AFRL-RI-RS-TR-2019-181

BASELINING ALGORITHMIC AND SIDE-CHANNEL ISSUES WITH CHALLENGES EXPLOITATION (BASIC)

TWO SIX LABS

SEPTEMBER 2019

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-2019-181 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)*
SEPTEMBER 2019	FINAL TECHNICAL REPORT	JAN 2015 – JUN 2019

4. TITLE AND SUBTITLE

BASELINING ALGORITHMIC AND SIDE-CHANNEL ISSUES WITH CHALLENGES EXPLOITATION (BASIC)

5a. CONTRACT NUMBER
FA8750-15-C-0077

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Scott Tenaglia

5d. PROJECT NUMBER
STAC

5e. TASK NUMBER
IN

5f. WORK UNIT NUMBER
VI

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

Two Six Labs
901 N. Stuart St., Suite 1000
Arlington, VA 22203

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 Street
Arlington, VA 22203

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

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2019-181

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Final report on the STAC project, Space/Time Analysis for Cybersecurity. Report covers performance of the Two Six Labs BASIC team, the Control team. Includes information on workflows.

15. SUBJECT TERMS

Algorithmic Complexity, Space and Time Vulnerabilities, Control Team, Cybersecurity Exploits, Workflows

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	26	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

TABLE OF CONTENTS

List of Figures.....iii
List of Tables.....iii
1 SUMMARY.....1
2 INTRODUCTION.....1
2.1 Program Description.....1
2.2 Two Six Labs Role.....1
2.3 Methodology.....1
2.4 Offshoot technology.....2
2.5 Performer Results.....2
3 METHODS, ASSUMPTIONS, AND PROCEDURES.....3
4 TECHNIQUES AND WORKFLOW.....4
4.1 Profiling.....4
4.1.1 Advantages.....4
4.1.2 Disadvantages.....4
4.2 Program Interaction.....5
4.2.1 Advantages.....5
4.2.2 Disadvantages.....5
4.3 Patching and Code Injection.....5
4.3.1 Advantages.....6
4.3.2 Disadvantages.....6
4.4 Simulation.....6
4.4.1 Advantages.....7
4.4.2 Disadvantages.....7
4.5 Debugging.....7
4.5.1 Advantages.....8
4.5.2 Disadvantages.....8
4.6 Decompilation.....8
4.7 Use Case.....8
4.7.1 Advantages.....9
4.7.2 Disadvantages.....9
5 BASICLLE.....10
6 TOOLS.....11
6.1 ACSploit.....11

6.1.1 Presentations....11
6.2 Audria....12
6.3 Byteman....12
6.4 BytecodeCounter....12
6.5 BytecodeViewer....13
6.6 Debug JDK....13
6.7 Eclipse Debugger....13
6.8 JD-GUI....13
6.9 JProfiler....13
6.10 Jython....13
6.11 Luyten....13
6.12 Libmaap....14
6.13 Soot....14
7 STAC TOOLS EVALUATION....15
8 RESULTS AND DISCUSSION....16
8.1 Engagement 2....16
8.2 Engagement 4....16
8.3 Engagement 5....16
8.4 Engagement 6....17
8.5 Engagement 7....17
9 CONCLUSION....19
10 REFERENCES....20
11 List of Symbols, Abbreviations, and Acronyms....20

FIGURES

Figure 1: BASIC Team Workflows 3

Figure 2: Percentage accuracy by question type for Researchers and Control Team (Image from Apogee Research Engagement 7 Report)..... 17

TABLES

Table 1: Two Six Labs Engagement 7 Results..... ..18

1 SUMMARY

2 INTRODUCTION

2.1 Program Description

The Space/Time Analysis for Cybersecurity (STAC) program seeks to enable analysts to identify algorithmic complexity (AC) and side channel (SC) vulnerabilities, in both space and time, in software at levels of scale and speed great enough to support a methodical search for them in the software upon which the U.S. government, military, and economy depend.

2.2 Two Six Labs Role

Two Six Labs BASIC (Baselining Algorithmic and Side-channel Issues with Challenges) team functioned as the control for TA1 performers.

TA1 performers were tasked with creating beyond state-of-the-art technology to enable automated identification of AC vulnerabilities.

Two Six Labs team made use of skilled analysts and current tools to establish a baseline of best performance for comparison against TA1 development efforts.

