find it performs competitively: in 36/46 cases its inferred invariants match PIE's, are stronger, or are more descriptive.

Thus, although NumInv can potentially return unsound invariants, our experience shows that it is practical and effective in removing invalid candidates and in handling difficult programs with complex invariants. We believe that NumInv strikes a practical balance between correctness and expressive power, allowing it to discover complex, yet interesting and useful invariants out of the reach of the current state of the art.

Full details of the approach and experimental results are present in the published paper [8]. In the remainder of this section we outline NumInv's approach in more detail, summarize key results focusing on running time inference, and sketch extensions we have worked on since the published paper.

4.3.1 Approach. NumInv generates invariants using the technique of *counterexample-guided invariant generation* (CEGIR). At a high level, CEGIR consists of two components: a *dynamic analysis* that infers candidate invariants from execution traces, and a *static verifier* to check candidates against the program code. If a candidate invariant is spurious, the verifier also provides counterexamples (*cex*s). Traces from these cexs are recycled to repeat the process, hopefully producing accurate results. These steps of inferring and checking repeat until no new cexs or (true) invariants are found. The CEGIR approach is basically exploiting the observation that inferring a sound solution directly is often harder than checking a (cheaply generated) candidate solution.

Other promising CEGIR algorithms, e.g., the ICE, PIE and *Guess-and-Check* tools, have been developed in recent years that take the same approach [77, 76, 78], though they refer to it differently. These approaches have been able to prove correctness of specifications by inferring inductive loop invariants, or sufficient and necessary preconditions. Some of these works (ICE and PIE) are verification oriented, i.e. they infer invariants to specifically prove a given assertion. In this approach, the computation of these "helper" invariants strictly depends on the given assertions, e.g., if the intended assertion is True then the inferred invariant can be just True.

NumInv has different goals and takes a different approach. Our goals are both discovery and verification, and our approach is to find the strongest possible invariant at any arbitrarily given location. When given an undocumented program, NumInv can discover interesting properties and provide formal specifications. For example, NumInv can reveal a stronger postcondition than the user might think to write down, and the user doesn't have to write down any postconditions at all. Moreover, when given a specific assertion, the resulting invariant from NumInv can help prove it (e.g., if the invariant matches or is stronger than the assertion). Empirically, NumInv can frequently infer invariants that are at least as strong as the postcondition, and frequently, stronger.

NumInv. NumInv infers candidate invariants using the algorithms from DIG [79, 75], which produce equality and inequality relations from traces. To check invariants, NumInv invokes KLEE [80], a symbolic executor that is able to synthesize test cases for failing tests.

KLEE as a "verifier". NumInv generates candidate invariants at program locations L of interest (e.g., at the start of loops or at the end of functions). To check whether a property p holds at a location L, NumInv asks KLEE to determine the reachability of the location L when guarded by $\neg p$. For example, to check whether the relation $x = qy + r$ is an invariant at some location L, NumInv modifies the program as follows

```
...
if (!(x==qy+r)){
  [L]
  save(x,y,q,r); //cex traces
  abort();
}
...
```

KLEE then runs this program, systematically exploring the space of possible inputs. If, during this process, location L is reached, then the relation does not hold, so a cex consisting of the values of the relevant input variables is saved for subsequent inference. On the other hand, KLEE may be able to explore all program paths and thus verify that indeed that invariant p holds. Or, if this is infeasible, NumInv terminates KLEE after some timeout.

The use of KLEE as the verifier is a key feature of NumInv. Because programs often contain a very large number of possible paths, KLEE rarely explores all of them. However, in our experience if it does not quickly find a counterexample for p then p very likely holds. This is true even when p is a nonlinear polynomial relation. As such, KLEE serves as

a practical improvement over existing theorem provers and constraint solvers, for which reasoning over general polynomial arithmetic is a significant challenge.

Inferring polynomial equalities and linear inequalities. NumInv uses two CEGIR algorithms to find candidate numerical relations p at program locations of interest. The first algorithm finds *polynomial equalities*. To do this, for each program location L, NumInv produces a *template* equation $c_1t_1 + c_2t_2 \cdots c_nt_n = 0$. This equation contains n unknown coefficients c_i and n *terms* t_i, with one term for each possible combination of relevant program variables, up to some degree d. NumInv calls KLEE on the program to systematically obtain many possible valuations of relevant variables at L. Each distinct observed valuation, which we call a *trace*, is substituted into the template to form an instantiated equation. After obtaining at least n traces, NumInv solves the c_i using the resulting set of equations. Substituting the solutions back into the template, we can extract candidate invariants. At this point, NumInv enters a CEGIR loop that tests the candidate invariants by using KLEE as described above. Any spurious invariants are dropped, and the corresponding cex traces are used to infer new candidates, as described above, until no additional true invariants are found.

NumInv's second algorithm tries to infer *linear inequalities* in the form of *octagons*, which are inequalities over two variables, containing eight edges. It refines the bounds on the candidate invariants using a divide-and-conquer algorithm. Once again, NumInv estimates and obtains an initial set of traces. It enumerates all possible octagonal inequality forms involving one and two variables and uses KLEE to check inequalities under these forms are within certain ranges $[minV, maxV]$. It then narrows this range, iteratively seeking tighter lower and upper bounds.

Finally, from the obtained equality and inequality invariants, NumInv removes any invariants that are logical implications of other invariants. For instance, we suppress the invariant $x^2 = y^2$ if another invariant $x = y$ is also found because the latter implies the former. We check possible implications using an SMT solver (checking whether the negation of the implication is unsatisfiable).

4.3.2 Results.

Computational Complexity. We used NumInv to discover invariants from a variety of interesting programs; details are present in our published paper. Here, we focus on NumInv's ability to discover invariants capturing a program's computational complexity, e.g., $O(n^3)$ where n is some input. Figure 5 shows the program triple with three nested loops, adapted from the program in Figure 2 of Gulwani *et al.* [28]. The complexity of this program, i.e., the total number of iterations of all three loops at location L, appears to be $O(nmN)$ at first glance. Additional analysis yields a more precise bound of $O(n + mn + N)$ because the number of iterations of the innermost loop is bounded by N instead of nmN and it furthermore directly affects the running time of the outermost loop [28].

When given this program, NumInv discovers an interesting and unexpected postcondition at location L about the counter variable t, which is a ghost variable introduced to count loop

```
void triple(int n, int m, int N){
  assert (0 <= n && 0 <= m && 0 <= N);
  int i = 0, j = 0, k = 0; int t = 0;
  while(i < n){//loop 1
    j = 0; t++;
    while(j<m){//loop 2
      j++; k=i; t++;
      while (k<N){k++; t++;}// loop 3
      i=k;
    }
    i++;
  }
  [L]
}
```

Figure 5: An Example Program That Has Muliple Polynomial Complexity Bounds

iterations:

$$N^2mt + Nm^2t - Nmnt - m^2nt - Nmt^2 + mnt^2 + Nmt \\ -Nnt - 2mnt + Nt^2 + mt^2 + nt^2 - t^2 - nt + t^2 = 0.$$

At first glance, this quartic (degree 4) equality with 15 terms looks incomprehensible and quite different than the expected bound $O(n + mn + N)$ or even $O(mnN)$. However, solving this equation for t, i.e., finding the roots, yields three solutions $t = 0$, $t = N + m + 1$, and $t = n - m(N - n)$. Careful analysis reveals that these results actually describe three distinct and exact bounds of this program:

$$\begin{array}{lll} t = 0 & \text{when} & n = 0, \\ t = N + m + 1 & \text{when} & n \leqslant N, \\ t = n - m(N - n) & \text{when} & n > N. \end{array}$$

Thus, NumInv can find numerical invariants that represent precise program complexity. More importantly, the obtained relations can describe expressive and nontrivial *disjunctive* invariants, which capture different possible complexity bounds of a program. As can be seen in [8], NumInv produced very promising results that capture the precise complexity bounds for the benchmarks programs.

Examples from Engagements. NumInv helped with attacking some of the canonical examples distributed by STAC white team. In Figure 6, the method `canonical()` is presented with integer inputs `guess`, `secret`, `n` and `t`. Depending on whether `guess` is less than or equal to `secret`, or not, there are two different cases, exhibiting two different sets of possible running times, this time depending on the value of `t`. In particular, if `guess <= secret`, then the running time of the method is $O(1)$, $O(n)$ or $O(n^3)$, respectively when `t` $= 0$, `t` $= 1$ or otherwise. On the other hand, if `guess` $>$ `secret`, the running time of the method is

$O(1)$, $O(n^2)$ or $O(n^3)$, respectively when `t` $= 0$, `t` $= 1$ or otherwise. Irrespective of the values of `guess` and `secret`, the lower bound of the complexity of the method is $O(1)$ and the upper bound is $O(n^3)$. Nevertheless, the program is still insecure, as to whether `guess` is less than or equal to `secret` can be observed, through its running time, when the value of `t` is 2. An analysis that searches only for the upper and lower bounds of the program, would not be able to classify it as insecure.

We ran NumInv on this method, using two different assertions at the beginning of the method, to artificially restrict the execution to the two cases of whether `guess` <= `secret` or `guess` > `secret`. For the two distinct cases, NumInv returned the invariant:

$$n^4 \cdot p^2 - n^3 \cdot p^3 - n^4 \cdot p + n^3 \cdot p^2 - n \cdot p^3 + p^4 + n \cdot p^2 - p^2 = 0.$$

This equation is equivalent to

$$p \cdot (p - n^3) \cdot (p - n) \cdot (p - 1) = 0,$$

which gives the 3 distinct cases:

$$\begin{aligned} p &= 1 \quad \text{when} \quad t = 1, \\ p &= n \quad \text{when} \quad t = 2, \\ p &= n^3 \quad \text{when} \quad t \neq 2 \text{ and } t \neq 1, \end{aligned}$$

with $p = 0$ corresponding to the cases where the assertion is false. Changing the assertion to `guess>secret`, we similarly obtain the result that $p = 1$ when $t = 1$, $p = n^2$ when $t = 2$, and $p = n^3$ otherwise. Comparing these two sets of possible complexities, we observe that whether `guess` > `secret` is observable, and thus deduce that this program is insecure to timing attacks.

4.3.3 Extensions. Since our published paper, a few more extensions have been applied to this work. The first is generalizing the equation solving part of the algorithm to instead use regression methods for approximating, instead the polynomial equation that best captures the relationship between the input variables and the program resources. As a result, the methodology works with more general programs, and allowing more noisy observations as well as probabilistic behavior. As a very simple example, consider the program in Figure 7. The number of loop iterations (`t`) of this program, is $\frac{n}{2}$ when `n` is even, and $\frac{(n+1)}{2}$ when `n` is odd, since the increment for `i` at each iteration is `a` which is equal to 2. It is not easy to capture this relationship with an equation of degree 1 over variables a, t and n. An equation of degree 2 that could work is $(t - \frac{n}{2}) \cdot (t - \frac{(n-1)}{2}) = 0$. Then, the degree of such an equation depends on the value of `a`, making it infeasible to generate equations that capture the complexity of the program for large values of `a`.

The approach we follow instead, is to generate enough traces, and then perform linear regression over the same terms considered in NumInv. For the program such as the one above, the relationship that a regression method would output is $t = \frac{n}{2}$, and more generally $t = \frac{n}{a}$ for `a` being a variable. The approach can also work on programs that involve random variables, and in many cases data structures, such as arrays.

```
void canonical(int guess, int t, int n, int secret) {
    int p = 0;
    int i = 0;
    assert(guess <= secret);
    if(guess <= secret){
        if(t == 1){
            p += 1;
        } else if(t == 2){
            for(i = 0; i<n;i++){
                p += 1;
            }
        } else{
            for(i = 0; i<n*n*n;i++){
                p += 1;
            }
        }
    } else {
        if(t == 1){
            p += 1;
        } else if(t == 2){
            for(i = 0; i<n*n;i++){
                p += 1;
            }
        } else{
            for(i = 0; i<n*n*n;i++){
                p += 1;
            }
        }
    }
    //%%%traces: int n, int p
}
```

Figure 6: An Example Program That Has Muliple Polynomial Complexity Bounds, and Which Bounds Are Different Depending on Whether `guess <= secret`

Secondly, we have extended the tool to apply the above procedure iteratively in order to find upper and lower bounds. Informally, the tool generates traces to obtain an equational relationship between the variables and the program resources using the regression method described above, and then filters the traces to keep those where the resource variable has a value larger (respectively smaller) than the one described by the relation, converging in this way to the upper (respectively lower) bound of the complexity of the program. For example, for the simple program in Figure 8, the procedure quickly returns the upper bound n^2. This approach has also been successful on examples that operate on data structures such as arrays, managing to find the upper bounds, for example, for various sorting algorithms, without being aware of the semantics of the program. As the approach tries to dynamically

```
void loopMod2(int n){
  assert (0 <= n);
  int i = 0, t = 0;
  int a = 2;
  while(i < n){
    t++;
    i+=a;
  }
}
```

Figure 7: An Example Program with Approximate Running Time $\frac{n}{a}$

```
void nsq(int n){
  assert (0 <= n);
  if (nondet()) {
    for (int i = 0; i < n; ++i) {
      t++;
    }
  } else {
    for (int i = 0; i < n*n; ++i) {
      t++;
    }
  }
}
```

Figure 8: An Example Program That Non-Deterministically Has Running Time Between n and n^2

find inputs that would cause the worst case complexity, it naturally works on smaller inputs better than on larger ones.

