AFRL-RI-RS-TR-2021-041

SPACE/TIME ANALYSIS FOR CYBERSECURITY (STAC)

UNIVERSITY OF UTAH

MARCH 2021

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-2021-041 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 2021	FINAL TECHNICAL REPORT	APR 2015 – JUL 2020

4. TITLE AND SUBTITLE

SPACE/TIME ANALYSIS FOR CYBERSECURITY (STAC)

5a. CONTRACT NUMBER
N/A

5b. GRANT NUMBER
FA8750-15-2-0092

5c. PROGRAM ELEMENT NUMBER
61102E

6. AUTHOR(S)

Michael D. Adams

5d. PROJECT NUMBER
STAC

5e. TASK NUMBER
UT

5f. WORK UNIT NUMBER
AH

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Utah
50 S. Central Campus Drive Room 3190
Salt Lake City UT 84112

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-2021-041

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 Utah/Irvine team's approach to the STAC program was focused on statically analyzing JVM byte code. It was one of the only teams that took a pure static analysis approach. This approach uncovered limitations of the state of the art in static analysis, and the Utah/Irvine team developed multiple foundational techniques to address these limitations. However, more foundational work is required before static analysis can reach its potential.

15. SUBJECT TERMS

Algorithmic complexity vulnerabilities, static analysis, dynamic analysis, vulnerability detection, side-channel attacks

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	17	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

Section		Page
1.0	SUMMARY……...…….	1
2.0	INTRODUCTION……...	2
3.0	METHODS ASSUMPTIONS AND PROCEDURES……...	3
4.0	RESULTS AND DISCUSSIONS..…….	4
4.1	Jaam: Static Analysis for Java Bytecode……..……..	4
4.2	Jade: A Java Decompiler…….……	6
5.0	MOST SIGNIFICANT PAPERS..…….	9
5.1	Push-down for Free (P4F)……..	8
5.2	Allocation Characterizes Polyvariance (ACP)……	8
5.3	Push-down for Free (P4F)……..	9
6.0	CONCLUSIONS…….	11
7.0	REFERENCES……..…….	12
LIST OF ACRONYMS…..…….		13

1.0 SUMMARY

The Space/Time Analysis for Cybersecurity (STAC) program aimed to develop techniques for detecting space and/or time side-channel vulnerabilities in compiled Java virtual-machine (JVM) bytecode programs. The Utah/Irvine team's approach to the STAC program was focused on statically analyzing JVM bytecode. It was one of the only teams that took a pure static analysis approach. This approach uncovered limitations of the state of the art in static analysis, and the Utah/Irvine team developed multiple foundational techniques to address these limitations. However, more foundational work is required before static analysis can reach its potential.

2.0 INTRODUCTION

Our project had two main efforts. The first was the development of the “Jaam” static analysis tool, the second was a contract extension to implement a well-tested Java decompiler, “Jade”. The code for Jaam is available at <https://github.com/ucombinator/jaam>, while the code for Jade is available at Jade is available at <https://github/ucombinator/jade>.

Jaam is a suite of static-analysis tools to help analysts determine whether a compiled JVM bytecode application contains vulnerabilities. In particular it aids with detecting algorithmic-complexity and side-channel vulnerabilities relating to resource usage (i.e., space) or running time (i.e., time). It does this via a number of tools described later in this document.

During the course of the project we found decompiling the JVM bytecode to be useful when analyzing an application for vulnerabilities. From time to time, it was also helpful to modify the decompiled source code and then compile it producing a modified version of the application being analyzed. In particular this made it easier to test hypotheses that analysts about the behavior of the application and potential vulnerabilities in the application. This required decompilers to produce decompiled code that was not just human readable but also compilable by standard Java compilation tools. To our disappointment and surprise, we found that no publicly available decompilers reliably produces compilable Java code. Thus, we applied for an extension to the contract to develop such a decompiler. This became the Jade decompiler discussed later in this document.

3.0 METHODS ASSUMPTIONS AND PROCEDURES

The project was organized around “engagements” that happened every six months. These were DARPA organized events in which we were provided with several JVM applications in the form of compiled JVM bytecode and a series of questions about whether certain types of vulnerabilities were present in that code. Our team focused on the use of static analysis to answer these questions. The results of these engagements were used to guide further development of the tools in preparation of the next engagement.

