AFRL-RI-RS-TR-2020-057

INTEGRATED SYMBOLIC EXECUTION FOR SPACE-TIME ANALYSIS OF CODE (ISSTAC)

VANDERBILT UNIVERSITY

MARCH 2020

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-2020-057 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WALTER S. KARAS
Work Unit Manager

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

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

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2020	FINAL TECHNICAL REPORT	APR 2015 – SEP 2019

4. TITLE AND SUBTITLE

INTEGRATED SYMBOLIC EXECUTION FOR SPACE-TIME ANALYSIS OF CODE (ISSTAC)

5a. CONTRACT NUMBER
FA8750-15-2-0087

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61102E

6. AUTHOR(S)

Daniel Balasubramanian, Tevfik Bultan, Gabor Karsai,
Corina Pasareanu

5d. PROJECT NUMBER
STAC

5e. TASK NUMBER
VA

5f. WORK UNIT NUMBER
ND

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Vanderbilt University
1025 16th Ave South
Nashville TN 37212

8. PERFORMING ORGANIZATION REPORT NUMBER

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

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

DARPA/I20
675 North Randolph St.
Arlington VA 22203

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

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2020-057

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
Cybersecurity hinges upon finding vulnerabilities in software systems before they are deployed in an environment open to malicious actors. As the implementation flaws in software systems are eliminated by increasingly sophisticated software analysis techniques, attacks relying on the inherent space-time complexity of algorithms used for building software systems are gaining prominence. When an adversary can inexpensively generate inputs that induce behaviors with expensive space-time resource utilization at the defender's end, in addition to mounting denial-of-service attacks, the adversary can also use the same inputs to facilitate side-channel attacks in order to infer some secret from the observed system behavior. In this project our objective was to develop automated and semi-automated analysis techniques and implement them in an industrial-strength tools that allow the efficient analysis of software (in the form of Java bytecode) with respect to these problems rapidly enough for inclusion in a state-of-the-art development process.

15. SUBJECT TERMS

Static code analysis, cybersecurity, side-channel analysis, symbolic execution, fuzzing

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WALTER S. KARAS**
U	U	U	UU	28	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

List of Figures

1 SUMMARY

Cybersecurity hinges upon finding vulnerabilities in software systems before they are deployed in an environment open to malicious actors. As the implementation flaws in software systems are eliminated by increasingly sophisticated software analysis techniques, attacks relying on the inherent space-time complexity of algorithms used for building software systems are gaining prominence. When an adversary can inexpensively generate inputs that induce behaviors with expensive space-time resource utilization at the defender's end, in addition to mounting denial-of-service attacks, the adversary can also use the same inputs to facilitate side-channel attacks in order to infer some secret from the observed system behavior.

In this project our objective was to develop automated and semi-automated analysis *techniques* and implement them in an industrial-strength *tools* that allow the efficient analysis of software (in the form of Java bytecode) with respect to these problems rapidly enough for inclusion in a state-of-the-art development process. The analysis was to result in (1) estimates for the worst-case space-time complexity and algorithms, (2) input constraints that implicitly characterize inputs to the algorithms that trigger the worst-case behavior, (3) concrete test examples for "bad" inputs, and (4) estimates for the information leakage from the system through observables related to time and memory resource utilization.

2 INTRODUCTION

The project's main objective was to perform the code analysis using a radically improved form of symbolic execution of Java bytecode that was expected to yield the results described above. Symbolic execution is a very powerful code analysis technique that not only implicitly enumerates all program execution paths, but is also capable of constructing the inputs that trigger those paths. To address the scalability, speed, and accuracy issues we have (1) applied new techniques to worst-case and side-channel analysis, (2) developed a novel and efficient model-counting constraint solver that was used in constructing the malicious inputs and computing the leakage, (3) implemented a cloud-based platform for distributed symbolic execution where multiple program execution paths are explored in parallel, (4) integrated fuzzing and symbolic execution to improve the performance of the analysis tool, (5) developed a black-box method for side-channel analysis, and (6) implemented a novel dynamic symbolic execution engine for Java.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

3.1 Challenges of worst case analysis

Our goal was to use symbolic execution to compute the worst-case complexity of Java byte code programs. The most challenging aspect of the work has been *scalability*. A simple algorithm for computing worst-case execution time is to perform a symbolic execution over the program up to a large depth and to *observe* the size of the longest terminating path L (in terms of some cost function defined over program paths). While such a naive approach would find the worst-case executions for any (bounded) program, it could hardly scale to any program of realistic size. We therefore had to solve the challenge of developing *heuristics* that

search the symbolic state space of the program in an intelligent way, to guide the analysis in an intelligent manner. While there has been a large amount of work devoted to heuristic search in program analysis, the majority of work has been done in the simple setting of guiding the analysis towards increasing coverage or finding program bugs (run-time errors or assert violation), and therefore could not be applied to our problem. There has been considerable less work on devising heuristics for worst case complexity. We therefore had to build new techniques and tools to address the problem. We addressed these challenges by developing heuristics tailored for guiding symbolic execution worst-case analysis such as Symbolic PathFinder-Worst Case Analysis (SPF-WCA) [14] and Canopy [15] as well as fuzzing and hybrid approaches, that combine fuzzing with symbolic execution and custom heuristics to guide the search for worst-case paths, e.g. Kelinci-Worst Case Analysis (WCA) [12] and Badger [18].

3.2 Challenges for side channel analysis

Side-channel attacks recover secret content in programs from non-functional characteristics of computations, such as time or memory consumption. Typical goals of side-channel attacks are the recovery of cryptographic keys and private information about users. Previous work on countering side-channel attacks focuses on modifications of the hardware platform for specific algorithms such as Intel's implementation of the Advanced Encryption Standard or requiring modifications to the platform [22] that are so significant that their rapid adoption seems unlikely. We proposed a symbolic execution approach for the automatic analysis of Java bytecode programs that delivers formal security guarantees that cover all possible executions of the corresponding system. The security guarantees are *quantitative* bounds on the amount of information that is contained in the side-channel observations of timing and memory based attacks. Over previous static analysis approaches [9] we worked with a *precise execution and memory model* for Java, as provided by Java Pathfinder's (JPF) custom Java Virtual Machine (JVM), allowing us to provide more precise results. Furthermore, JPF enables precise reasoning about multi-threading and the effects of garbage collection which were not addressed in previous work.