4.4 Fuzzing for Complexity Attacks and Side-Channel Attacks

In the previous three sections we have focused on the *Sound, Overapproximate* analysis component of SOUCIS. In particular, *Blazer*, Java Numeric Analysis (JANA), and NumInv all aim to draw conclusions that are true for all program runs. On the other hand, such kinds of analysis have difficulty scaling to large programs, or else have difficulty producing sufficiently precise results. NumInv makes use of dynamic analysis to get around some of these problems, but we could even go one step further, leaning even more heavily on *underapproximate* analysis approaches. In this section, we consider work we did to apply *fuzz testing* (or simply, *fuzzing*) to discover complexity attacks (and some initial forays into discovering side channels).

Fuzzing is a bug-finding methodology that has good traction and success in the software development ecosystem. Fuzzing involves automatically and efficiently generating and executing test cases for a program while monitoring the program for failure. There are many different parameters and variables for fuzzing: how test cases are generated, what exactly is monitored, and how failure is defined. In general, developers consider crashes as failures, and fuzzing has been very effective at finding single inputs that can crash applications, especially applications that perform file based I/O. Fuzzers like libfuzzer [9] and American Fuzzy Lop (AFL) [83] are very effective in this space. Later, we present a general framework that describes these fuzzers and their variations.

Existing fuzzers work well when programs fit into an existing fuzzers methodology. Fuzzers today do well in a few specific scenarios: the program does file I/O, and the goal of the analysis to find crashes. What about programs that don't perform file I/O, such as web servers? Or if the goal of the analysis is to find timing-based side-channels? Then the utility of existing fuzzers and fuzzing algorithms is much less clear.

This section starts by reviewing the state of the art in fuzz testing. Next, it discusses our own fuzzer design, and key features that it needed to support STAC programs, notably I/O mocking, memory checkpointing, multi-run (k) fuzzing. Finally, we conclude with a summary of results obtained to date, and sketch plans for future extensions involving integration with static analysis.

4.4.1 Fuzzing in review. There are many different dynamic analyses that can be described as "fuzzing." A unifying feature of fuzzers is that they operate on, and produce, concrete inputs. Otherwise, fuzzers might be instantiated with many different design choices and many different parameter settings. In this section, we outline the basics of how fuzzers work, and then touch on the advances of 32 recently published papers.

Most modern fuzzers follow the procedure outlined in Figure 9. The process begins by choosing a corpus of "seed" inputs with which to test the target program. The fuzzer then repeatedly mutates these inputs and evaluates the program under test. If the result produces "interesting" behavior, the fuzzer keeps the mutated input for future use and records what was observed. Eventually the fuzzer stops, either due to reaching a particular goal (e.g., finding a certain sort of bug) or reaching a timeout.

Different fuzzers record different observations when running the program under test. In a "black box" fuzzer, a single observation is made: whether the program crashed. In "gray box" fuzzing, observations also consist of intermediate information about the execution, for ex-

Core fuzzing algorithm:

```
corpus ← initSeedCorpus()
queue ←
observations ← ∅
while ¬isDone(observations,queue) do
    candidate ← choose(queue, observations)
    mutated ← mutate(candidate,observations)
    observation ← eval(mutated)
    if isInteresting(observation,observations) then
        queue ← queue ∪ mutated
        observations ← observations ∪ observation
    end if
end while
```

parameterized by functions:

- **initSeedCorpus**: Initialize a new seed corpus.
- **isDone**: Determine if the fuzzing should stop or not based on progress toward a goal, or a timeout.
- **choose**: Choose at least one candidate seed from the queue for mutation.
- **mutate**: From at least one seed and any observations made about the program so far, produce a new candidate seed.
- **eval**: Evaluate a seed on the program to produce an observation.
- **isInteresting**: Determine if the observations produced from an evaluation on a mutated seed indicate that the input should be preserved or not.

Figure 9: Fuzzing, in a Nutshell

ample, the branches taken during execution as determined by pairs of basic block identifiers executed directly in sequence. "White box" fuzzers can make observations and modifications by exploiting the semantics of application source (or binary) code, possibly involving sophisticated reasoning. Gathering additional observations adds overhead. Different fuzzers make different choices, hoping to trade higher overhead for better bug-finding effectiveness.

Usually, the ultimate goal of a fuzzer is to generate an input that causes the program to crash. In some fuzzer configurations, *isDone* checks the queue to see if there have been any crashes, and if there have been, it breaks the loop. Other fuzzer configurations seek to collect as many different crashes as they can, and so will not stop after the first crash. For example, by default, `libfuzzer` [9] will stop when it discovers a crash, while AFL will continue and attempt to discover different crashes. Other types of observations are also desirable, such as longer running times that could indicate the presence of algorithmic complexity vulnerabilities [84]. In any of these cases, the output from the fuzzer is some

concrete input(s) and configurations that can be used from outside of the fuzzer to reproduce the observation. This allows software developers to confirm, reproduce, and debug issues.

Fuzzing is an active area of research, and many works have been published in the last 5 or 6 years that improve upon this basic algorithm. We consider each part of the algorithm, next, and describe papers that aim to advance it.

initSeedCorpus. Skyfire [85] and Orthrus [86] propose to improve the initial seed selection by running an up-front analysis on the program to bootstrap information both for creating the corpus and assisting the mutators. QuickFuzz [87, 88] allows seed generation through the use of grammars that specify the structure of valid, or interesting, inputs. DIFUZE performs an up-front static analysis to identify the structure of inputs to device drivers prior to fuzzing [89].

mutate. SYMFUZZ [90] uses a symbolic executor to determine the number of bits of a seed to *mutate*. Several other works change *mutate* to be aware of taint-level observations about the program behavior, specifically mutating inputs that are used by the program [91, 92, 93, 94]. Where other fuzzers use pre-defined data mutation strategies like bit flipping or rand replacement, MutaGen uses fragments of the program under test that parse or manipulate the input as mutators through dynamic slicing [95]. Scheduled Document Object Model (DOM) Fuzzer (SDF) uses properties of the seeds themselves to guide mutation [96]. Sometimes, a grammar is used to guide mutation [97, 98]. Chizpurfle's [99] mutator exploits knowledge of Java-level language constructs to assist in-process fuzzing of Android system services.

eval. Driller [100] and MAYHEM [92] observe that some conditional guards in the program are difficult to satisfy via brute force guessing, and so (occasionally) invoke a symbolic executor during the *eval* phase to get past them. Semi-Symbolic Fuzz Testing (S2F) also makes use of a symbolic executor during *eval* [101]. Other work focuses on increasing the speed of *eval* by making changes to the operating system [102] or using different low level primitives to observe the effect of executions [103, 104, 98]. T-Fuzz [105] will transform the program to remove checks on the input that prevent new code from being reached. The MEDS [106] memory error detector, like AddressSanitizer (ASAN), performs run time analysis to detect errors during fuzzing.

isInteresting. While most papers focus on the crashes, some work changes *observation* to consider different classes of program behavior as interesting, e.g., longer running time [84], or differential behavior [107]. Steelix [93] and Angora [94] instrument the program so that finer grained information about progress towards satisfying a condition is exposed through *observation*. Dowser and VUzzer [108, 91] uses a static analysis to assign different rewards to program points based on either a likely-hood estimation that traveling through that point will result in a vulnerability, or for reaching a deeper point in the CFG.

choose. Several works select the next input candidate based on whether it reaches particular areas of the program [91, 109, 110, 111]. Other work explores different algorithms for selecting candidate seeds [112, 113].

4.4.2 A Java Fuzzer for Side Channel and Complexity Attacks.

As part of SOUCIS we have been building a fuzzer for Java programs. It works by using the Java ASM bytecode rewriting framework to carry out program rewriting in the style of AFL and libfuzzer, along

with fuzzing algorithms that match AFL and libfuzzer. The architecture of our fuzzer is similar to those in the state of the art (per Figure 9), but includes some important extensions. It most resembles libfuzzer in that it requires a user to write a *harness* which invokes the target program's method(s) to fuzz; this harness is repeatedly called by the main fuzzing loop with new, random input.

Our fuzzer contains a number of novel features. First of all, we have worked to make it support programs that are interactive, i.e., servers or user-facing programs, rather just batch programs that process files. We do this by supporting what we call *I/O mocking*. Second, we have worked to support fuzzing *stateful* programs, i.e., those whose internal state changes with each interaction. The standard assumption of existing fuzzers is that the state of the target program is not affected from one test to another. To address this issue, we integrate *memory checkpointing* as part of our fuzzer, so that we can quickly roll back state changes as a result of a particular test. Finally, we have generalized the fuzzing loop to support different classes of input, i.e., secret values vs. public ones. Doing so allows us to, for example, fix public values while repeatedly generating new secret values. This support is necessary for finding side channels.

I/O mocking For some applications, it is obvious how a fuzzer would concretely execute the application with input, such as a program that takes input from a file or `stdin`. However, many applications do not fit neatly into this structure, such as server software or cloud applications. These applications take input from a network socket or even more general constructs like web server frameworks, where input is provided through specific Application Programming Interfaces (APIs) or an application framework that abstracts the source of the I/O. Existing fuzzing methodologies, both in fuzzers that we have created as well as libfuzzer, support these types of applications by having developers and analysts write harnesses that connect the fuzzer-generated inputs to the methods in the application that can be fuzzed. However, writing these harnesses is a tedious manual process, and errors in the harnesses could result in false positives.

We identify locations in the program we are testing where I/O methods are used, and use program re-writing to replace calls to those methods with calls to methods internal to the fuzzer. Then, instead of creating a specific harness that chooses an entry point in the program to start fuzzing from, we start the program from main and allow it to run normally. In specific instances, we could start the program from an intermediate state if such a state is identifiable, for example, running a program until it processes input, then saving program state at that point and reverting to it for each test case. As the fuzzed program reads data, instead of invoking library I/O routines, it invokes our mocked I/O routines, and so data generated by the fuzzer is immediately returned.

Memory checkpointing. One concern with in-process fuzzing is that methods invoked by the fuzzer might make global changes to memory, and that sequences of these global changes to memory might lead the program program into a state that would be impossible outside of the fuzzing environment. This would produce false positives. To mitigate this, we integrated a memory checkpointing system, **CROCHET** [114], into our Java fuzzer. Doing so allowed us to snapshot the state before each fuzzing iteration and roll back changes made

```
1  private byte[] secret;
2  public boolean safeCheck(byte[] input) {
3    int r = true;
4    for (int i = 0;
5         i < min(secret.length, input.length);
6         i++)
7      if (i < secret.length && i < input.length)
8        r &= secret[i] == input[i];
9    return r;
10 }
11
12 public boolean unsafeCheck(byte[] input) {
13   for (int i = 0;
14        i < min(secret.length, input.length);
15        i++)
16     if (i < secret.length && i < input.length)
17       if( secret[i] != input[i])
18         return false;
19   return true;
20 }
```

Figure 10: Password Checking Programs, Safe and Unsafe Variants. Due to the Early Exit in the Unsafe Variant, an Attacker That Can Observe Timing Difference Between Evaluations with their own `input` can Coerce the Program into Leaking the Content of `secret`

after, starting fuzzing from a fresh context each time. Overhead from **CROCHET** seems low (3% to 5%), helping the overall use of it to scale.

***k*-Fuzzing.** Current fuzzers, like AFL and libfuzzer, consider programs that take a single input. They generate new inputs and run programs on those inputs, looking for violations of safety properties. What about programs and safety properties that are parameterized around 2-safety, such as timing-based side-channels? For a motivating example, consider the password checking program in figure 10.

Instead of generating one input, we would like our fuzzer to generate pairs of inputs, (P, S) to provide as the `input` and `secret` variables in the program, respectively. If our fuzzer generates two test cases, (P, S_1) and (P, S_2) where the Hamming distance is minimal between P and S_1 but maximal between P and S_2, then the observed running times should demonstrate the side-channel.