4.0 RESULTS AND DISCUSSIONS

4.1 Jaam: Static Analysis for Java Bytecode

With the conclusion of the main STAC program, development has ended. The code for this project is available at <https://github.com/ucombinator/jaam>. With that tool, running 'createJaam' on a challenge apps directory runs all the analyses and records them in a file that can be loaded into the visualizer. This integrates a control-flow analysis, a data-flow analysis, a decompiler, and a loop classifier.

Additional information on building and running the tool is available in the README.md file at < https://github.com/ucombinator/jaam/README.md>

Our approach to this program focused on statically analyzing programs unlike other teams that used dynamic analysis or fuzzing. This presented us with a unique set of issues.

Early in the project we had issues with ensuring the analysis covered the entire program. This is because the analysis handled only code that could be directly reached from the program entry point. However, as the programs being analyzed were for JVM, most of the code in the programs is reached only from a callback.

This lead to what we called the "library problem". Namely, if our analysis skipped over library code, it would miss the callbacks, but if our analysis went into library code it would get lost in that code and never come back. This was due to the large size and interconnectedness of these libraries.

This became a problem that we grappled with throughout the program. We developed a number of foundational techniques (e.g., Allocation Characterizes Polyvariance (ACP), Push-down for Free (P4F), and Demand Control-flow Analysis (DCA). See Section 5.0 Most-Significant-Papers) that can help with this problem, but they did not entirely solve the problem.

The one technique that did work well was using a "class-hierarchy analysis" (CHA). This type of analysis is able start in the middle of a program. This allows us to analyze callbacks even if we do not analyze library code that calls those callbacks. However, CHA is a low precision analysis and does not provide as much information about the code as the other analyses.

It may be possible to hybridize CHA with DCA so that more precise analyses are done on-demand where needed. However, as DCA was developed late in the project, and thus we were unable to explore this and more research into this is necessary.

Based on the CHA analysis, we developed a visualization that showed the hierarchy of what methods called what other methods. Selecting a particular method would then show the intra-prodecural data flow of that method.

To the method-call hierarchy, we added nodes for the various loops in the program. This made it easy to see what functions contained or were called from loops whether directly or indirectly.

We further developed a loop classifier that color coded loop nodes according to their type. This classification allowed us to determine precisely what variables controlled the number of iterations taken by the loop. This classifier handled a number of standard types of loops (e.g., for-each loops, count-up loops, etc.) as well as a generic loop analysis that can handle any loop not already classified. While this generic loop analyzer does not report what type a loop is, it reports what variables control the loop.

Based on these loop classifiers we modified the data-flow visualization to display the values controlling the number of iterations in specific loops.

Finally, it is worth noting that throughout the project there were numerous changes and tweaks to streamline how the data-flow information is visualized to reduce the amount of information that the user has to sift through when exploring an application. Performance optimizations were implemented to improve interactive usability, and numerous small changes and tweaks were made in order to improve usability and present the data-flow information in a more comprehensible manner.

4.2 Jade: A Java Decompiler

With the conclusion of the main STAC program, development has ended. The code for this project is available at <https://github.com/ucombinator/jade>. Information on building and running the tools is available in the README.md file at <https://github.com/ucombinator/jade/README.md>

While there are a large number of JVM-to-Java decompilers available today, we have found that they are unreliable and often produce Java code that does not compile properly. We first discovered this during the DARPA STACSpace/Time Analysis for Cybersecurity engagements when none of the existing Java decompilers could be relied upon to produce valid Java code. Given the mature state of most of the Java ecosystem, we found this fact surprising. So we tested several decompilers against widely used Java applications and libraries. We tested JDCore, JadX, Procyon, Fernflower, and Jad. We tested them against several widely used Java applications and libraries. These include Junit, Apache Commons Lang, Spring Core, Apache Hadoop Common, and jEdit. All of the decompilers that we tested failed to reliably produce compilable Java code for the applications we tested. Note that these tests include the standard decompilers used by the IDEs IntelliJ (i.e., Fernflower) and Eclipse (i.e., JDCore, Procyon, FernFlower, or Jad depending on settings).