3.3 Challenges for cloud-based symbolic execution

Even though symbolic execution is often referred to as "embarassingly parallel", there are practical issues that must be resolved when implementing a parallel (distributed) version, such as the underlying platform on which the analysis will be performed. Using an off-the-shelf analysis platform, such as Apache Spark, brings the benefit of mature technology with built-in tools. On the other hand, we found that using heavyweight tools such as these can bring overhead and issues that are difficult to debug [3].

Ultimately, we used several custom components for our cloud-based symbolic execution framework [3]. This gave us the capability to run multiple related analyses (JPF, SPF, and SPF-WCA) on top of the same framework.

3.4 Overall approach

We proposed a comprehensive symbolic execution approach for the time/space analysis of Java bytecode programs. Symbolic execution [13] is a systematic program analysis technique which efficiently explores multiple program behaviors all at once, by manipulating symbolic constraints collected over program paths. In recent years symbolic execution has shown significant success in practice, scaling to millions of lines of code [10] and finding applications in the security domain [2, 8]. The technique is attractive because it is automatic, produces no false alarms and generates inputs that can be re-run by developers. We will build upon SPF [20]—a mature symbolic execution tool for Java bytecode, part of the JPF verifica-tion tool-set. JPF has been under development at the National Aeronautics and Space Administration (NASA) Ames Research Center since 1999 and was open-sourced in 2005. The tool-set has an active user and developer base in academia, industry, and government agencies and labs (initially 6000 downloads/month). The largest program analyzed was a web-application with 276,556,150 instructions executed.

Applying SPF to our proposed effort of automatically detecting time/space vulnerabilities and quantifying the results required significant extensions and additions to both the underlying techniques and algorithms as well as the tool's implementation and infrastructure. Additionally, we built and integrated a cloud-based computing framework on top of which SPF can run. Our overall approach is depicted in Figure 1. The proposed tool-set takes as input a Java bytecode program which is analyzed with respect to worst-case complexity and side-channel attacks. The worst case analysis tool produces the worst-case complexity (upper and lower) bounds together with actual test inputs (and input constraints) that expose the worst case behavior. The side-channel analysis tool produces an upper bound of channel leakage together with a *confidence* measure on the analysis results. The toolset has the following components, developed by our technical tasks.

1. The *worst-case complexity analysis* (WCA) component implements efficient algorithms for computing lower and upper bounds on the algorithmic complexity of a program, and produces input constraints and test inputs that expose the vulnerabilities. This is achieved through a combination of techniques that efficiently explore both an under-and over-approximation of the program. The first technique performs a beam-search symbolic execution to quickly find the worst case paths while the second technique performs a static analysis along the paths given by the first technique defining a program slice that is smaller and easier to analyze than the whole program. The under-and over-approximation techniques are used in a complementary way allowing us to minimize the imprecision introduced by each one of them separately.

2. The *side-channel analysis* (SCA) component implements a symbolic quantitative information flow analysis that precisely quantifies the leakage of discovered channels. The technique takes into account the effects of garbage collection and handles multithreading. Further, loop acceleration is used to speed-up the analysis.

3. At the heart of our toolset is a powerful automata based *model-counting constraint-solver* (MCS) capable of handling complex constraints, involving numeric and string expressions, as well as model counting necessary for quantifying the analysis results. Further, the solver provides widening and transitive closure operations for accelerating the analysis techniques.

4. Horizontal scaling is provided by building a *cloud-based distributed symbolic execution framework* on which the processing is performed.

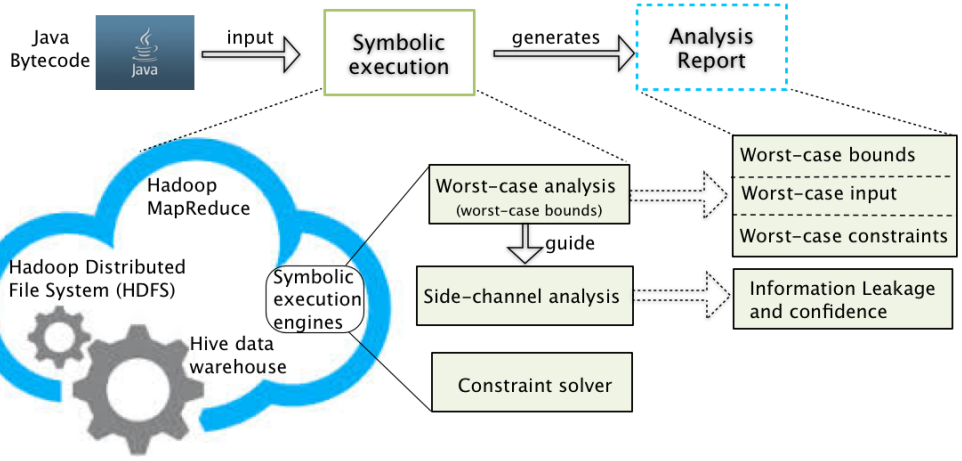

Figure 1. Overview of Our Framework

Evaluation metrics: We briefly summarize our metrics for the evaluation of the performance of the tools.

- *Scalability*: We provided scalability through algorithmic advancements and through a parallel implementation using a cloud-based distributed architecture. The former was achieved as follows: The worst-case execution analysis performs an iterative "beam-search" symbolic execution to quickly guide the symbolic execution towards the paths that consume the most time/space. This analysis is complemented by a static analysis that is guided by the paths and bounds computed by the beam-search symbolic execution, for increased efficiency. Both worst-case and side-channel analysis techniques are implemented using an iterative approach that can benefit from caching constraints and pruning previously analyzed paths. Further, loop acceleration is used to speed-up the computations. The distributed architecture provides true horizontal scaling: The number of parallel execution paths that can be explored at a time is directly proportional to the number of nodes in the cloud.
- *Speed*: Speed can be measured in terms of the sheer computational speed of the tool and the amount of manual intervention that is required. We provide a speedup over symbolic execution based techniques using algorithmic improvements and a distributed

architecture as described above. Our techniques are fully automated, but human input can be used to further tune them, e.g. to explore deep paths of interest. Further, the techniques are "anytime," i.e., the more they run, the more precise the obtained results, while at the same time giving useful intermediate results.

- *Accuracy*: The proposed techniques are carefully crafted to maximize precision and eliminate false alarms. For WCA, we provide not only upper and lower bounds, but also actual inputs that expose worst case complexity (thus "no false alarms"). The side channel analysis not only discovers side channels, but it also quantifies the leakage and provides a confidence measure in the results; approximation is introduced carefully, and iterative techniques are used to produce results that become more precise with each iteration.