Extending fuzzing to multiple inputs fits into our previous algorithmic definition 9 by expanding the notion of a "seed" into a "test case" which could have k different inputs. One might represent a public channel, another would represent a secret channel, and so on. The particular concrete inputs in one seed are tied together to produce a single observation, but mutators are free to change any input in any way.

We have begun preliminary work to extend our existing fuzzer to carry out k-fuzzing.

In particular, we have organized the fuzzing loop to support this direction. However, we have yet to flesh out all of the remaining elements for it to work beyond toy examples. For example, mutation strategies should be augmented so that they generate variants of input (P, S) but also so that they vary P and S at different rates. This could be important, since without this control, the odds of producing a mutation that demonstrate a violation of safety seem low. For example, in the `unsafeCheck` method, if `secret` and `input` each change on every iteration, the fuzzer would learn less about how to change `secret` such that the running time changed.

4.4.3 Evaluation and Next Steps. We used our fuzzer as part of the STAC Engagement that ended in February 2018. We had some success using it in a targeted fashion. For the program `battleboats_1`, we set up the fuzzer harness to repeatedly call suspicious, stateless function. In the harness we created inputs for each of the function arguments, based on random data provided from the fuzzer main loop. We set the scoring function for the fuzzer to prioritize both coverage (us usual) and running time, with the result that the fuzzer found a vulnerable input.

In `stegosaurus` we initiated a sessions object such that we could fuzz different request parameters. In particular we wanted to see if there was a side channel in space (while the problem mentioned time), so we fuzzed the key parameter for encrypting the image and changed scoring to be the size of the encrypted image. We also mocked the `imageIO` class so that we only needed to load a file once or could make the input be from the fuzzer. We found a large gap in the sizes between of an encrypted image based on the secret used to encrypt. This strongly pointed to the presence of a vulnerability.

Unfortunately, our fuzzer was not well-enough engineered to be of use for more programs. Only by the end of the take-home engagement did we have a more customizable testing tool for exploring particular suspect parts of the program. At the live engagement we found many more places where the fuzzer's easy customization for mocking and scoring would have been useful. With further engineering, we believe the approach will prove to be generally useful.

Further work. A key goal of SOUCIS is to combine overapproximating ("static") and underapproximating ("dynamic") analysis. We see this interaction with NumInv, and there is potential for it with our Java fuzzer too, by involving static analysis.

Static analysis can help fuzzing in three ways. Firstly, it can help with the process of automatically creating mock I/O interfaces and inserting them into the application. Secondly, it can guide the process of inserting instrumentation that will help the fuzzer better distinguish between interesting states. Finally, it can guide transformations of fuzzed code that allow the fuzzer to efficiently and precisely discern intermediate results about gatekeeping operations that the fuzzer has a low chance of guessing, in the style of Steelix [93] and laf-tintel [115]. We believe all of these directions constitute fruitful areas for subsequent exploration.

4.5 Evaluating Fuzz Testing

Fuzz testing seems to work well in practice, and this motivated us to explore using it for SOUCIS. However, fuzz testing's long list of practical successes only says that it works, not *why* it works. Since we are interested in applying fuzzing to a more sophisticated problem (side channel and complexity attacks), we are also interested in understanding the key elements that are the root of fuzz testing's success. That way, we can retain these elements in our tools, varying the rest. While identification of key ideas of fuzzing algorithms may, in principle, be drawn from mathematical analysis, fuzzers are primarily evaluated experimentally, in practice. As such, we examined prior, published experimental evaluations and assessed their results.

We examined 32 recently published papers on fuzz testing located by perusing top-conference proceedings and other quality venues, and studied their experimental evaluations. To find these papers, we started from 10 high-impact fuzzing papers published in top security venues. Then we chased citations to and from these papers. As a sanity check, we also did a keyword search of titles and abstracts of the papers published since 2012. Finally, we judged the relevance based on target domain and proposed advance, filtering papers that did not fit. Unfortunately, rather than yielding interesting scientifically valid insights, we found that every evaluation in these 32 papers had serious flaws in its methodology.

To confirm that such flaws would manifest as issues in practice, we carried out more than 50000 Central Processing Unit (CPU) hours of experiments, comparing AFLFast [111] with AFL as a baseline B. We chose AFLFast as it was a recent advance over the state of the art; its code was publicly available; and we were confident in our ability to rerun the experiments described by the authors in their own evaluation and expand these experiments by varying parameters that the original experimenters did not. We targeted three binutils programs (*nm*, *objdump*, and *cxxfilt*) and two image processing programs (*gif2png* and *FFmpeg*) used in prior fuzzing evaluations [112, 91, 90, 113, 101]. We found that experimental results revealed the potential for serious flaws if best practices are not followed. In particular,

Fuzzing performance under the same configuration can vary substantially from run to run. Thus, comparing only single runs, as $\frac{2}{3}$ of the examined papers seem to, does not give a full picture. For example, on *nm*, one AFL run found just over 1200 crashing inputs while one AFLFast run found around 800. Yet, comparing the median of 30 runs tells a different story: 400 crashes for AFL and closer to 1250 for AFLFast. Comparing averages is still not enough, though: We found that in some cases, via a statistical test, that an apparent difference in performance was not statistically significant.

Fuzzing performance can vary over the course of a run. This means that short timeouts (of less than 5 or 6 hours, as used by 11 papers) may paint a misleading picture. For example, when using the empty seed, AFL found no crashes in *gif2png* after 13 hours, while AFLFast had found nearly 40. But after 24 hours AFL had found 39 and AFLFast had found 52. When using a non-empty seed set, on *nm* AFLFast outperformed AFLFast at 6 hours, with statistical significance, but after 24 hours the trend reversed. We similarly found *substantial performance variations based on the seeds used*; e.g., with an empty seed AFLFast found more than 1000 crashes in *nm* but with a small valid seed it found only 24, which was statistically indistinguishable from the 23 found by AFL. And yet, most papers treated the choice of seed casually, apparently assuming that any valid seed would work equally well,

without providing particulars.

The most direct measure of fuzzing effectiveness is unique bugs found, and yet only about $\frac{1}{4}$ *of papers used this measure.* Most instead counted the number of crashing inputs found, and then applied a heuristic procedure in an attempt to de-duplicate inputs that trigger the same bug. The two most popular heuristics were AFL's coverage profile (used by 8 papers) and (fuzzy) stack hashes [116] (used by 7 papers). Unfortunately, there is reason to believe these *de-duplication heuristics are ineffective.* In a small experiment, we computed a portion of ground truth by applying all patches to *cxxfilt* from the version we fuzzed up until the present, grouping all crashing inputs that a particular patch addressed, i.e., how many now resulted in a graceful exit. We found that all 57142 crashing inputs deemed unique by coverage profiles were addressed by merely 13 patches, and confirmed that each patch represented a distinct conceptual bugfix. This represents a dramatic overcounting of the number of bugs. Ultimately, while AFLFast found many more "unique" crashing inputs than AFL, it only had a slightly higher likelihood of finding more bugs in a given run. Stack hashes did better, but still over-counted bugs. Instead of the bug mapping to, say 500 AFL coverage-unique crashes in a given trial, it would map to about twelve stack hashes, on average. Stack hashes were also subject to false negatives: roughly 1 in 13 hashes also mapped to a different crash not associated with the bug, which means that a bug could be ultimately missed. This experiment suggests that reliance on heuristics for evaluating performance is unwise. A better approach is to measure against ground truth directly by assessing fuzzers against known bugs, e.g., as we did above, or by using a synthetic suite such as Cyber Grand Challenge (CGC) [117] or Large-scale Automated Vulnerability Addition (LAVA) [118], as done by 6 papers we examined.

Overall, *fuzzing performance may vary with the target program.* In our experiments, we found that while AFLFast performed generally better than AFL on *binutils* programs (basically matching its originally published result, when using an empty seed), it did not provide a statistically significant advantage on the image processing programs. And yet, *few papers use a common, diverse benchmark suite*; about 6 used CGC or LAVA-M, and 2 discussed the methodology in collecting real-world programs, while the rest used a few handpicked programs, with little overlap in these choices (and no overlap when versions are considered) among papers. As a result, individual evaluations may present misleading conclusions internally, and results are hard to compare across papers.

Our study suggests that meaningful scientific progress on fuzzing requires that claims of algorithmic improvements be supported by more solid evidence. *Every evaluation in the 32 papers we looked at lacks some important aspect in this regard.* Fortunately, there are some simple guidelines that future papers can follow that will address the issues. In particular, researchers should perform multiple trials and use statistical tests; they should evaluate different seeds, and should consider longer ($\geqslant$ 24 hour vs. 5 hour) timeouts; and they should evaluate bug-finding performance against ground truth. We also identify some important questions still to be decided by the community. Notably, we argue for the establishment and adoption of a good fuzzing benchmark. The practice of hand selecting a few particular targets, and varying them from paper to paper, is problematic. A well-designed and agreed-upon benchmark would address this problem.

Detailed discussion of our results can be found in our published paper [10].

Table 5: Basic-Block Attributes

Type	Attribute name
Block-level attributes	String Constants
	Numeric Constants
	No. of Transfer Instructions
	No. of Calls
	No. of Instructions
	No. of Arithmetic Instructions
Inter-block attributes	No. of offspring
	Betweenness

4.6 Machine Learning for Vulnerability Detection

Another promising kind of underapproximating analysis is *machine learning*. Machine learning cannot be certain that it always draws sound conclusions, but machine learning algorithms have shown great promise in practice.

We developed a neural network approach for vulnerability detection. The basic idea is to learn a neural network model to detect whether two binary codes are similar or not. Using this functionality, given a new binary function, we can detect whether it is similar to any of the code samples with complexity or side channel vulnerabilities in our database. In the following, we will present the approach, the evaluation results, and issues to apply this technique to the STAC problem. More details and results can be found in our published paper [11].

4.6.1 Neural Network-based Code Similarity Detection. We first formally define the general code similarity detection problem, which can be task dependent. We assume there exists an *oracle* $\pi : \mathcal{F} \times \mathcal{F} \rightarrow \{-1, 1\}$ determining the code similarity metric for a given task, where $\mathcal{F}$ is the domain of all binary functions. This oracle is unknown, and we would like to learn it. Given two binary program functions f_1, f_2, $\pi(f_1, f_2) = 1$ indicates that they are similar; otherwise, $\pi(f_1, f_2) = -1$ indicates that they are dissimilar.

The objective of code similarity embedding problem is to find a mapping ϕ which maps a function f to a vector representation μ. Intuitively, such an embedding should capture enough information for detecting similar functions. That is, given an easy-to-compute similarity function $Sim(\cdot, \cdot)$, (e.g., cosine function of two vectors), and two binary functions f_1, f_2, i.e., $Sim(\phi(f_1), \phi(f_2))$ is large if $\pi(f_1, f_2) = +1$, and is small otherwise.

One advantage of learning the embedding (i.e., the mapping ϕ) is that it enables efficient computation. The similarity between two functions can be computed using an inexpensive similarity function between two vectors, without incurring the cost of expensive graph matching algorithms.

Neural Network-based Program Embedding Our basic idea is to first use a preprocessing step to represent a binary function into a directed graph, i.e., a control-flow graph, which captures the semantic information of the original binary function. In particular, we leverage the existing work [119] to use a control-flow graph representation to represent each

binary function. On each node, which corresponds to a basic block, we compute several attributes in Table 5, which have been used in the literature [119]. In doing so, each binary function can be converted into *Attributed Control Flow Graph (ACFG)* using a dissembler, i.e., IDA Pro [120].

Then, we develop a neural network to compute an embedding for an ACFG. To this aim, we leverage a general framework, called `Structure2vec` [121]. `Structure2vec` is inspired by graphical model inference algorithms where vertex-specific features x_v are aggregated recursively according to graph topology g. After a few steps of recursion, the network will produce a new feature representation (or embedding) for each vertex which takes into account both graph characteristics and long-range interaction between vertex features. More specifically, we denote $\mathbb{N}(v)$ as the set of neighbors of vertex v in graph g. Then one variant of the `Structure2vec` network will initialize the embedding $\mu_v^{(0)}$ at each vertex as 0, and update the embeddings at each iteration as

$$\mu_v^{(t+1)} = \mathbb{F}\big(x_v, \sum_{u \in \mathbb{N}(v)} \mu_u^{(t)}\big), \ \forall v \in \mathbb{V}. \tag{3}$$

In this fixed-point update formula, $\mathbb{F}$ is a generic nonlinear mapping which we will specify our choice later. Based on the update formula, one can see that the embedding update process is carried out based on the graph topology, and in a synchronous fashion. A new round of embedding sweeping across the vertices will start only after the embedding update for all vertices from the previous round has finished. It is easy to see that the update also defines a process where the vertex features x_v are propagated to the other vertices via the nonlinear propagation function $\mathbb{F}$. Furthermore, the more iterations one carries out the update, the farther away a vertex feature will propagate to distant vertices and get aggregated nonlinearly at distant vertices. In the end, if one terminates the update process after T iterations, each vertex embedding $\mu_v^{(T)}$ will contain information about its T-hop neighborhood determined by both graph topology and the involved vertex features.