Challenge programs created by the TA2 performers were used in a series of evaluation events throughout the life of the STAC program to measure the performance of the control team (Two Six Labs BASIC team) against the TA1 performers' novel tools and techniques.

The overall results of these engagements can be found in summary form here, or in considerably more granular detail in the engagement reports filed by the BASIC team.

2.3 Methodology

To establish, maintain, and improve on current best practices for AC vulnerability exploitation, a series of workflows were developed by the BASIC team over the life of the program. The conceptual structure of the workflows was flexible, allowing the team to adapt to changing conditions during the challenges, and the team was expected to make use of the *current* best software available, which meant updating or changing programs and utilities used over the life of the program.

As part of the development of these workflows, a toolkit of utilities and training materials was compiled and named BASICLLE (Baselining Algorithmic and Side-channel Issues with Challenges Live Linux Environment), a ready-to-use Linux ISO intended for use by an analyst during engagement challenges (though the tools and workflows are broad enough to work on or with any real-world challenge in the AC space).

2.4 Offshoot technology

Over the life of the program, a series of workflows for discovering and exploiting AC vulnerabilities were developed, see section 3 for a full breakdown of these workflows and attendant technologies.

Additionally, as part of the process of developing workflows and participating in the engagement challenges, the Two Six Labs BASIC team was also allowed to develop their own custom tool called **ACSploit**—a set of worse-case inputs to commonly used algorithms. ACSploit was successfully utilized over the life of the STAC program, demonstrated publicly at Black Hat Asia 2019 [1], and released publicly on Github [2].

2.5 Performer Results

For a quantitative analysis of performance by the various STAC teams, and particularly for a measure of the BASIC team against the TA 2 performers, we have included data gathered by TA4 over the course of the STAC program.

TA4 handled STAC challenge questions for all performers, and recorded the results for each of the engagements.

See section 8, Results and Discussion, for a high-level breakdown of performer results, or the reports submitted by TA4 for the full documents.

3 METHODS, ASSUMPTIONS, AND PROCEDURES

The BASIC team made use of a workflow to solve challenge problems. This workflow was refined repeatedly over the course of the engagements to provide a foundation for analysis utilizing the tools and techniques established by the team.

Lessons learned from the engagements produced six distinct workflows classified along two major axes: interactive vs. non-interactive and top-down vs. bottom-up.

The first axis divides workflows on the degree to which they require a running instance of the program.

Fully interactive workflows operate directly on the running program, while non-interactive workflows never run the program.

The second axis indicates how the workflow builds understanding of the program. A bottom-up workflow breaks the program down into a more fundamental representation whose components are each analyzed in turn. A top-down workflow, on the other hand, attempts to analyze the program as a whole. Eventually, all workflows should identify one or more points of interest (POIs), particular pieces of code (e.g. methods, loops, etc.) that may contain an AC or SC vulnerability. Figure 1 plots the six workflows along these axes.