4 RESULTS AND DISCUSSION

4.1 Static analysis

Our goal with the static analysis was to narrow down which areas of a program are likely to contain vulnerabilities. The results of the static analysis are presented to the user in an intuitive manner so that potential vulnerabilities can be quickly inspected. Static analysis is a good choice for the initial pass because it can scale to large programs, although it may lead to false positives.

We implemented a tool for static analysis and visualization named Janalyzer. Internally, Janalyzer uses the Watson Libraries for Analysis (WALA) [11], which is a framework for Java bytecode analysis. In a nutshell, Janalyzer performs many static analyses based on the computed call-graph and control-flow information, such as loop analysis, unbalanced branch identification, and dependency analysis. Moreover, Janalyzer has a graphical user interface (GUI) with which users interact, and seamlessly integrates with other dynamic analysis tools, like jBond, which we developed for this project.

Janalyzer uses the Averroes [1] tool to construct call graphs. First, Averroes is used to generate a placeholder library that summarizes the original library in terms of method calls. Then, Janalyzer uses the WALA [11] to analyze the original program along with the placeholder library and to generate a call graph. Without this type of approximation, the generation of the call graph alone can take a long time depending on the size of the libraries used by the target program. To increase the precision of the call graph when the target program uses reflection, Janalyzer uses the Tamiflex [6] tool to dynamically monitor the program and record which classes are loaded reflectively. Afterwards, this list of classes is given to Averroes so that it can know precisely which classes to use in reflective calls. The generated call graph is then sound with respect to the dynamic runs. Moreover, Janalyzer also allows the user to input additional methods that should be considered as entry points. This is needed for web-based applications with callbacks.

Loops, particularly nested loops, are a natural source for potential time and space vulnerabilities. As the nesting level of a loop increases, resource consumption in a program

(execution time, memory consumed, packets sent over a network) inside the loops also increases. Thus, identifying nested loops may help identify complexity vulnerabilities.

The difficulty is that realistic programs may contain thousands of loops, most of which are not vulnerable. Our Janalyzer uses two heuristics to rank loops. First, it identifies all intra- and inter-procedural nested loops (i.e., loops that are nested across method calls) and ranks them according to their nesting level. Second, it identifies loops that contain "suspicious" exit conditions. We define a loop exit condition as suspicious if a variable upon which it depends is possibly modified inside the loop body. For instance, a loop whose exit condition depends on a stack being non-empty is suspicious if elements were pushed onto the stack inside the loop body itself.

A list of loops is created and ordered by nesting level. In principle, this list can form the starting point of a fully automatic analysis, where each suspicious method is further analyzed with symbolic execution, possibly in parallel, to confirm the vulnerability. In practice, the list is visualized and displayed to the user, who can choose which methods to analyze further. The visualization overlays the loop sequences on top of the program's call graph to display them in context. When a particular loop sequence is selected, the methods containing the loops are displayed.

We have developed a simple technique for identifying *unbalanced branches* in the code, with the goal of discovering potential side channels. The technique is built on control-dependency graph (CDG), and for each branching point it derives which parts of paths are common and which are not.

The Janalyzer takes the semantics of methods such as *Thread.sleep()* into account in its cost model. For example, with the time cost model, the cost of a call to *Thread.sleep()* is weighted according to a constant factor multiplied by amount of time to sleep (assuming a constant parameter). Currently the analysis assumes all loops have bound 1 and the recursion depth is set to 0. Loop bounds, if available, could be used to improve the analysis.

4.2 Dynamic symbolic execution

As the program progressed, the vast majority of the challenge programs involved some sort of peer-to-peer communication, often implemented with web server frameworks. In other words, the programs were not structured as single-threaded applications whose *main* method triggered the application logic, but rather as multi-threaded applications whose *main* method established a listening thread that ran indefinitely and implicitly invoked callbacks on separate threads that implemented the logic of the application. The original symbolic execution engine that we used, SPF, was not designed to handle this type of multi-threaded and web server based architecture. Analyzing such applications with SPF requires the manual extraction of code snippets and test drivers to symbolically execute those snippets.

We thus decided to implement a dynamic symbolic execution (DSE) engine [4] capable of analyzing multi-threaded programs that use web server frameworks and peer-to-peer communication *without requiring simplifications to those programs.*

The original design of our DSE engine used only compile-time instrumentation and worked in the following way. For every bytecode instruction in the application's classes,

additional bytecode instructions were injected; these additional bytecode instructions drive the execution of a symbolic "shadow" JVM that simultaneously executes the application symbolically. We intentionally chose not to instrument the libraries used by an application; in other words, we do not symbolically execute code implemented in the libraries used by an application. This design decision was made to increase scalability at the cost of precision.

This original design that used only compile-time instrumentation was efficient and lightweight, but ran into issues when run on complex programs. The primary reason was that it was difficult to determine when instrumented code had invoked a method implemented in uninstrumented code and vice versa; this led to errors in maintaining the symbolic shadow stack. Several factors, including virtual methods, callbacks, native methods, and static class initializers, contributed to this.

For this reason, our design evolved to include the use of a native Java Virtual Machine Tool Interface (JVMTI) agent. This native agent registers to receive a callback notification upon the occurrence of two events: method entry and method exit. This allows the agent to intervene any time a method is invoked so that it can maintain the symbolic shadow stack. Unfortunately, these two callbacks are also the most expensive callbacks that a native JVMTI agent can receive, and this led to a large reduction in performance (at least an order of magnitude for the programs we tried).

Some of the particularly relevant features of our DSE engine include:

- The capability to symbolically execute multiple threads.
- Support for symbolic arrays, including both symbolic content and symbolic array sizes.
- Constraints logged to a mongo database and solved offline.
- The capability for the analyst to specify initial constraints on the symbolic variables.
- Automatic checking for array index out of bounds exceptions.
- Automatic detection and maximization of loop bound constraints.

The last feature in the list above uses static analysis to detect which branches in a program correspond to loops and injects instructions that cause additional constraints to be generated. These additional constraints use the maximization feature of the Z3 Satisfiability Modulo Theories (SMT) solver to attempt to maximize the value of (symbolic) loop bounds. For example, a loop whose (integer) bound depended on unconstrained user input would be maximized to the maximium value of an integer, and the analyst would be notified accordingly.

We built several helper tools to compliment our DSE engine, including a GUI tool from which the analysis can be driven, such as providing input and viewing the generated path constraints.