Instead of manually specifying the parameters in the nonlinear mapping $\mathbb{F}$, we model it as a neural network, and learn the parameters in it. In particular, we design $\mathbb{F}$ to have the following form

$$\mathbb{F}(x_v, \sum_{u \in \mathbb{N}(v)} \mu_u) = \tanh(W_1 x_v + \sigma(\sum_{u \in \mathbb{N}(v)} \mu_u)) \tag{4}$$

where x_v is a d-dimensional vector for graph node (or basic-block) level features, W_1 is a $d \times p$ matrix, and p is the embedding size as explained above. To make the nonlinear transformation $\sigma(\cdot)$ more powerful, we will define σ itself as an n layer fully-connected neural network:

$$\sigma(l) = \underbrace{P_1 \times \mathrm{ReLU}(P_2 \times ...\mathrm{ReLU}(P_n l))}_{n \text{ levels}}$$

where P_i $(i = 1, ..., n)$ is a $p \times p$ matrix. We refer to n as the *embedding depth*. Here, ReLU is the rectified linear unit, i.e., $\mathrm{ReLU}(x) = \max\{0, x\}$.

The original learning algorithm for a `Structure2vec` model is through a supervision on an instance (i.e., the ACFG of a binary function) and its label. However, in our case, we do not have labels for binary functions. Therefore, we devise a novel learning approach to train the parameters in $\mathbb{F}$.

Learning Parameters Using Siamese Architecture. We use the Siamese architecture [122] combined with the graph embedding `Structure2vec` network. The Siamese architecture uses two identical graph embedding networks, i.e., `Structure2vec`, which join at the top. Each graph embedding network will take one ACFG g_i ($i = 1, 2$) as its input and outputs the embedding $\phi(g_i)$. The final outputs of the Siamese architecture is the cosine distance of the two embeddings. Further, the two embedding networks share the same set of parameters; thus during training the two networks remain identical.

Given a set of K pairs of ACFGs $\langle g_i, g_i' \rangle$, with ground truth pairing information $y_i \in \{+1, -1\}$, where $y_i = +1$ indicates that g_i and g_i' are similar, i.e. $\pi(g_i, g_i') = 1$, or $y_i = -1$ otherwise. We define the Siamese network output for each pair as

$$\text{Sim}(g, g') = \cos(\phi(g), \phi(g')) = \frac{\langle \phi(g), \phi(g') \rangle}{||\phi(g)|| \cdot ||\phi(g')||}$$

where $\phi(g)$ is produced by the `Structure2vec` model.

Then to train the the model parameters $W_1, P_1, \ldots, P_n$, and W_2, we will optimize the following objective function

$$\min_{W_1, P_1, \ldots, P_n, W_2} \sum_{i=1}^{K} \big(\text{Sim}(g_i, g_i') - y_i\big)^2. \tag{5}$$

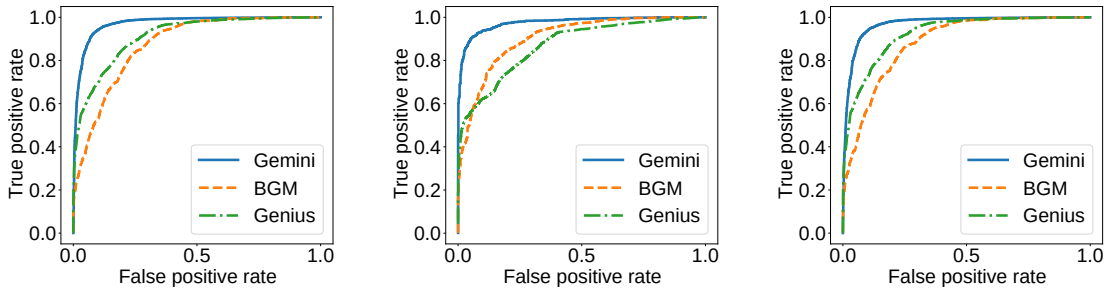
We can optimize the objective (5) with stochastic gradient descent [123]. The gradients of the parameters are calculated recursively according to the graph topology. In the end, once the Siamese network can achieve a good performance (e.g., using AUC as the measure), the training process terminates, and the trained graph embedding network can convert an input graph to an effective embedding suiteable for similarity detection.

Dataset construction. Training the model requires a large amount of data on the ground truth about oracle π, which may be difficult to obtain. To tackle this issue, we construct a training dataset using a *default* policy. Intuitively, the embedding generated for each binary function should try to capture invariant features of the function across different architectures and compilers. We implement this intuition by constructing a dataset as follows using a *default oracle*. Assuming a set of source codes are collected, we can compile them into program binaries for different architectures, using different compilers, and with different optimizations. In doing so, the default oracle determines that two binary functions are similar if they are compiled from the same source code, or dissimilar otherwise. To construct the training dataset, for each binary function g, one other similar function g_1 and one dissimilar function g_2 are sampled to construct two training samples, $\langle g, g_1, +1 \rangle$ and $\langle g, g_2, -1 \rangle$.

4.6.2 Evaluation. We evaluate our approach on the previous vulnerability detection task proposed in [119]. In all evaluations, our approach exhibits superior advantages over the state-of-the-art approach [119]. In particular, we compile OpenSSL toolkit (version 1.0.1f and 1.0.1u) using GNU Compiler Collection (GCC) v5.4. The compiler is set to emit code in x86, MIPS, and Advanced RISC Machine (ARM) architectures, with optimization levels O0-O3. In total, we obtain 18,269 binary files containing 129,365 ACFGs. We split Dataset

Table 6: The Number of ACFGs in Dataset I

	Training	Validation	Testing
x86	30,994	3,868	3,973
MIPS	41,477	5,181	5,209
ARM	30,892	3,805	3,966
Total	103,363	12,854	13,148



(a) Results on the Similarity Testing Set

(b) Results on the Large-Graph Subset

(c) Results on the Small-Graph Subset

Figure 11: ROC Curves for Different Approaches Evaluated on the Testing Similarity Dataset

I into three disjoint subsets of functions for training, validation, and testing respectively. The statistics are presented in Table 6. During the split, we guarantee that no two binary functions compiled from the same source function are separated into two different sets among training, validation and testing sets. In doing so, we can examine whether the model can generalize to unseen functions. We choose this dataset to compare with the previous state-of-the-art [119]. In particular, we compare our approach with two approaches, named BGM and `Genius` respectively, proposed in [119].

Accuracy. We construct a similarity testing dataset as follows: from the testing set in Dataset I, for each ACFG g in the set, we randomly select two ACFGs g_1, g_2 from the testing dataset, such that the ground truth labels of $\langle g, g_1 \rangle$ and $\langle g, g_2 \rangle$ are $+1$ and -1 (i.e., from the same source function vs. not) respectively. This similarity testing dataset consists of 26,265 pairs of ACFGs. Figure 11a illustrates the Receiver Operating Characteristic (ROC) curves for our neural network model as well as two baseline approaches. We can see that that our approach outperforms both Binary Graph Matching (BGM) and Genius by a large margin.

To further examine the performance of our approach on graphs with different sizes, we split the similarity-accuracy testing set into a large-graph subset and a small-graph subset. The large-graph subset contains only pairs of two ACFGs which both have at least 10 vertices. The small-graph subset contains the rest. The ROC curves of different approaches evaluated over the large-graph subset and the small-graph subset are plotted in Figure 11b and Figure 11c respectively. From both figures, we have consistent observations: 1) our approach outperforms both BGM and Genius significantly; and 2) `Genius` outperforms BGM

on small graphs, the BGM performs better than Genius on large graphs, and our approach outperforms both BGM and Genius on large as well as small graphs.

Issues applying to STAC challenge problems. We intended to apply this approach to STAC, but ultimately were unsuccessful. The most important issue was that our approach was adept at identifying whether a new binary function was semantically similar to an existing, vulnerable function. The problem is that we had insufficient training data to effectively assess the method's utility. In the engagements, most vulnerabilities were increasingly challenging, and used concepts not used in previous engagements. This dissimilarity hindered the application/utility of this technique. Potentially, by the end of the STAC program, a sufficient corpus of vulnerable functions could exist to enable training of such tools to find vulnerabilities in code.

4.7 Workbench

The tools we have developed, as well as most third-party tools, are very command-line driven. Results tend to be spread across multiple files, and analyst notes are generally done in a very ad hoc manner. Additionally, sharing analysis results often involves either shared folders or relaying files over some mechanism such as git or email. Shared folders are impractical for analysts spread across multiple organizations; git is ill-suited to many types of files; and distribution channels like email make it easy to omit certain team members, and potentially cumbersome to bring new team members up to speed.

As a solution to these issues, we developed a shared *workbench*. It allows team members to run some of our analysis tools, view files, and add annotations describing progress on analyses. All analyses and notes are fully available to all team members, and adding a new team member involves generating a public key certificate that they can then install into their browser of choice. File organization and visualization is standardized for all team members, which makes it easier to communicate about analysis results.

The workbench is implemented as a distributed service. A coordinator node provides the UI, connects to a local database, and distributes processing tasks to worker nodes. Connecting to the coordinator requires a digital certificate, signed by a trusted authority. Currently, this authority is a self-signed certificate specific to the workbench system, and all users and worker nodes are given client certificates. There is no password access to the coordinator, so all connections must employ Public Key Infrastructure (PKI).

Worker nodes with available processing capacity periodically poll the coordinator for any jobs currently queued but not running. Each job runs in a separate thread, and may start additional threads or external processes, as necessary. The worker also periodically updates the coordinator on the status of all jobs currently assigned to that node. This allows the coordinator to detect failed or stalled workers, and restart or reassign jobs as needed. The system is flexible in the number of workers: A new worker node merely needs to identify itself (using valid credentials) to the coordinator node, and it becomes part of the distributed system. A worker node departing the system simply stops responding; any jobs currently assigned to it will eventually be reassigned.

Both the coordinator and worker nodes are implemented in Java, running in Tomcat containers. We use Tomcat's built-in authentication and SSL capabilities, in order to avoid having to write (and thoroughly test) our own. The emphasis on a secure design allows us to run nodes on either owned or leased (such as Amazon Web Services (AWS) cluster nodes) platforms that might be targeted by intrusion attempts. This prevents our nodes from being recruited into botnets or being used to acquire Sensitive but Unclassified (SBU) DARPA data. The UI is written in Javascript, using Google Closure and JQuery.

The database is a MongoDB instance that provides unauthenticated access to localhost, and so is colocated with the coordinator node. For this reason, no other processes should be run on the same host. All of the challenge problems are loaded into the database, along with any derived or uploaded files, in a single *files* collection, with problem names stored in the *programs* collection. Requests, whether issued by users or by workers as part of a multi-stage processing tasks, are stored in another *requests* collection. Worker nodes' job status information is stored in the *work* collection. Users and workers can annotate files with simple key-value pairs, which are stored in the *annotations* collection. Finally, users