Note that unlike a lower-level assembly like x86, JVM bytecode is designed to be easy to analyze. It is fully annotated with the types and structures of all values, and methods are explicitly marked with a complete method signature. Thus, decompiling JVM bytecode should be comparatively easy. The fact that these decompilers failed to produce valid Java code is both surprising and disappointing.

A manual inspection of the decompiled code shows that the problems are not fundamental to decompilation. Indeed, the results are generally reasonable Java source code. The problems are minor errors that a human reading the code might overlook but cause errors when passed to a Java compiler. For example, we found missing local variable declarations, missing or incorrectly added casts, and even statements that the Java compiler flags as unreachable (which in Java is a compilation error).

After examining the types of mistakes made by these decompilers, we believe this happened due to two causes. First, the designers of those decompilers may have focused on how the code looks to a human rather than what happens when that code is passed to a Java compiler. Second, they insufficiently tested what happens when the decompiled code is passed to a Java compiler. In the decompiler that we propose to develop, avoiding these mistakes is a primary goal.

Thus we developed a JVM-to-Java decompiler that has as its primary goal the generation of valid, compilable Java. It was designed from the ground-up to to ensure the correctness of the Java it produces. Throughout development, we tested the decompiler against existing corpora of Java bytecode. For example, the Maven repository https://mvnrepository.com/ hosts over 8 million artifacts that we could test against.

The architecture for our decompiler and test suite consisted of multiple parts. The core decompilation consists of class structure decompilation, basic method decompilation and advanced method decompilation. Around this we wrap a testing system that, first, takes bytecode from the Maven repository then passes it through the decompiler to produce Java code. Then it passes the Java code though the javac compiler and compares the result for equivalence against the original bytecode.

This architecture was designed so that during development we could do continuous integration and testing. This allowed us to monitor the progress of the tool and quickly detect unexpected failures. By the end of the project, we expected to be able to correctly decompile all Java projects in the Maven repository. This would give us a high confidence in the correctness of the resulting decompiler.

In order to do the actual decompilation, we split the decompilation into three parts: class structure decompilation, basic method decompilation, and advanced method decompilation.

With regard to class structure, JVM bytecode includes complete information about each class, interface or enum. This includes the types and signatures of all fields and methods. Thus decompiling this part of the code into Java is relatively straight forward. This leaves only the methods bodies to be decompile, which we leave as their own separate decompilation step.

We further split method decompilation into basic decompilation and advanced decompilation. The goal of basic method decompilation is to do the simplest possible decompilation. This is done so that we have a decompilation that is correct even if not necessarily the most idiomatic. For example, it may decompile what was originally a for loop into an equivalent while loop. Since while loops are such a general constructs, decompiling all loops to them is relatively easy and thus less likely to fail.

The goal of advanced method decompilation is to produce more idiomatic Java code than is produced by basic method decompilation. Advanced decompilation will contain multiple separate transformations for different Java idioms. For example, one transformation could (when possible) decompile into for loops instead of while loops. These transformations each have a higher risk of failing than basic decompilation. However, by separating advanced decompilation from basic decompilation, we can use basic decompilation as a backup for when a particular

advanced transformation fails, and thus maintain a high assurance of the decompiler as a whole producing correct Java code.

The progress on this project proceeded well and many of the parts of this decompiler were in development, when funding was cut due to the principle investigator moving to a new institution and trouble with moving that contract the institution. Thus the project is in an incomplete state, but initial results were positive. Thus it remains a project worth funding and developing.

5.0 MOST SIGNIFICANT PAPERS

5.1 Push-down Control-flow Analysis for Free (P4F)

Traditional control-flow analysis (CFA) for higher-order languages introduces spurious connections between callers and callees, and different invocations of a function may pollute each other's return flows. Recently, three distinct approaches have been published that provide perfect call-stack precision in a computable manner: Context-Free Approach to Control-Flow Analysis (CFA2), Push-Down Control-Flow Analysis (PDCFA), and Allocating Abstract Control (AAC). Unfortunately, implementing CFA2 and PDCFA requires significant engineering effort. Furthermore, all three are computationally expensive. For a monovariant analysis, CFA2 is in O(2^n), PDCFA is in O(n^6), and AAC is in O(n^8).