The overhead imposed by our DSE engine depends on the ratio of instrumented to uninstrumented code that is executed; the higher this ratio, the higher the overhead. In other

words, the more code that is executed symbolically, the greater the timing overhead. On the programs on which we tested, timing overhead typically ranged between 10x - 30x.

In regard to the size of the programs that can be analyzed with our DSE engine, the only "hard" limit is the maximum method size limit imposed by the JVM itself: the total size of a method's bytecode is limited to 65536 total bytes. Because our instrumentation injects additional bytecode instructions into every method (it injects between two and seven instructions for every original bytecode instruction), the original (uninstrumented) bytecode of the program under analysis should be approximately 32kB or less. In practice, we encountered only a handful of programs which exceeded this size limit.

Running the DSE engine on the engagement programs sometimes required changes to the *structure* of the application jars (it did not require modifications to the programs); this was also the case with our Janalyzer tool. This happened with programs that used the Vaadin and Spring web server frameworks.

Additionally, packaging the library jars inside the application jar often led to errors with both the DSE and Janalyzer. Unbundling the library jars from the application jar resolved these issues.

One promising avenue for increasing the precision of our DSE engine without sacrificing scalability is the addition of *symbolic library stubs*. Recall that we intentionally designed our DSE engine so that library jars are not instrumented; unfortunately, this loss of precision does affect some analyses. For instance, the *ImageProcessor* challenge program from engagement 1 contained an algorithmic complexity vulnerability that manifested with a particular choice of pixel values in the input image. Even though the vulnerability itself was a result of user-defined code in the challenge program, a successful symbolic analysis needs library jars (in this particular case, the *java.awt.image.BufferedImage* class) to store symbolic variables.

Symbolic library stubs solve this problem by providing a simplified *symbolic* version of library calls that are used whenever the corresponding library method is invoked. They increase the precision of the analysis by allowing library methods that would not otherwise be executed symbolically to update the symbolic state of the analysis appropriately. Early in the Space-Time Analysis for Cybersecurity (STAC) program, it was mentioned that such symbolic library stubs had been researched and developed during the Intelligence Advanced Research Projects Activity (IARPA) Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) program. Unfortunately, the teams involved in the STONESOUP program were not willing to share this work. *The precision and applicability of our work would be greatly increased by the addition of symbolic library stubs.*

4.3 Worst-case analysis: SPF-WCA

To find worst-case complexity vulnerabilities, the Carnegie Mellon University (CMU) team first developed a technique based on symbolic execution[14]. The technique uses an efficient guided analysis to compute bounds on the worst-case complexity (for increasing input sizes) and to generate test values that trigger the worst-case behaviors. The resulting bounds are fitted to a function to obtain a prediction of the worst-case program behavior at any in-put sizes. Comparing these predictions to the programmers' expectations or to theoretical

asymptotic bounds can reveal vulnerabilities or confirm that a program behaves as expected. To achieve scalability we use path policies to guide the symbolic execution towards worst-case paths. The policies are learned from the worst-case results obtained with exhaustive exploration at small input sizes and are applied to guide exploration at larger input sizes, where un-guided exhaustive exploration is no longer possible. To achieve precision we use path policies that take into account the history of choices made along the path when deciding which branch to execute next in the program. Furthermore, the history computation is context-preserving, meaning that the decision for each branch depends on the history computed with respect to the enclosing method. We implemented the technique in the Symbolic PathFinder tool [20]. We showed experimentally that it can find vulnerabilities in complex Java programs and can outperform established symbolic techniques. The work was presented at the International Conference on Software Testing (ICST) 2017 conference where it won the **best paper award**.

4.4 Worst-case analysis: Canopy

The CMU team further explored heuristic analysis techniques [15] for finding costly paths in programs, where the cost refers to the execution time or memory consumed by the program. To this end, we developed Canopy, an analysis framework that can support various software engineering tasks, such as finding vulnerabilities related to denial-of-service attacks, guiding compiler optimizations or finding performance bottlenecks in software. The analysis performs sampling over symbolic program paths, which are computed with a symbolic execution over the program, and uses Monte Carlo Tree Search (MCTS) to guide the search for costly paths. We implemented the proposed method in SPF and we evaluated it on Java programs. Our experiments showed the promise of the technique for finding performance bottlenecks in software.

4.5 AFL-style fuzzing with Kelinci

Motivated by the challenging Defense Advanced Research Projects Agency (DARPA) engagements we also explored alternative analysis techniques, such as grey-box fuzzing [12]. Grey-box fuzzing is a random testing technique that has been shown to be effective at finding security vulnerabilities in software. The technique leverages program instrumentation to gather information about the program with the goal of increasing the code coverage dur-ing fuzzing, which makes grey-box fuzzers extremely efficient vulnerability detection tools. One such tool is American Fuzzy Lop (AFL), a grey-box fuzzer for C programs that has been used successfully to find security vulnerabilities and other critical defects in countless software products. We developed Kelinci, a tool that interfaces AFL with instrumented Java programs. The tool does not require modifications to AFL and is easily parallelizable. Applying AFL-type fuzzing to Java programs opens up the possibility of testing Java based applications using this powerful technique. We show the effectiveness of Kelinci by applying it on the image processing library Apache Commons Imaging, in which it identified a bug within one hour.

4.6 Badger: Integration of fuzzing with symbolic execution for worst-case analysis

We also explored hybrid approaches[18]. Hybrid testing approaches that involve fuzz testing and symbolic execution have shown promising results in achieving high code coverage, uncovering subtle errors and vulnerabilities in a variety of software applications. In [18] we describe Badger - a new hybrid approach for complexity analysis, with the goal of discovering vulnerabilities which occur when the worst-case time or space complexity of an application is significantly higher than the average case.

Badger uses fuzz testing to generate a diverse set of inputs that aim to increase not only coverage but also a resource-related cost associated with each path. Since fuzzing may fail to execute deep program paths due to its limited knowledge about the conditions that influence these paths, we complement the analysis with a symbolic execution, which is also customized to search for paths that increase the resource-related cost. Symbolic execution is particularly good at generating inputs that satisfy various program conditions but by itself suffers from path explosion. Therefore, Badger uses fuzzing and symbolic execution in tandem, to leverage their benefits and overcome their weaknesses.

We implemented our approach for the analysis of Java programs, based on Kelinci and SPF. We evaluated Badger on Java applications, showing that our approach is significantly faster in generating worst-case executions compared to fuzzing or symbolic execution on their own.