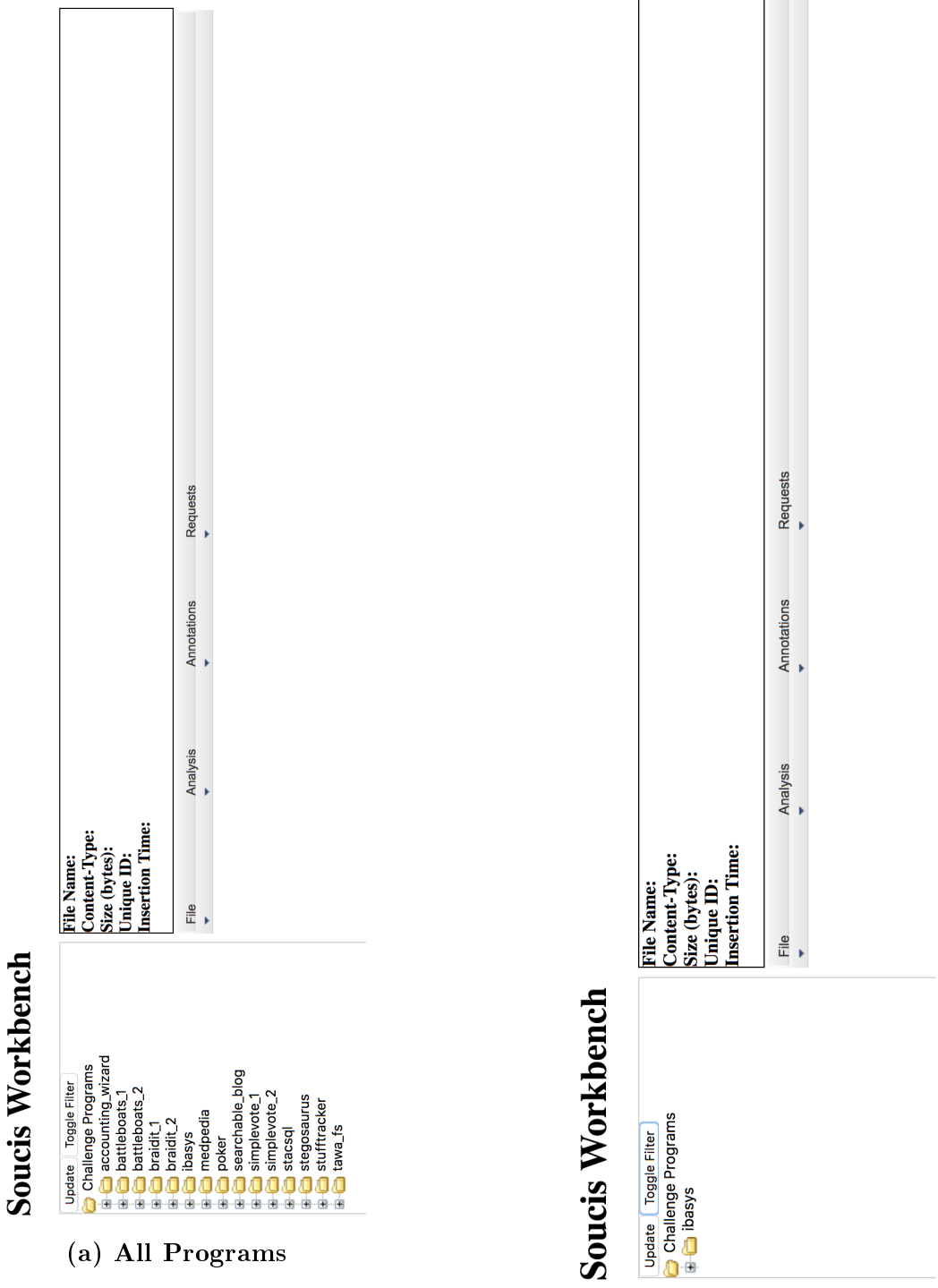
can register interest in a particular problem, allowing the UI to filter out problems that are not of interest (Figure 12), and this information is stored in the *users* collection.

n The workbench integrates with several tools. When a file is selected, the *Analysis* menu indicates which analysis tool is available, as seen in Figure 13. Currently supported are decompilation (Figure 14), fuzzing, and Pidgin taint analysis (Figure 15a). When a user submits a request, it shows up in the *Requests* menu, as shown in Figure 15.

For many known Multipurpose Internet Mail Extensions (MIME) types, the workbench is able to display files with useful formatting (Figure 16). The default is to use `application/octet-stream` as the MIME type with a hexdump display. A context menu allows the user to override this, displaying the file as `text/plain` or `text/html` (Figure 17). The workbench also allows users to download files, and upload both single files and directories.

Some analysis tools, such as the decompiler, generate annotations automatically, as seen in Figure 18. Users can also add annotations manually, in order to capture manual inspection of source code, documentation, and analysis results (Figure 19). Annotations are simple text key-value pairs, but a key can be specified multiple times, and all values for that key will be concatenated in the display. Since annotations cannot currently be edited or deleted, this is a way to attach new pieces of information to a file as work progresses.

There are a number of enhancements to the workbench that we had planned. Some refactoring of the database and UI would improve performance, which can occasionally be slow. Tool integration is incomplete, both in terms of the number of our tools available through the workbench and the functionality of these tools and visualization of output. For example, we would like to be able to connect taint analysis output directly to decompiled source files. There is some limited ability to do this, but there is considerable work still to be done. We would also like to be able to schedule long-running fuzzer jobs, and monitor the output as the jobs run. Currently, the database is not searchable, and this would be a valuable addition. Bulk download of directories would also be helpful, rather than file-by-file downloads. Additionally, we envisioned file editing, both for plain text and JavaScript Object Notation (JSON), but these have not been implemented yet.



(a) All Programs

(b) Programs Filtered by User Interest

Figure 12: Workbench Challenge Programs Listing

Soucis Workbench

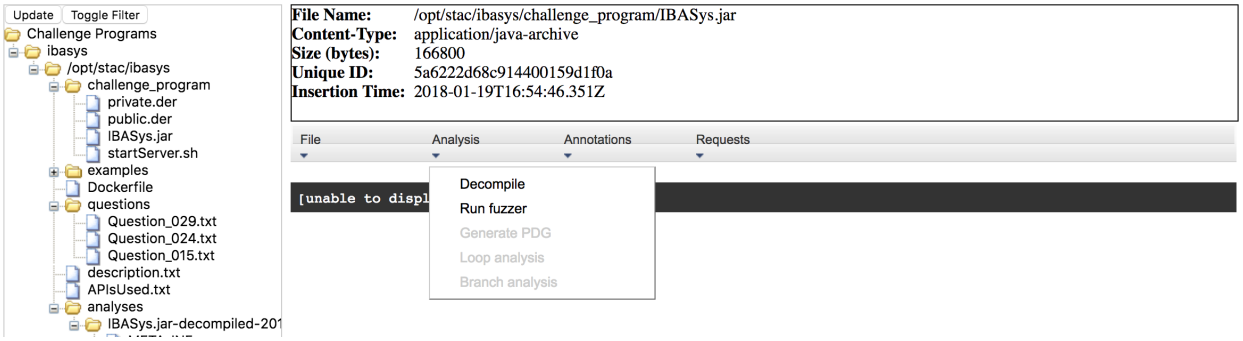


Figure 13: Analysis Menu, as Seen for a JAR File

Soucis Workbench

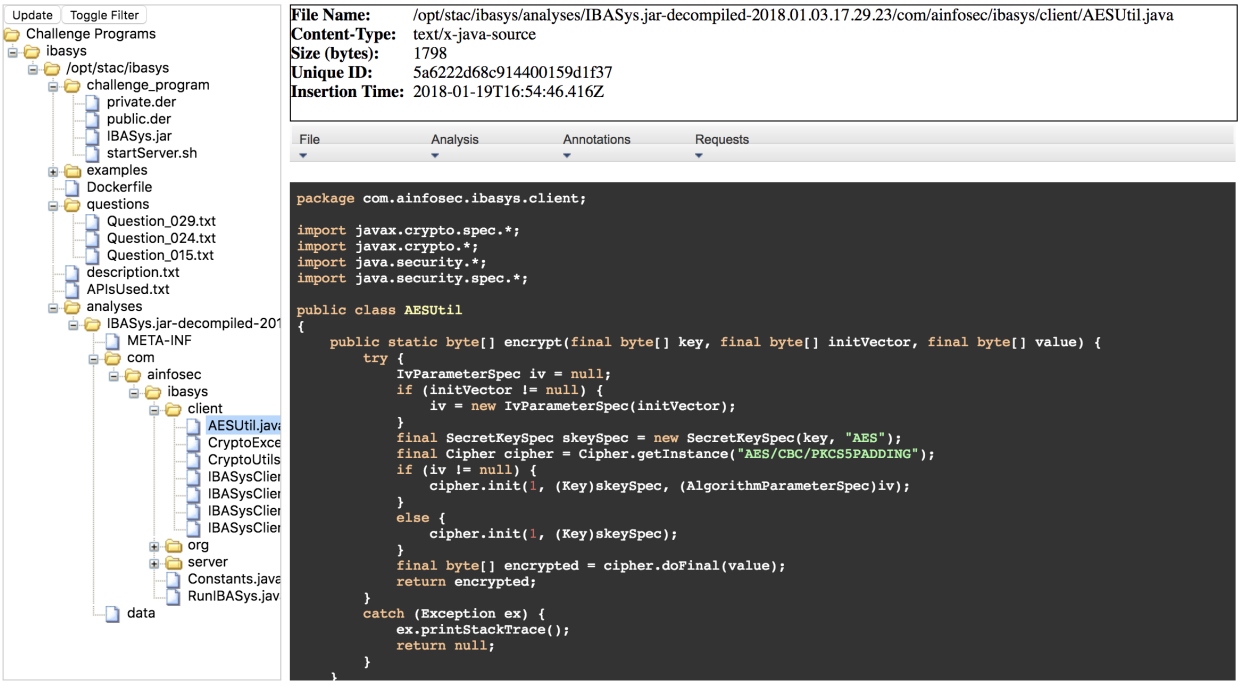
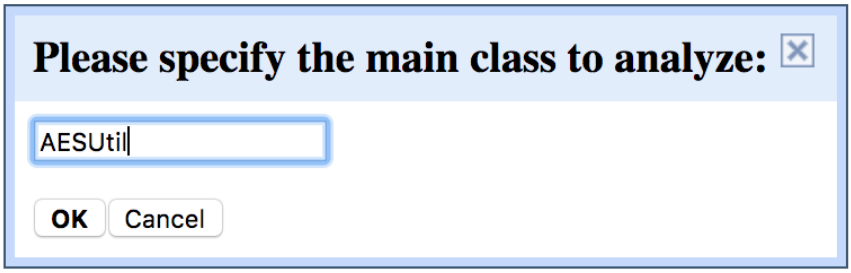
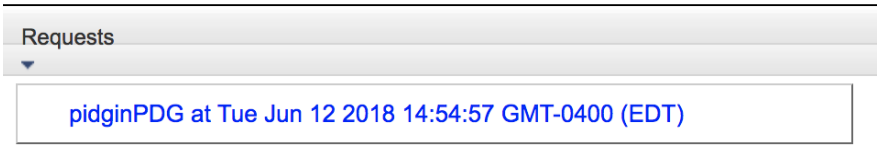


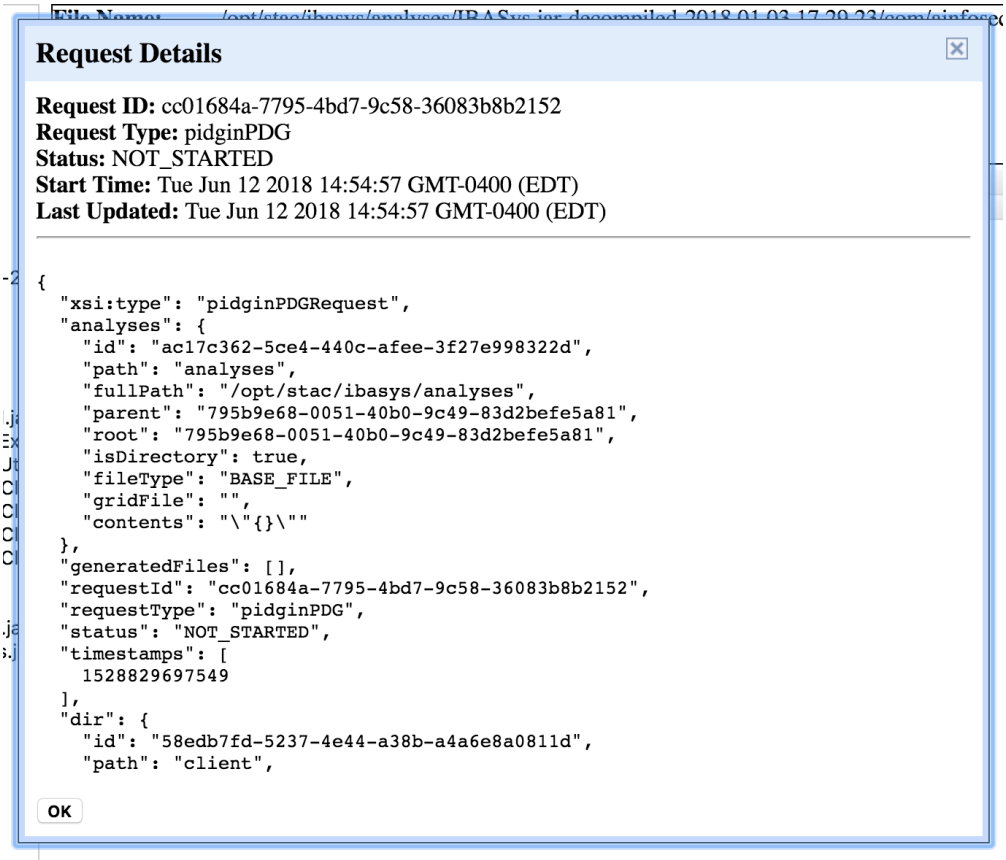
Figure 14: Decompilation Output



(a) Pidgin Program Dependent Graph (PDG) Request Dialog



(b) Request List



(c) Details of Request

Figure 15: Job Requests

File Name:	/opt/stac/ibasys/examples/images/correct_image.jpg
Content-Type:	image/jpeg
Size (bytes):	345273
Unique ID:	5a6222d68c914400159d1f12
Insertion Time:	2018-01-19T16:54:46.365Z