In this paper, we describe a new technique that builds on these but is both straightforward to implement and computationally inexpensive. The crucial insight is an unusual state-dependent allocation strategy for the addresses of continuations. Our technique imposes only a constant-factor overhead on the underlying analysis and costs only O(n^3) in the monovariant case. We present the intuitions behind this development, benchmarks demonstrating its efficacy, and a proof of the precision of this analysis.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In Proceedings of the 43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16. ACM, New York, NY, USA, January 2016. doi: 10.1145/2837614.2837631.

5.2 Allocation Characterizes Polyvariance (ACP)

The polyvariance of a static analysis is the degree to which it structurally differentiates approximations of program values. Polyvariant techniques come in a number of different flavors that represent alternative heuristics for managing the trade-off an analysis strikes between precision and complexity. For example, call sensitivity supposes that values will tend to correlate with recent call sites, object sensitivity supposes that values will correlate with the allocation points of related objects, the Cartesian product algorithm supposes correlations between the values of arguments to the same function, and so forth.

In this paper, we describe a unified methodology for implementing and understanding polyvariance in a higher-order setting (i.e., for control-flow analyses). We do this by extending the method of abstracting abstract machines (AAM), a systematic approach to producing an abstract interpretation of abstract-machine semantics. AAM eliminates recursion within a language's semantics by passing around an explicit store, and thus places importance on the

strategy an analysis uses for allocating abstract addresses within the abstract heap or store. We build on AAM by showing that the design space of possible abstract allocators exactly and uniquely corresponds to the design space of polyvariant strategies. This allows us to both unify and generalize polyvariance as tunings of a single function. Changes to the behavior of this function easily recapitulate classic styles of analysis and produce novel variations, combinations of techniques, and fundamentally new techniques.

Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP '16, pages 407–420. ACM, New York, NY, USA, September 2016. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951936.

5.3 Demand Control-flow Analysis (DCA)

Points-to analysis manifests in a functional setting as control-flow analysis. Despite the ubiquity of demand points-to analyses, there are no analogous demand control-flow analyses for functional languages in general. We present demand 0CFA, a demand control-flow analysis that offers clients in a functional setting the same pricing model that demand points-to analysis clients enjoy in an imperative setting. We establish demand 0CFA's correctness via an intermediary exact semantics, demand evaluation, that can potentially support demand variants of more-precise analyses.

Germane K., McCarthy J., Adams M.D., Might M. (2019) Demand Control-Flow Analysis. In: Enea C., Piskac R. (eds) Verification, Model Checking, and Abstract Interpretation. VMCAI 2019. Lecture Notes in Computer Science, vol 11388. Springer, Cham.

6.0 CONCLUSIONS

Ours was one of the only teams that took a pure static analysis approach. As we discovered, this hindered our team as the state of the art in static analysis is not yet well developed enough to work well for large scale analyses such as required by this project. However, this stimulated foundational work to address these limitations. Some of these, such as DCA, were developed late enough in the program that they were not able to be applied. However, these pave the way for future programs in allowing static analysis to reach its potential.

7.0 REFERENCES

Germane K., McCarthy J., Adams M.D., Might M. (2019) Demand Control-Flow Analysis. In: Enea C., Piskac R. (eds) Verification, Model Checking, and Abstract Interpretation. VMCAI 2019. Lecture Notes in Computer Science, vol 11388. Springer, Cham.

Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP '16, pages 407–420. ACM, New York, NY, USA, September 2016. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951936.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In Proceedings of the 43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16. ACM, New York, NY, USA, January 2016. doi: 10.1145/2837614.2837631.

LIST OF ACRONYMS

AAC – Allocating Abstract Control

ACP – Allocation Characterizes Polyvariance

CFA2 – Context-Free Approach to Control-Flow Analysis

CHA – Class-hierarchy analysis

DARPA – Defense Advanced Research Projects Agency

DCA – Demand Control-flow Analysis

IDE – Integrated Development Environment

JVM – Java virtual-machine

PDCFA – Push-Down Control-Flow Analysis

P4F – Push-down for Free

STAC – Space/Time Analysis for Cybersecurity

0CFA – Zero Control-flow Analysis