4.7 Multi-run side-channel analysis

The CMU team also did extensive work on side channel analysis based on symbolic execution and model counting [19, 5, 21, 16] . Side-channel attacks recover confidential information from non-functional characteristics of computations, such as time or memory consumption. In [19] we describe a program analysis that uses symbolic execution to quantify the information that is leaked to an attacker who makes multiple side-channel measurements. The analysis also synthesizes the concrete public inputs (the "attack") that lead to maximum leakage, via a novel reduction to Max-SMT solving over the constraints collected with symbolic execution. Furthermore model counting and information-theoretic metrics are used to compute an attacker's remaining uncertainty about a secret after a certain number of side-channel measurements are made. We have implemented the analysis in the Symbolic PathFinder tool and applied it in the context of password checking and cryptographic functions, showing how to obtain tight bounds on information leakage under a small number of attack steps.

The work was further extended in [5] with analysis for string-manipulating systems, in [21] with synthesis of adaptive side-channel attacks, and in [16] with analysis of probabilistic systems.

4.8 Attack synthesis

The University of California at Santa Barbara (UCSB) team has developed an automated technique for online adaptive attack synthesis which can be used to extract information

from program functions that leak secret data through a side channel. In this approach we synthesize attack steps dynamically and consider noisy program environments. Our approach consists of an offline profiling phase using symbolic execution, witness generation, and profiling to construct a noise model. During our online attack synthesis phase, we use weighted model counting and numeric optimization to automatically synthesize attack inputs. We experimentally evaluate the effectiveness of our approach on DARPA STAC problems and demonstrated that this approach can be effective in identifying exploits for side-channel vulnerabilities.

We extended this approach later on and integrated meta-heuristics for generation of optimum attacks. We use symbolic execution to extract path constraints, automata-based model counting to estimate the probability of execution paths, and meta-heuristic methods to maximize information gain based on entropy for synthesizing adaptive attack steps.

4.9 Co-Co-Channel: Static side-channel analysis

We developed a new technique for scalable detection of side-channels in software. Given a program and a cost model for a side-channel (such as time or memory usage), we de-compose the control flow graph of the program into nested branch and loop components, and compositionally assign a symbolic cost expression to each component. Symbolic cost expressions provide an over-approximation of all possible observable cost values that components can generate. Queries to a satisfiability solver on the difference between possible cost values of a component allow us to detect the presence of imbalanced paths (with respect to observable cost) through the control flow graph. When combined with taint analysis that identifies conditional statements that depend on secret information, our technique answers the following question: Does there exist a pair of paths in the program's control flow graph, differing only on branch conditions influenced by the secret, that differ in observable side-channel value by more than some given threshold? Additional optimization queries allow us to identify the minimal number of loop iterations necessary for the above to hold or the maximal cost difference between paths in the graph. We perform symbolic execution based feasibility analyses to eliminate control flow paths that are infeasible. We implemented our techniques in a prototype tool called COmpositional COnstraint-based side-CHANNEL analyzer (CoCo-Channel) for analyzing Java programs. Our experiments demonstrate its favourable performance against state-of-the-art tools as well as its effectiveness and scalability on a set of sizable, realistic Java server-client and peer-to-peer applications.

4.10 Profit: Dynamic side-channel analysis

We developed a black-box, dynamic technique to detect and quantify side-channel information leaks in networked applications that communicate through a Transport Layer Security (TLS)-encrypted stream. Given a user-supplied profiling-input suite in which some aspect of the inputs is marked as secret, we run the application over the inputs and capture a collection of variable-length network packet traces. The captured traces give rise to a vast side-channel feature space, including the size and timestamp of each individual packet as well as their aggregations (such as total time, median size, etc.) over every possible subset

of packets. Finding the features that leak the most information is a difficult problem.

Our approach addresses this problem in three steps: 1) Global analysis of traces for their alignment and identification of emphphases across traces; 2) Feature extraction using the identified phases; 3) Information leakage quantification and ranking of features via estimation of probability distribution.

We implemented this approach in a tool called Profit and experimentally evaluated it on DARPA STAC problems. Our experimental results demonstrate that, given suitable profiling-input suites, Profit is successful in automatically detecting information-leaking features in applications, and correctly ordering the strength of the leakage for differently-leaking variants of the same application.

4.11 JIT induced side-channels

Side-channel vulnerabilities in software are caused by an observable imbalance in resource usage across different program paths. We showed that just-in-time (JIT) compilation, which is crucial to the runtime performance of modern interpreted languages, can introduce timing side channels in cases where the input distribution to the program is non-uniform. Such timing channels can enable an attacker to infer potentially sensitive information about predicates on the program input. We defined three attack models under which such side channels are harnessable and five vulnerability templates to detect susceptible code fragments and predicates. We also developed profiling algorithms to generate the representative statistical information necessary for the attacker to perform accurate inference. We systematically evaluated the strength of these JIT-based side channels on the java.lang.String, java.lang.Math, and java.math.BigInteger classes from the Java standard library, and on the JavaScript built-in objects String, Math, and Array. We carried out our evaluation using two widely adopted, open-source, JIT-enhanced runtime engines for the Java and JavaScript languages: the Oracle HotSpot Java Virtual Machine and the Google V8 JavaScript engine, respectively. Finally, we demonstrated a few examples of JIT-based side channels in the Apache Shiro security framework and the GraphHopper route planning server, and showed that they are observable over the public Internet.

4.12 ABC: A model counting constraint solver

In recent years constraint solvers have become essential components of program analysis techniques for detecting vulnerabilities and bugs in programs. For quantitative program analyses, such as quantifying the information leakage, checking the satisfiability of a constraint is not sufficient, and it is necessary to count the number of solutions. We developed a constraint solver that, given a constraint, (1) constructs an automaton that accepts all solutions that satisfy the constraint, (2) generates a function that, given a length bound, gives the total number of solutions within that bound. Our approach relies on the observation that, using an automata-based constraint representation, model counting reduces to path counting, which can be solved precisely. We implemented this approach for string constraints in a tool called Automata Based model Counter (ABC).