File Analysis Annotations Requests

(a) Image File

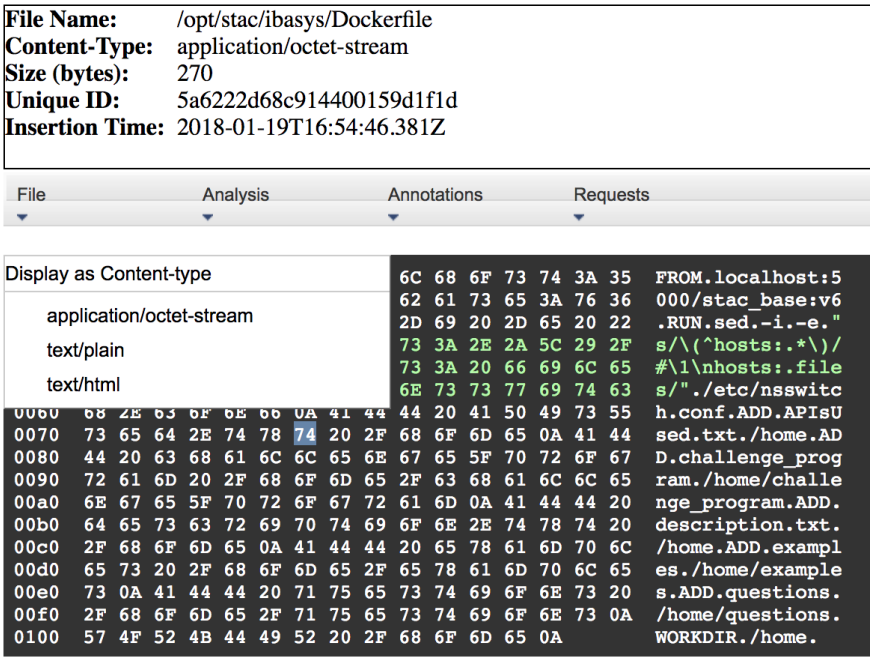
File Name:	/opt/stac/ibasys/Dockerfile
Content-Type:	application/octet-stream
Size (bytes):	270
Unique ID:	5a6222d68c914400159d1f1d
Insertion Time:	2018-01-19T16:54:46.381Z

File Analysis Annotations Requests

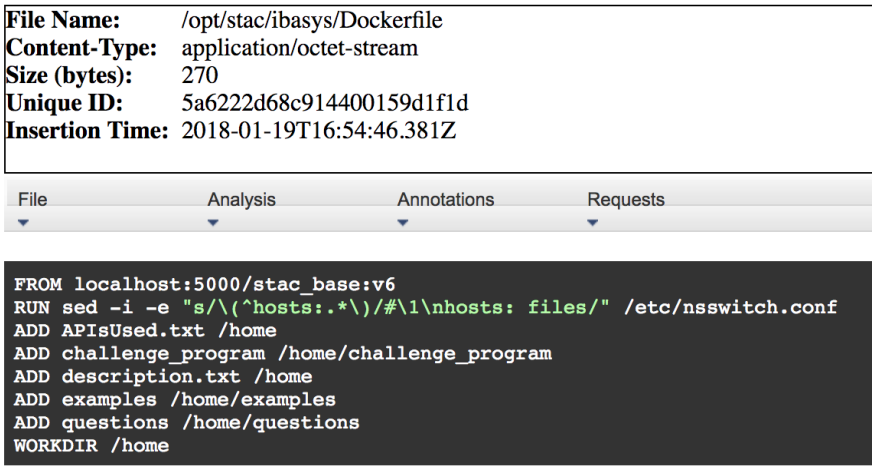
```
0000  46 52 4F 4D 20 6C 6F 63 61 6C 68 6F 73 74 3A 35  FROM.localhost:5
0010  30 30 30 2F 73 74 61 63 5F 62 61 73 65 3A 76 36  000/stac_base:v6
0020  0A 52 55 4E 20 73 65 64 20 2D 69 20 2D 65 20 22  .RUN.sed.-i.-e."
0030  73 2F 5C 28 5E 68 6F 73 74 73 3A 2E 2A 5C 29 2F  s/\(^hosts:.*\)/
0040  23 5C 31 5C 6E 68 6F 73 74 73 3A 20 66 69 6C 65  #\1\nhosts:.file
0050  73 2F 22 20 2F 65 74 63 2F 6E 73 73 77 69 74 63  s/"./etc/nsswitc
0060  68 2E 63 6F 6E 66 0A 41 44 44 20 41 50 49 73 55  h.conf.ADD.APIsU
0070  73 65 64 2E 74 78 74 20 2F 68 6F 6D 65 0A 41 44  sed.txt./home.AD
0080  44 20 63 68 61 6C 6C 65 6E 67 65 5F 70 72 6F 67  D.challenge_prog
0090  72 61 6D 20 2F 68 6F 6D 65 2F 63 68 61 6C 6C 65  ram./home/challe
00a0  6E 67 65 5F 70 72 6F 67 72 61 6D 0A 41 44 44 20  nge_program.ADD.
00b0  64 65 73 63 72 69 70 74 69 6F 6E 2E 74 78 74 20  description.txt.
00c0  2F 68 6F 6D 65 0A 41 44 44 20 65 78 61 6D 70 6C  /home.ADD.exampl
00d0  65 73 20 2F 68 6F 6D 65 2F 65 78 61 6D 70 6C 65  es./home/example
00e0  73 0A 41 44 44 20 71 75 65 73 74 69 6F 6E 73 20  s.ADD.questions.
00f0  2F 68 6F 6D 65 2F 71 75 65 73 74 69 6F 6E 73 0A  /home/questions.
0100  57 4F 52 4B 44 49 52 20 2F 68 6F 6D 65 0A        WORKDIR./home.
```

(b) Unknown File Type

Figure 16: Automatic Rendering



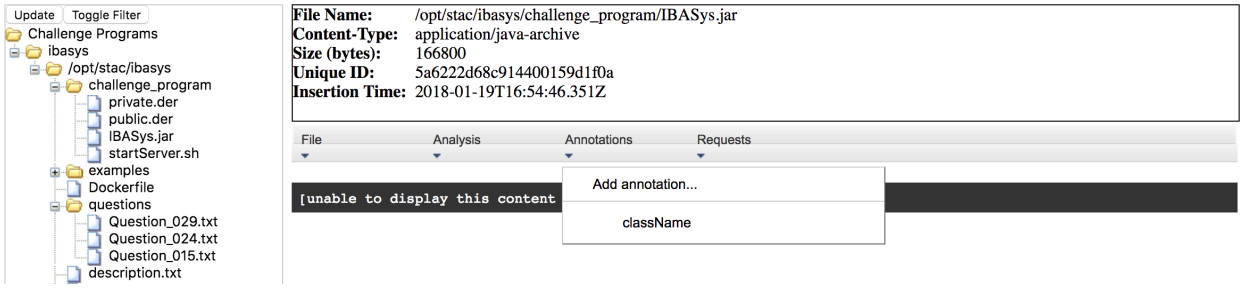
(a) Alternate Display Selection



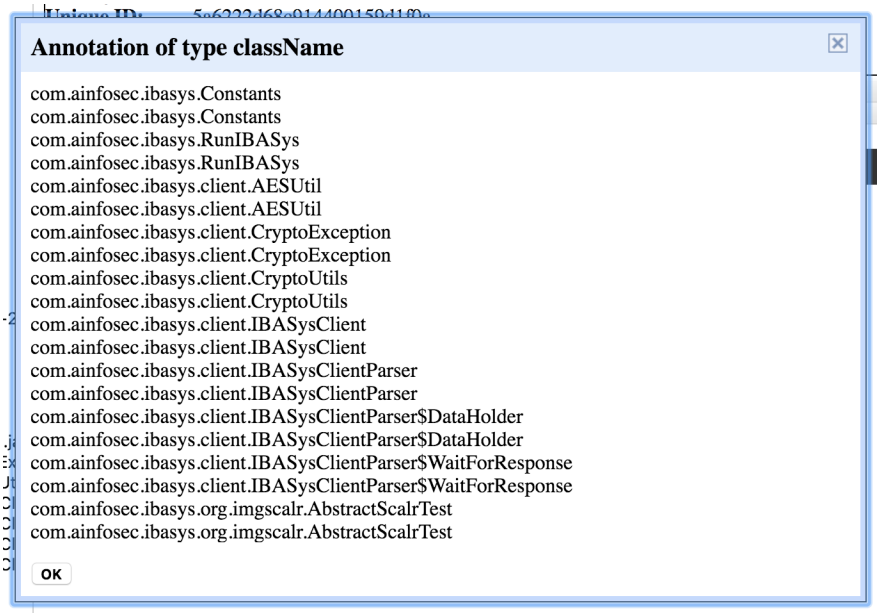
(b) Rendering a File as `text/plain`

Figure 17: Manual Rendering Specification

Soucis Workbench

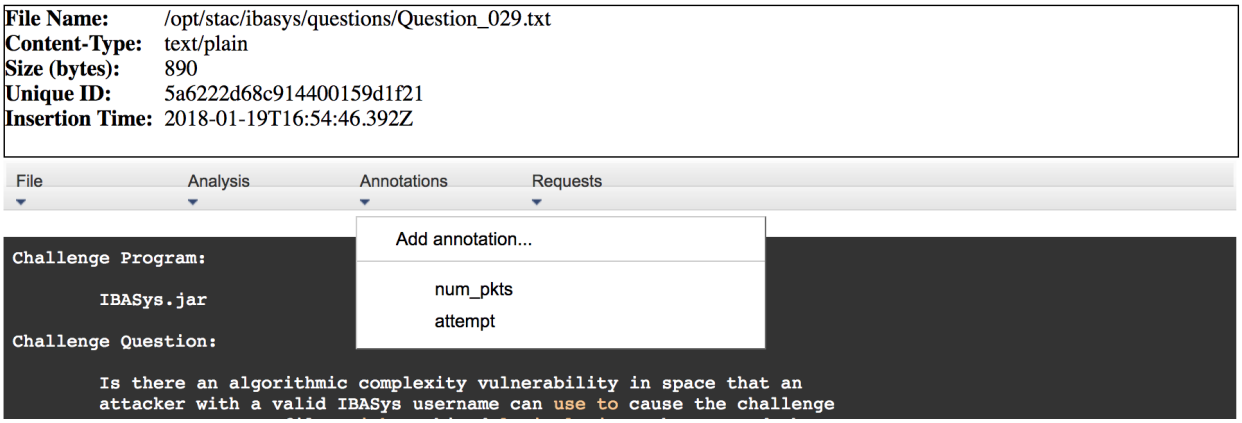


(a) Auto-Generated Annotations

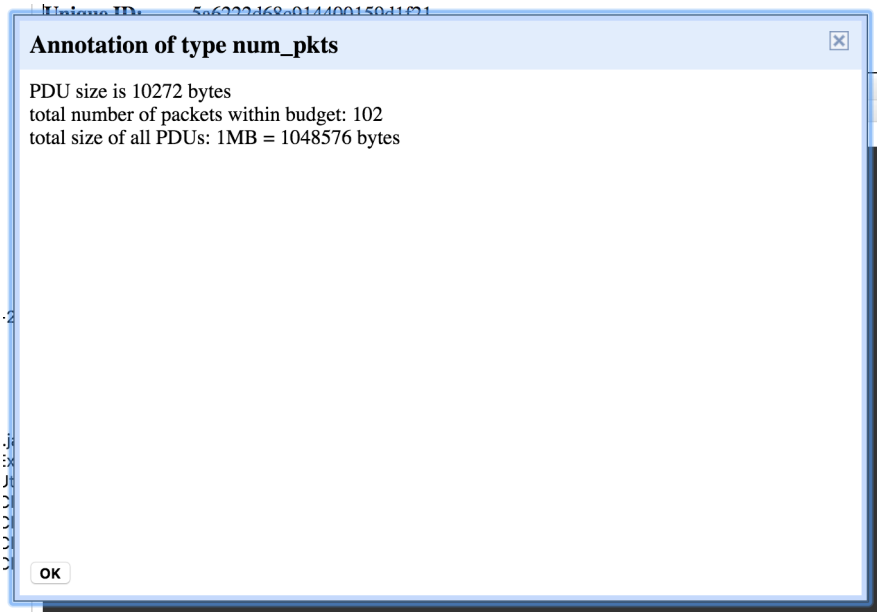


(b) Annotation Details

Figure 18: Annotations from Decompilation



(a) Added User Annotations



(b) Annotation Details

Figure 19: User-Generated Annotations

5 CONCLUSION

This report has presented the technical accomplishments of the project **SOUCIS: Sound Over- & Under-Approximations of Complexity & Information Security**. The goal of the project has been to develop analysis techniques that identify, or prove the absence of, *side channels* and *complexity attacks* in Java Virtual Machine programs.