Next, we extended ABC in order to handle both string and numeric constraints. In this

extension of ABC, we first construct a multi-track deterministic finite state automaton that accepts all solutions to the given constraint. A multi-track automaton is a generalization of finite state automaton, where a multi-track automaton accepts a string tuple (instead of a single string) by reading multiple input tracks simultaneously. Since automata can only represent regular sets, we limit the numeric constraints to linear integer arithmetic, and for non-regular string constraints we over-approximate the solution set. Counting the number of accepting paths in the generated automaton solves the model counting problem. Our approach is parameterized in the sense that, we do not assume a finite domain size during automata construction, resulting in a potentially infinite set of solutions, and our model counting approach works for arbitrarily large bounds. We experimentally demonstrated the effectiveness of our approach on a large set of string and numeric constraints extracted from software applications. We experimentally compared ABC to five existing model counting constraint solvers for string and numeric constraints and demonstrated that ABC is as efficient and as or more precise than other solvers. Moreover, ABC can handle mixed constraints with string and integer variables that no other tool can.

Efficiency of constraint solvers is crucial for efficiency of modern program analysis techniques. In order to improve the efficiency of ABC, we developed a constraint caching framework to expedite potentially expensive satisfiability and model-counting queries. Integral to this framework is a novel constraint normalization procedure that we developed under which the cardinality of the solution set of a constraint, but not necessarily the solution set itself, is preserved. We extended these constraint normalization techniques to string constraints in order to support analysis of string-manipulating code. We used a group-theoretic framework to formalize our normalization techniques, which generalizes earlier results on constraint normalization. We also developed a parameterized caching approach where, in addition to storing the result of a model-counting query, we also store a model-counter object in the constraint store that allows us to efficiently recount the number of satisfying models for different maximum bounds. We implemented our caching framework in our tool Cashew, which is built as an extension of the Green caching framework, and integrated it with SPF and our model-counting constraint solver ABC. Our experiments show that constraint caching can significantly improve the performance of symbolic and quantitative program analyses. For instance, Cashew can normalize the 10,104 unique constraints in the SMC/Kaluza benchmark down to 394 normal forms, achieve a 10x speedup on the SMC/Kaluza-Big dataset, and an average 3x speedup in our SPF-based side-channel analysis experiments.

4.13 Research on model counting

Model counting is of central importance in quantitative reasoning about systems [7]. Examples include computing the probability that a system successfully accomplishes its task without errors, and measuring the number of bits leaked by a system to an adversary in Shannon entropy. Most previous work in those areas demonstrated their analysis on programs with linear constraints, in which cases model counting is polynomial time. Model counting for nonlinear constraints is notoriously hard, and thus programs with nonlinear constraints are not well-studied. In [7] we surveyed state-of-the-art techniques and tools for

model counting with respect to SMT constraints, modulo the bitvector theory, since this theory is decidable, and it can express nonlinear constraints that arise from the analysis of computer programs. We integrate these techniques within the Symbolic Pathfinder platform and evaluate them on difficult nonlinear constraints generated from the analysis of cryptographic functions.

4.14 DifFuzz: Differential Fuzzing for Side-Channel Analysis

In more recent work, we developed DifFuzz [17], a fuzzing-based approach for detecting side-channel vulnerabilities related to time and space. DifFuzz automatically detects these vulnerabilities by analyzing two versions of the program and using resource-guided heuristics to find inputs that maximize the difference in resource consumption between secret-dependent paths. The methodology of DifFuzz is general and can be applied to programs written in any language. We developed an implementation that targets analysis of Java programs, and uses and extends the Kelinci and AFL fuzzers. We evaluate DifFuzz on a large number of Java programs, coming from STAC engagements and real-world applications, and demonstrate that it can reveal unknown side-channel vulnerabilities in popular applications. We also show that DifFuzz compares favorably against Blazer and Themis, two state-of-the-art analysis tools for finding side-channels in Java programs.

5 CONCLUSIONS

This section summarizes some of our important findings during this project.

5.1 Lessons from dynamic symbolic execution

While we believe that dynamic symbolic execution is an important and powerful analysis technique capable of identifying many types of interesting program inputs, applying DSE to the provided challenge programs yielded relatively few actionable insights relevant to the challenge questions. One reason is that, as a whole, the threat-model for a complexity vulnerability is extremely broad and difficult to codify. The complexity vulnerabilities in the provided challenge programs manifested in a variety of ways; for example, sometimes they manifested with a particular choice of "magic" input values, and other times they manifested with a particularly large input. At its core, symbolic execution is a search procedure that relies on heuristics to prune the exponential search space and identify "promising" inputs that lead to a vulnerability. Without a more narrow threat-model, developing scalable and precise heuristics that can be used to identify complexity vulnerabilities on STAC-sized engagement programs requires further research.

5.2 Lessons from fuzzing

In this project, we have started by using symbolic execution and model counting for both worst case execution and side-channel analysis. We developed various heuristics to scale the symbolic execution to large programs, as outlined in the sections above. However, the nature of the engagements pushed us towards exploring alternative techniques, such as fuzzing, towards finding quickly reported vulnerabilities. We have thus developed Kelinci, an AFL-based fuzzer for Java, and Kelinci-WCA, which extends Kelinci with cost-guided heuristcs.

We have further developed techniques for both worst-case and side-channel analysis, that are based on fuzzing (see above).

The advantage of fuzzing over symbolic execution is that it can be applied more or less "out of the box", without worrying too much about the underlying libraries that would need to be modeled for symbolic execution. Fuzzing also has an advantage over static analysis, because it always reports vulnerabilities that are real, whereas static analysis can give warnings that turn out to be infeasible (that this may be a nuisance to the users of the analysis tool).

Furthermore, if run long enough, fuzzing eventually finds the target vulnerabilities, especially if they are "shallow", meaning that they are not guarded by some complicated conditions in the code (which seemed to be the case in the engagements). If the latter is the case, fuzzing typically has trouble finding inputs that satisfy those complex conditions, but symbolic execution is typically good at that. We are therefore believing that the right direction for future research is to combine the two techniques, to benefit from their strengths while at the same time to overcome their weaknesses. This was the motivation behind our work on Badger, a hybrid fuzzing/symbolic execution approach for worst case analysis. Recently we have also developed HyDiff (under submission), which is a hybrid fuzzing/symbolic execution approach that targets differential analysis. The latter can be used for e.g. finding side channels.

Another drawback of AFL-style fuzzing is its rather poor handling of programs that accept inputs that have complex structure, for instance inputs that are specified by a grammar.

Typically, fuzzers like AFL take a set of seed inputs and leverage random mutations to continually improve the inputs with respect to a *cost*, e.g. program code coverage, to discover vulnerabilities or bugs. Following this methodology, fuzzers are very good at generating *unstructured* inputs that achieve high coverage. However fuzzers are less effective when the inputs are structured, say they conform to an input grammar. Due to the nature of random mutations, the overwhelming abundance of inputs generated by this common fuzzing practice often adversely hinders the effectiveness and efficiency of fuzzers on grammar-aware applications. The problem of testing becomes even harder, when the goal is to not only achieve increased code coverage, but also to find complex vulnerabilities related to other cost measures, say high resource consumption in an application.