The technical keystones of the SOUCIS project are the use of *sound over-approximating static analysis* in conjunction with *precise under-approximating analysis*. For the first, which emphasizes verifying properties of *all* of a program's runs, we have carried out the following work:

- *Blazer*: A tool that employs a novel *decomposition* strategy for proving the absence of timing side channels in Java programs. This decomposition strategy is potentially more scalable than *self-composition* strategies typically used for proving properties of multiple program runs (here, that differences in the timing of these runs do not leak secret information).

- JANA: A sound numeric static analysis for Java programs. JANA was developed in conjunction with *Blazer*. Much prior work on *abstract interpretation* existed, but no prior tool of the size and complexity of JANA was available. As it turned out, bringing together ideas from prior, simpler tools was not straightforward, so we carried out a systematic study of an implementation of several features to understand interactions among the combinations.

- NumInv: A numeric invariant generator. This tool combines equation solving and test generation to produce sound, yet precise numeric invariants for interesting programs. Such invariants could be used by *Blazer*, but we also found we could infer resource usage estimates directly, even ones involving interesting, non-linear bounds.

For under-approximate analysis, we considered the following directions:

- Fuzz testing is a testing technique whereby inputs to a program are randomly generated in a feedback loop in an attempt to uncover evidence of a vulnerability. Prior fuzz testing work largely focused on identifying memory errors (resulting in crashes) in file-based C/C++ programs. For SOUCIS, we developed a fuzzer for Java bytecode that instead could find complexity attacks, and contained novel features for fuzzing server programs. These features include checkpointing (to handle statefulness) and I/O mocking (to handle network messages and user interactions). Our fuzzer's architecture could also be outfitted in a way to discover side channel attacks.

  In the process of developing our fuzzer we studied fuzzing more generally. In a systematic review, we found that the empirical evaluation methodology in the published literature is flawed, in that it does not follow best practices, and carried out experiments to demonstrate as much. We make recommendations for improving this practice.

- Machine learning techniques aim to infer functions (such as classification functions) based on input/output examples. We developed a *program similarity detection* technique, based on neural networks, that could be used to identify when two functions are

materially the same. We used this technique to identify vulnerable functions based on prior examples.

To put all of these pieces together we developed a collaborative analyst *workbench*. The workbench permitted organizing the process of studying a potentially vulnerable program, permitting the analyst to more easily run SOUCIS tools, and tools from off the shelf. Results could visualized, stored, and communicated with team members.

The results of the SOUCIS project include code artifacts, technical reports, and published papers appearing in peer-reviewed scientific venues.

6 REFERENCES

[1] Genkin, D., Pipman, I., and Tromer, E. "Get Your Hands Off My Laptop: Physical Side-Channel Key-Extraction Attacks on PCs". In *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems*, 2014.

[2] Kocher, P. C. "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems". In *Proceedings of the International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1996.

[3] Pasareanu, C. S., Phan, Q., and Malacaria, P. "Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT". In *Proceedings of the IEEE workshop on Computer Security Foundations* , 2016.

[4] Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., and Wei, S. "Decomposition Instead of Self-composition for Proving the Absence of Timing Channels". In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pp. 362–375, New York, NY, USA, 2017. ACM.

[5] Cousot, P. and Cousot, R. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1977.

[6] Cousot, P. and Halbwachs, N. "Automatic Discovery of Linear Restraints Among Variables of a Program". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1978.

[7] Wei, S., Mardziel, P., Ruef, A., Foster, J. S., and Hicks, M. "Evaluating Design Tradeoffs in Numeric Static Analysis for Java". In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018*, pp. 653–682, 2018.

[8] Nguyen, T., Antonopoulos, T., Ruef, A., and Hicks, M. "Counterexample-guided approach to finding numerical invariants". In Bodden, E., Schäfer, W., van Deursen, A., and Zisman, A., editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pp. 605–615. ACM, 2017.

[9] "libFuzzer". `https://llvm.org/docs/LibFuzzer.html`. Accessed June 11, 2018.

[10] Klees, G. T., Ruef, A., Cooper, B., Wei, S., and Hicks, M. "Evaluating Fuzz Testing". `https://www.cs.umd.edu/~mwh/papers/fuzzeval-report.pdf`, May 2018. Under review.

[11] Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., and Song, D. "Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection". In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 363–376. ACM, 2017.

[12] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. "A Static Analyzer for Large Safety-critical Software". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, 2003.

[13] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. "The Astrée analyzer". In *Programming Languages and Systems - 27th European Symposium on Programming, (ESOP)*, pp. 21–30. Springer, 2005.

[14] Barthe, G., D'Argenio, P. R., and Rezk, T. "Secure information flow by self-composition". In *Proceedings of the IEEE workshop on Computer Security Foundations*, 2004.

[15] Barthe, G., Crespo, J. M., and Kunz, C. "Relational Verification Using Product Programs". In *Proceedings of the International Symposium on Formal Methods (FM)*, 2011.

[16] Terauchi, T. and Aiken, A. "Secure information flow as a safety problem". In *Proceedings of the Static Analysis Symposium (SAS)*, 2005.

[17] Unno, H., Kobayashi, N., and Yonezawa, A. "Combining type-based analysis and model checking for finding counterexamples against non-interference". In *ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS)*, 2006.

[18] Naumann, D. A. "From Coupling Relations to Mated Invariants for Checking Information Flow". In *European Symposium on Research in Computer Security (ESORICS)*, 2006.

[19] Sousa, M. and Dillig, I. "Cartesian Hoare Logic for Verifying K-safety Properties". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)* , 2016.

[20] Reynolds, J. C. The Craft of Programming. Prentice Hall International series in computer science. Prentice Hall, 1981.

[21] Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., and Hoffmann, J. "Relational cost analysis". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.

[22] Assaf, M., Naumann, D. A., Signoles, J., Totel, E., and Tronel, F. "Hypercollecting semantics and its application to static analysis of information flow". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.

[23] "DARPA Space/Time Analysis for Cybersecurity (STAC) program". `http://www.darpa.mil/program/space-time-analysis-for-cybersecurity`, 2017. Accessed June 11, 2018.

[24] Benton, N. "Simple relational correctness proofs for static analyses and program transformations". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2004.

[25] Yang, H. "Relational separation logic". *Theoretical Computer Science*, 375(1-3), 2007.

[26] Berdine, J., Chawdhary, A., Cook, B., Distefano, D., and O'Hearn, P. W. "Variance analyses from invariance analyses". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[27] Podelski, A. and Rybalchenko, A. "Transition Invariants". In *Proceedings of the ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2004.

[28] Gulwani, S., Jain, S., and Koskinen, E. "Control-flow Refinement and Progress Invariants for Bound Analysis". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, pp. 375–385, 2009.

[29] Gulwani, S. and Zuleger, F. "The Reachability-bound Problem". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, 2010.

[30] Langkemper, S. "The password guessing bug in Tenex". `https://www.sjoerdlangkemper.nl/2016/11/01/tenex-password-bug/`, 2016. Accessed June 11, 2018.

[31] Snelting, G., Giffhorn, D., Graf, J., Hammer, C., Hecker, M., Mohr, M., and Wasserrab, D. "Checking probabilistic noninterference using JOANA". *it - Information Technology*, 56(6), 2014.

[32] Wijaya, D., Talukdar, P. P., and Mitchell, T. "PIDGIN: Ontology Alignment Using Web Text As Interlingua". In *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pp. 589–598, New York, NY, USA, 2013. ACM.

[33] Chen, J., Feng, Y., and Dillig, I. "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, pp. 875–890, New York, NY, USA, 2017. ACM.

[34] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps". *SIGPLAN Not.*, 49(6) pp. 259–269, June 2014.

[35] Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities". In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2000.

[36] Brumley, D. and Boneh, D. "Remote Timing Attacks Are Practical". In *USENIX Security*, 2003.

[37] Bortz, A. and Boneh, D. "Exposing Private Information by Timing Web Applications". In *Proceedings of the World Wide Web*, 2007.

[38] Brodkin, J. "Huge portions of the Web vulnerable to hashing denial-of-service attack". `http://arstechnica.com/business/2011/12/huge-portions-of-web-vulnerable-to-hashing-denial-of-service-attack/`, 2011. Accessed June 11, 2018.

[39] Goodin, D. "Long passwords are good, but too much length can be a DoS hazard". `http://arstechnica.com/security/2013/09/long-passwords-are-good-but-too-much-length-can-be-bad-for-security/`, 2013. Accessed June 11, 2018.

[40] Fu, Z. "Modularly combining numeric abstract domains with points-to analysis, and a scalable static numeric analyzer for Java". In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2014.

[41] Ferrara, P. "Generic combination of heap and value analyses in abstract interpretation". In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2014.

[42] Ferrara, P., Müller, P., and Novacek, M. "Automatic inference of heap properties exploiting value domains". In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2015.

[43] Fähndrich, M. and Logozzo, F. "Clousot: Static contract checking with Abstract Interpretation". *Formal Verification of Object-Oriented Software*, 2010.

[44] Calcagno, C., Distefano, D., O'Hearn, P. W., and Yang, H. "Compositional Shape Analysis by Means of Bi-Abduction". *Journal of the ACM*, 58(6), December 2011.

[45] Henry, J., Monniaux, D., and Moy, M. "Pagai: A path sensitive static analyser". *Electronic Notes in Theoretical Computer Science*, 289, 2012.

[46] Mardziel, P., Magill, S., Hicks, M., and Srivatsa, M. "Dynamic Enforcement of Knowledge-based Security Policies using Probabilistic Abstract Interpretation". *Journal of Computer Security*, 2013.

[47] Cousot, P. and Cousot, R. "Static determination of dynamic properties of programs". In *Proceedings of the Second International Symposium on Programming*, 1976.

[48] Gopan, D., DiMaio, F., Dor, N., Reps, T., and Sagiv, M. "Numeric domains with summarized dimensions". In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[49] De, A. and D'Souza, D. "Scalable Flow-sensitive Pointer Analysis for Java with Strong Updates". In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2012.

[50] Wei, S. and Ryder, B. G. "State-Sensitive Points-to Analysis for the Dynamic Behavior of JavaScript Objects". In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2014.

[51] Ryder, B. G. "Dimensions of Precision in Reference Analysis of Object-oriented Programming Languages". In *Proceedings of the Conference on Compiler Construction (CC)*, 2003.

[52] Bravenboer, M. and Smaragdakis, Y. "Strictly Declarative Specification of Sophisticated Points-to Analyses". In *Proceedings of Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2009.

[53] Shivers, O. Control-Flow Analysis of Higher-Order Languages or Taming Lambda. PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[54] Smaragdakis, Y., Bravenboer, M., and Lhoták, O. "Pick your contexts well: understanding object-sensitivity". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011.

[55] "T. J. Watson Libraries for Analysis (WALA)". `http://wala.sourceforge.net/`. Accessed June 11, 2018.

[56] Miné, A. "APRON numerical abstract domain library". `http://apron.cri.ensmp.fr/library/`. Accessed June 11, 2018.

[57] Jeannet, B. and Miné, A. "Apron: A Library of Numerical Abstract Domains for Static Analysis". In *Proceedings of the Conference on Computer Aided Verification (CAV)*, 2009.

[58] Singh, G., Püschel, M., and Vechev, M. "ETH Library for Numerical Analysis". `http://elina.ethz.ch` and `https://github.com/eth-srl/ELINA`. Accessed June 11, 2018.

[59] Singh, G., Püschel, M., and Vechev, M. T. "Fast polyhedra abstract domain". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2017.

[60] Blackburn, S. M., Garner, R., Hoffman, C., Khan, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In *Proceedings of Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[61] Burnham, K. P., Anderson, D. R., and Huyvaert, K. P. "AIC model selection and multimodel inference in behavioral ecology: some background, observations, and comparisons". *Behavioral Ecology and Sociobiology*, 65(1), 2011.

[62] Ball, T. and Rajamani, S. K. "Automatically Validating Temporal Safety Properties of Interfaces". In *SPIN Symposium on Model Checking of Software*, pp. 103–122. Springer, May 2001.

[63] Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. "Lazy Abstraction". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 58–70. ACM, 2002.

[64] Das, M., Lerner, S., and Seigle, M. "ESP: path-sensitive program verification in polynomial time". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, pp. 57–68, 2002.

[65] De Moura, L. and Bjørner, N. "Z3: An efficient SMT solver". In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pp. 337–340. Springer, 2008.