To address this shortcoming of fuzzing we are currently investigating an adaptive grammar-based fuzzing approach to effectively and efficiently generate inputs that expose costly executions in programs. The approach takes as input a user-provided grammar, which describes the input space of the program under analysis, and uses it to generate test inputs. The approach assumes that the grammar description is *approximate* since precisely describing the input program space is often difficult as a program may accept unintended inputs due to e.g., errors in parsing (see for example one of the calculator examples from the engagements). Yet these inputs may reveal worst-case complexity vulnerabilities. The novelty of our approach is then twofold: (1) Given the user-provided grammar, our approach attempts to discover whether the program accepts unexpected inputs outside of the provided grammar, and if so,

it repairs the grammar via grammar mutations. The repaired grammar serves as a specification of the actual inputs accepted by the application. (2) Based on the refined grammar, it generates concrete test inputs. It starts by treating every production rule in the grammar with equal probability of being used for generating concrete inputs. It then adaptively refines the probabilities along the way by increasing the probabilities for rules that have been used to generate inputs that improve a cost, e.g., code coverage or arbitrary used-defined cost. Evaluation results show that our approach significantly outperforms state-of-the-art baselines, such as grammar based random fuzzing or UC Berkeley's PerfFuzz.

5.3 Implications for Further Research

The project has tried to address a very difficult problem: finding space-time complexity and side channel vulnerabilities in large code bases. We summarize our suggestions for further research below.

The biggest problem our team has encountered was scaling: tools that worked well on small examples (∼1K source lines of code (SLOC)) did not scale well to large ones (∼10-100K SLOC or above). Clearly, there is a need for scaling up the tools. From the approaches we pursued, dynamic symbolic analysis (DSE), fuzzing coupled with symbolic execution (Kelinci, Badger), and black-box statistical analysis (Profit) seem to be the most promising.

The second major problem was related to libraries. Today's large software application packages are built on layers and layers of libraries. Unless these libraries are analyzed and somehow modeled the approaches will not scale. The goal is to find issues within the applications, not in the libraries but unless the library is abstract with a model the analysis will not scale. Clearly there is a need for finding useful models for library services.

The third major problem was the need for human insight. In many cases the tools were useful in (1) pre-processing the code base to be analyzed for a human analyst, and in (2) testing hypotheses produced by a human analyst. Such a semi-automated process appears to be the most productive as fully automatic analysis does not seem feasible. There is clearly a need for such human-machine collaboration where the result is bigger than what either participant can achieve alone.

6 REFERENCES

[1] Karim Ali and Ondrej Lhoták. Averroes: Whole-Program Analysis without the Whole Program. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, pages 378–400, 2013.

[2] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 1083–1094, New York, NY, USA, 2014. ACM.

[3] Daniel Balasubramanian, Dmitriy Kostyuchenko, Kasper Søe Luckow, Rody Kersten, and Gabor Karsai. A cloud-based execution framework for program analysis. In *Software*

Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings, pages 139–154, 2018.

[4] Daniel Balasubramanian, Zhenkai Zhang, Dan McDermet, and Gabor Karsai. Dynamic symbolic execution for the analysis of web server applications in java. In *Symposium on Applied Computing, Software Verification and Testing - 34th ACM/SIGAPP Symposium on Applied Computing, SAC SVT 2018, Cyprus, April 8 - 12, 2019*, 2019.

[5] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 193–204, 2016.

[6] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 241–250, 2011.

[7] Mateus Borges, Quoc-Sang Phan, Antonio Filieri, and Corina S. Pasareanu. Model-counting approaches for nonlinear numerical constraints. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, pages 131–138, 2017.

[8] Ricardo Corin and Felipe Andrés Manzano. Taint analysis of security code in the klee symbolic execution engine. In *Proceedings of the 14th International Conference on Information and Communications Security*, ICICS'12, pages 264–275, Berlin, Heidelberg, 2012. Springer-Verlag.

[9] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 431–446, 2013.

[10] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

[11] IBM. T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`.

[12] Rody Kersten, Kasper Søe Luckow, and Corina S. Pasareanu. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2511–2513, 2017.

[13] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[14] Kasper Søe Luckow, Rody Kersten, and Corina S. Pasareanu. Symbolic complexity analysis using context-preserving histories. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 58–68, 2017.

[15] Kasper Søe Luckow, Corina S. Pasareanu, and Willem Visser. Monte carlo tree search for finding costly paths in programs. In *Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, pages 123–138, 2018.

[16] Pasquale Malacaria, M. H. R. Khouzani, Corina S. Pasareanu, Quoc-Sang Phan, and Kasper Søe Luckow. Symbolic side-channel analysis for probabilistic programs. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 313–327, 2018.

[17] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. Diffuzz: Differential fuzzing for side-channel analysis. *CoRR (to appear at ICSE'19)*, abs/1811.07005, 2018.

[18] Yannic Noller, Rody Kersten, and Corina S. Pasareanu. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 322–332, 2018.

[19] Corina S. Pasareanu, Quoc-Sang Phan, and Pasquale Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 387–400, 2016.

[20] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[21] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 328–342, 2017.

[22] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pages 494–505, 2007.

Appendix A Publications and Presentations

CMU Publications

1. Shirin Nilizadeh, Yannic Noller, Corina S. Pasareanu: DifFuzz: Differential Fuzzing for Side-Channel Analysis. ICSE 2019 (to appear)

2. Xuan Bach D. Le, Lingfeng Bao , David Lo, Xin Xia, Shanping Li, Corina S Pasareanu: On Reliability of Patch Correctness Assessment. ICSE 2019 (to appear)

3. Pasquale Malacaria, M. H. R. Khouzani, Corina S. Pasareanu, Quoc-Sang Phan, Kasper Se Luckow: Symbolic Side-Channel Analysis for Probabilistic Programs. CSF 2018: 313-327

4. Tegan Brennan, Seemanta Saha, Tevfik Bultan, Corina S. Pasareanu: Symbolic path cost analysis for side-channel detection. ISSTA 2018: 27-37

5. Yannic Noller, Rody Kersten, Corina S. Pasareanu: Badger: complexity analysis with fuzzing and symbolic execution. ISSTA 2018: 322-332