[66] Leroy, X. "Formal certification of a compiler back-end or: programming a compiler with a proof assistant". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 42–54. ACM, 2006.

[67] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. "The Daikon system for dynamic detection of likely invariants". *Science of Computer Programming*, pp. 35–45, 2007.

[68] Feret, J. "Static analysis of digital filters". In *Programming Languages and Systems - 27th European Symposium on Programming, (ESOP)*, pp. 33–48. Springer, 2004.

[69] Gulwani, S. "SPEED: Symbolic Complexity Bound Analysis". In *Proceedings of the Conference on Computer Aided Verification (CAV)*, pp. 51–62, 2009.

[70] Hoffmann, J., Aehlig, K., and Hofmann, M. "Resource Aware ML". In *Proceedings of the Conference on Computer Aided Verification (CAV)*, pp. 781–786, 2012.

[71] Ngo, V. C., Dehesa-Azuara, M., Fredrikson, M., and Hoffmann, J. "Verifying and Synthesizing Constant-Resource Implementations with Types". In *2017 IEEE Symposium on Security and Privacy*, pp. 710–728, 2017.

[72] Roozbehani, M., Feron, E., and Megrestki, A. "Modeling, Optimization and Computation for Software Verification". In *Hybrid Systems: Computation and Control*, pp. 606–622. ACM, 2005.

[73] Sankaranarayanan, S., Sipma, H. B., and Manna, Z. "Scalable analysis of linear systems using mathematical programming". In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pp. 25–41. Springer, 2005.

[74] Nguyen, T. Automating Program Verification and Repair Using Invariant Analysis and Test-input Generation. PhD thesis, University of New Mexico, August 2014.

[75] Nguyen, T., Kapur, D., Weimer, W., and Forrest, S. "DIG: A Dynamic Invariant Generator for Polynomial and Array Invariants". *ACM Transactions on Software Engineering and Methodology*, 23(4) pp. 30:1–30:30, 2014.

[76] Padhi, S., Sharma, R., and Millstein, T. "Data-driven Precondition Inference with Learned Features". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, pp. 42–56, 2016.

[77] Garg, P., Neider, D., Madhusudan, P., and Roth, D. "Learning Invariants Using Decision Trees and Implication Counterexamples". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 499–512, 2016.

[78] Sharma, R., Gupta, S., Hariharan, B., Aiken, A., Liang, P., and Nori, A. V. "A data driven approach for algebraic loop invariants". In *Programming Languages and Systems - 27th European Symposium on Programming, (ESOP)*, pp. 574–592. Springer, 2013.

[79] Nguyen, T., Kapur, D., Weimer, W., and Forrest, S. "Using Dynamic Analysis to Discover Polynomial and Array Invariants". In *Proceedings of the International Conference on Software Engineering, (ICSE)*, pp. 683–693. IEEE, 2012.

[80] Cadar, C., Dunbar, D., and Engler, D. R. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In *Proceedings of the Conference on Operating Systems Design and Implementation*, volume 8, pp. 209–224. USENIX Association, 2008.

[81] Dillig, I., Dillig, T., Li, B., and McMillan, K. "Inductive Invariant Generation via Abductive Inference". In *Proceedings of Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pp. 443–456, 2013.

[82] Gulwani, S., Mehra, K. K., and Chilimbi, T. M. "SPEED: precise and efficient static estimation of program computational complexity". In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 127–139, 2009.

[83] "American Fuzzing Lop (AFL)". `http://lcamtuf.coredump.cx/afl/`. Accessed June 11, 2018.

[84] Petsios, T., Zhao, J., Keromytis, A. D., and Jana, S. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities". In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pp. 2155–2168, New York, NY, USA, 2017. ACM.

[85] Wang, J., Chen, B., Wei, L., and Liu, Y. "Skyfire: Data-Driven Seed Generation for Fuzzing". In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pp. 579–594, 2017.

[86] Shastry, B., Leutner, M., Fiebig, T., Thimmaraju, K., Yamaguchi, F., Rieck, K., Schmid, S., Seifert, J., and Feldmann, A. "Static Program Analysis as a Fuzzing Aid". In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, pp. 26–47, 2017.

[87] Grieco, G., Ceresa, M., and Buiras, P. "QuickFuzz: an automatic random fuzzer for common file formats". In *Proceedings of the 9th International Symposium on Haskell*, pp. 13–20. ACM, 2016.

[88] Grieco, G., Ceresa, M., Mista, A., and Buiras, P. "QuickFuzz Testing for Fun and Profit". *Journal of Systems Software*, 134(C) pp. 340–354, December 2017.

[89] Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., and Vigna, G. "DIFUZE: Interface Aware Fuzzing for Kernel Drivers". In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pp. 2123–2138, 2017.

[90] Cha, S. K., Woo, M., and Brumley, D. "Program-Adaptive Mutational Fuzzing". In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pp. 725–741, Washington, DC, USA, 2015. IEEE Computer Society.

[91] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. "Vuzzer: Application-aware evolutionary fuzzing". In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[92] Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. "Unleashing Mayhem on Binary Code". In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pp. 380–394, Washington, DC, USA, 2012. IEEE Computer Society.

[93] Li, Y., Chen, B., Chandramohan, M., Lin, S.-W., Liu, Y., and Tiu, A. "Steelix: program-state based binary fuzzing". In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 627–637. ACM, 2017.

[94] Chen, P. and Chen, H. "Angora: Efficient Fuzzing by Principled Search". In *Security and Privacy (SP), 2018 IEEE Symposium on*. IEEE, 2018.

[95] Kargén, U. and Shahmehri, N. "Turning Programs Against Each Other: High Coverage Fuzz-testing Using Binary-code Mutation and Dynamic Slicing". In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 782–792, New York, NY, USA, 2015. ACM.

[96] Lin, Y., Liao, F., Huang, S., and Lai, Y. "Browser fuzzing by scheduled mutation and generation of document object models". In *International Carnahan Conference on Security Technology, ICCST 2015, Taipei, Taiwan, September 21-24, 2015*, pp. 1–6, 2015.

[97] Yoo, H. and Shon, T. "Grammar-based adaptive fuzzing: Evaluation on SCADA modbus protocol". In *2016 IEEE International Conference on Smart Grid Communications, SmartGridComm 2016, Sydney, Australia, November 6-9, 2016*, pp. 557–563, 2016.

[98] Han, H. and Cha, S. K. "IMF: Inferred Model-based Fuzzer". In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pp. 2345–2358, 2017.

[99] Iannillo, A. K., Natella, R., Cotroneo, D., and Nita-Rotaru, C. "Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations". In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*, pp. 1–11, 2017.

[100] Stephens, N., Grosen, J., Salls, C., Dutcher, A., and Wang, R. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2016.

[101] Zhang, B., Ye, J., Feng, C., and Tang, C. "S2F: Discover Hard-to-Reach Vulnerabilities by Semi-Symbolic Fuzz Testing". In *13th International Conference on Computational Intelligence and Security, CIS 2017, Hong Kong, China, December 15-18, 2017*, pp. 548–552, 2017.

[102] Xu, W., Kashyap, S., Min, C., and Kim, T. "Designing New Operating Primitives to Improve Fuzzing Performance". In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pp. 2313–2328, New York, NY, USA, 2017. ACM.

[103] Schumilo, S., Aschermann, C., Gawlik, R., Schinzel, S., and Holz, T. "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels". In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pp. 167–182, 2017.

[104] Henderson, A., Yin, H., Jin, G., Han, H., and Deng, H. "VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices". In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, pp. 3–25, 2017.

[105] Peng, H., Shoshitaishvili, Y., and Payer, M. "T-Fuzz: fuzzing by program transformation". In *Security and Privacy (SP), 2018 IEEE Symposium on*. IEEE, 2018.

[106] Han, W., Joe, B., Lee, B., Song, C., and Shin, I. "Enhancing Memory Error Detection for Large-Scale Applications and Fuzz Testing". In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[107] Petsios, T., Tang, A., Stolfo, S. J., Keromytis, A. D., and Jana, S. "NEZHA: Efficient Domain-Independent Differential Testing". In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pp. 615–632, 2017.

[108] Haller, I., Slowinska, A., Neugschwandtner, M., and Bos, H. "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations". In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pp. 49–64, Berkeley, CA, USA, 2013. USENIX Association.

[109] Lemieux, C. and Sen, K. "FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage". *arXiv preprint arXiv:1709.07101*, 2017.

[110] Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. "Directed Greybox Fuzzing". In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[111] Böhme, M., Pham, V.-T., and Roychoudhury, A. "Coverage-based Greybox Fuzzing As Markov Chain". In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pp. 1032–1043, New York, NY, USA, 2016. ACM.

[112] Woo, M., Cha, S. K., Gottlieb, S., and Brumley, D. "Scheduling Black-box Mutational Fuzzing". In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pp. 511–522, New York, NY, USA, 2013. ACM.

[113] Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., and Brumley, D. "Optimizing Seed Selection for Fuzzing". In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pp. 861–875, Berkeley, CA, USA, 2014. USENIX Association.

[114] Bell, J. and Pina, L. "CROCHET: Checkpoint and Rollback via Lightweight Heap Traversal on Stock JVMs". In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2018.

[115] "laf-tintel". `https://gitlab.com/laf-intel/laf-llvm-pass`. Accessed October 14, 2017.

[116] Molnar, D., Li, X. C., and Wagner, D. A. "Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs". In *Proceedings of the 18th Conference on USENIX Security Symposium*, pp. 67–82, 2009.

[117] "Darpa Cyber Grand Challenge (CGC) Binaries". `https://github.com/CyberGrandChallenge/`. Accessed June 11, 2018.

[118] Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W. K., Ulrich, F., and Whelan, R. "LAVA: Large-Scale Automated Vulnerability Addition". In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pp. 110–121, 2016.

[119] Feng, Q., Zhou, R., Xu, C., Cheng, Y., Testa, B., and Yin, H. "Scalable Graph-based Bug Search for Firmware Images". In *ACM Conference on Computer and Communications Security (CCS'16)*, October 2016.

[120] "The IDA Pro Disassembler and Debugger". `http://www.datarescue.com/idabase/`, 2015. Accessed June 11, 2018.

[121] Dai, H., Dai, B., and Song, L. "Discriminative Embeddings of Latent Variable Models for Structured Data". In *International Conference on Machine Learning*, 2016.

[122] Bromley, J., Guyon, I., LeCun, Y., Sickinger, E., and Shah, R. "Signature Verification Using A "Siamese" Time Delay Neural Network". In *Proceedings of the Conference on Neural Information Processing Systems (NIPS)*, 1993.

[123] Kingma, D. P. and Ba, J. "Adam: A method for stochastic optimization". *arXiv preprint arXiv:1412.6980*, 2014.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

ACFG	Attributed Control Flow Graph
AFL	American Fuzzy Lop
AFLFast	National University of Singapore's extension of AFL
ARM	Advanced RISC Machine – a family of RISC architectures for computer processors
cex, cexs	counterexample(s)
BGM	Binary Graph Matching
CEGIR	Counter-Example Guided Invariant Generation
CFG	Control Flow Graph
CGC	DARPA Cyber Grand Challenge
CVE	Common Vulnerabilities and Exposures
DARPA	Defense Advanced Research Products Agency
DIG	Dynamic Invariant Generator
ELINA	ETH Library for Numerical Analysis
ESOP	European Symposium on Programming
GUI	Graphical User Interface
HOLA	Hoare Logic with Abduction program suite
I/O	Input/Output
ICE	Implication Counter-Examples learning model
JANA	Java Numeric Analysis - University of Maryland's sound numeric static analysis for Java
JAR	Java Archive
JSON	JavaScript Object Notation
LAVA	Large-scale Automated Vulnerability Addition
LAVA-M	A second LAVA corpus, with more than one bug at a time injected into source code
MIME	Multipurpose Internet Mail Extensions
MIPS	A RISC instruction set architecture (originally Microprocessor without Interlocked Pipeline Stages)
NLA	Non-Linear Arithmetic program suite
NumInv	Numeric Invariant hybrid analysis developed by University of Maryland
OpenSSL	A robust, commercial-grade, and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols.
PDG	Program Dependenc Graph
PIE	Precondition Inference Engine
PKI	Public Key Infrastructure
PLDI	Programming Language Design and Implementation Conference
RISC	Reduced Instruction Set Computer
ROC	Receiver Operating Characteristic
RQ	Research Question
SBU	Sensitive but Unclassified
SC	Side Channel

SOUCIS	Sound Over- and Under-Approximations of Complexity and Information Security
SSL	Secure Sockets Layer
STAC	Space/Time Analysis for Cybersecurity
UI	User Interface
WALA	T. J. Watson Library for Analysis
x86	A family of Intel-based instruction set architectures