6. Kasper Se Luckow, Corina S. Pasareanu, Willem Visser: Monte Carlo Tree Search for Finding Costly Paths in Programs. SEFM 2018: 123-138

7. Rody Kersten, Kasper Se Luckow, Corina S. Pasareanu: POSTER: AFL-based Fuzzing for Java with Kelinci. ACM Conference on Computer and Communications Security 2017: 2511-2513

8. Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, Tevfik Bultan: Synthesis of Adaptive Side-Channel Attacks. CSF 2017: 328-342

9. Kasper Se Luckow, Rody Kersten, Corina S. Pasareanu: Symbolic Complexity Analysis Using Context-Preserving Histories. ICST 2017: 58-68

10. Mateus Borges, Quoc-Sang Phan, Antonio Filieri, Corina S. Pasareanu: Model-Counting Approaches for Nonlinear Numerical Constraints. NFM 2017: 131-138

11. Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, Tevfik Bultan: String analysis for side channels with segmented oracles. SIGSOFT FSE 2016: 193-204

12. Corina S. Pasareanu, Quoc-Sang Phan, Pasquale Malacaria: Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT. CSF 2016: 387-400

UCSB Publications

Book

1. Tevfik Bultan, Fang Yu, Muath Alkhalaf, Abdulbaki Aydin: String Analysis for Software Verification and Security. Springer 2017, ISBN 978-3-319-68668-4.

Refereed full papers

1. Nicolas, Rosner, Burak Kadron, Lucas Bang, and Tevfik Bultan. Detecting and Quantifying Side Channels in Networked Applications. Proceedings of the 26th Network and Distributed System Security Symposium (NDSS 2019), February 24-27, 2019.

2. Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. Parameterized Model Counting for String and Numeric Constraints, Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018), pp. 400410, Lake Buena Vista, Florida, November 4-9, 2018.

3. Nestan Tsiskaridze, Lucas Bang, Joseph McMahan, Tevfik Bultan, and Timothy Sherwood. Information Leakage in Arbiter Protocols, Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA 2018), pp. 404421, October 7-10, 2018, Los Angeles, CA.

4. Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu. Symbolic path cost analysis for side-channel detection, Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018), pp. 2737, Amsterdam, The Netherlands, July 16-21, 2018.

5. Lucas Bang, Nicolas Rosner, and Tevfik Bultan. Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations, Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroSP 2018), pp. 307322, London, United Kingdom, April 24-26, 2018.

6. Tegan Brennan, Nestan Tsiskaridze, Nicolas Rosner, Abdulbaki Aydin and Tevfik Bultan. Constraint Normalization and Parameterized Caching for Quantitative Program Analysis. Proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2017), pp. 535546, Paderborn, Germany, September 4-8, 2017.

7. Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of Adaptive Side-Channel Attacks. Proceedings of the 2017 IEEE Computer Security Foundations Symposium (CSF 2017), pp. 328342, Santa Barbara, CA, USA, August 21-25, 2017.

8. Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. String Analysis for Side Channels with Segmented Oracles. Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2016), pp. 193204, Seattle, WA, USA, November 13-18, 2016.

9. Lucas Bang, Abdulbaki Aydin, and Tevfik Bultan. Automatically Computing Path Complexity of Programs. Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015), pp. 6172, Bergamo, Italy, August 30September 4, 2015.

10. Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. Automata-based model counting for string constraints. Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015), pp. 255272, San Francisco, CA, USA, July 1824, 2015.

Refereed short/workshop papers

1. Seemanta Saha, Ismet Burak Kadron, William Eiers, Lucas Bang, and Tevfik Bultan. Attack Synthesis for Strings using Meta-Heuristics. Java PathFinder Workshop (JPF 2018). Lake Buena Vista, Florida, November 5, 2018.

2. Tegan Brennan, Seemanta Saha, and Tevfik Bultan. Symbolic path cost analysis for side-channel detection. Poster paper. Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE 2018), pp. 424-425, Gothenburg, Sweden, May 27 - June 03, 2018.

3. Tegan Brennan. Path cost analysis for side channel detection. Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017), pp. 416419, Santa Barbara, CA, USA, July 10 - 14, 2017.

Invited papers/abstracts

1. Tevfik Bultan. Side-Channel Analysis via Symbolic Execution and Model Counting. ACM SIGSOFT Software Engineering Notes 43(4): 55 (2018).

2. Tevfik Bultan. Side Channel Analysis Using a Model Counting Constraint Solver and Symbolic Execution. Proceedings of the 36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2016), 6:1-6:2, Chennai, India, December 13-15, 2016.

3. Tevfik Bultan. String Analysis for Vulnerability Detection and Repair. Proceedings of the 22nd International Symposium on Model Checking Software (SPIN 2015), pp. 39, Stellenbosch, South Africa, August 24-26, 2015.

Vanderbilt Publications

Refereed full papers

1. Daniel Balasubramanian, Dmitriy Kostyuchenko, Kasper Se Luckow, Rody Kersten, Gabor Karsai: A Cloud-Based Execution Framework for Program Analysis. International Conference on Software Engineering and Formal Methods (SEFM) 2018: 139-154

2. Daniel Balasubramanian, Zhenkai Zhang, Dan McDermet, Gabor Karsai: Dynamic Symbolic Execution for the Analysis of Web Server Applications in Java. ACM Symposium on Applied Computing, Software Verification and Testing (ACM SAC SVT) 2019.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

AFL	American Fuzzy Lop
ABC	Automata Based model Counter
CMU	Carnegie Mellon University
CoCo-Channel	COmpositional COnstraint-based side-CHANNEL analyzer
DARPA	Defense Advanced Research Projects Agency
IARPA	Intelligence Advanced Research Projects Activity
ICST	International Conference on Software Testing
JPF	Java PathFinder
JVM	Java Virtual Machine
MCS	Model-Counting Constraint Solver
MCTS	Monte Carlo Tree Search
NASA	National Aeronautics and Space Administration
SMT	Satisfiability Modulo Theories
STONESOUP	Securely Taking On New Executable Software of Uncertain Provenance
SCA	Side-channel Analysis
STAC	Space-Time Analysis for Cybersecurity
SPF-WCA	Symbolic PathFinder-Worst Case Analysis
TLS	Transport Layer Security
UCSB	University of California at Santa Barbara
WALA	Watson Libraries for Analysis
WCA	Worst-Case Analysis
CDG	control-dependency graph
DSE	dynamic symbolic execution
GUI	graphical user interface
JIT	just-in-time
SLOC	source lines of code