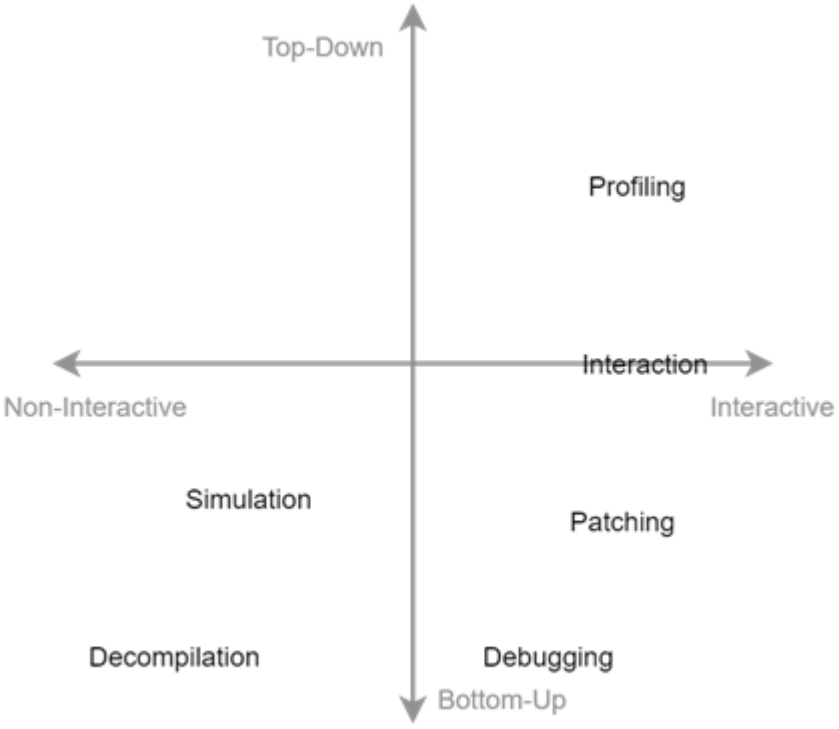

Figure 1: BASIC Team Workflows

4 TECHNIQUES AND WORKFLOW

4.1 Profiling

Profiling is a top-down dynamic analysis method that measures a program's memory usage, execution time, and frequency of instructions executed, as well as other program and system information during calls to targeted functions.

A profiling tool instruments the program and monitors its behavior, displaying the collected statistics to the operator. Java profilers usually rely on the introspection features of the Java Virtual Machine (JVM) that is executing the target program. Profiling provides a simple way to quickly understand how the program's resource usage changes in response to different inputs.

In the context of STAC, profiling helps an analyst gain a rough, but focused, understanding of a program's runtime artifacts and behavior. This assists in the discovery of side channel and algorithmically complex POIs, as well as in determining the reachability of POIs. An example of this would be identifying a list of methods that consume the most runtime for specific program inputs, potentially leading to the discovery of an SC Time vulnerability.

This workflow is useful:

1. For finding POIs without prior detailed static analysis. An analyst can run the program several times with different inputs and look for methods that consume significant CPU time or memory for those inputs
2. When developing a POC, as profiling is a simple way to ensure that the target method is invoked the appropriate number of times and confirm an analyst's understanding of the POI targeted by the POC

4.1.1 Advantages

1. Can reveal AC POIs
2. Allows analysts to verify that AC POCs work as expected

4.1.2 Disadvantages

1. Manual
2. Limited to observations of behavior induced by known inputs
3. Separating signal and noise in performance statistics can be difficult
4. Performance under profiling can differ significantly from real-world performance due to profiler overhead

4.2 Program Interaction

Program interaction is the process of running the application and providing input to drive its operations. This can be done either through pre-written scripts that remove the human from the loop or via real-time manual usage of the program UI. This workflow does not modify or "peek

inside" the program. Rather, as an extremely top-down approach, it works entirely from the outside to build a holistic picture of program behavior.

Interacting directly with the program can be a powerful tool to understanding its features. By manually analyzing program's behaviors and user interface, the analyst builds a top-down view of program functionality. This helps the analyst prioritize the parts of code on which they wish to perform deeper analysis, such as disassembly or decompilation. Automating interaction with the program through scripting allows the analyst to repeatably gather data on a variety of aspects of program behavior, from timing to RAM usage, and can be a significant aid in uncovering SC vulnerabilities and confirming all types of POCs.

This workflow is useful:

1. For initial triage and exploration of the program
2. For final tweaking of POCs and exploitation development
3. For generating truly accurate data as ground truth for further work
4. For generating data for simulation

4.2.1 Advantages

1. Requires minimal prior understanding
2. Generates data on all program outputs (intentional and unintentional)
3. Automatable

4.2.2 Disadvantages

1. Limited to the actions offered/supported by the program UI
2. Limited feedback (program output only)

4.3 Patching and Code Injection

Because of the structure of executable .jar programs and their execution by the JVM, it is easy to patch existing code and/or inject new code to alter the functioning of the target program. This process is heavily influenced by knowledge gained from decompilation. Patching and injecting code can be a shortcut to achieving some of the same goals as debugging and program interaction with much less time and effort invested by the analyst.

Patching an existing program can be an efficient way to perform dynamic analysis. Potential results range from determining code reachability (similar to debugging) to modifying fundamental program operations. Variables can be assigned to trigger some

AC/SC effect, even if the way users modified those variables are unknown. In this manner, patching an existing program can be a fast and effective way to validate suspected attack vectors.

For several applications, it is necessary for an AC/SC attacker to coordinate with other users of the challenge program. In this scenario, scripting a solution that performs normal challenge program behavior for most operations would be cumbersome. Changing the client program to insert malicious input at the appropriate time(s) is often far simpler.

This workflow is useful:

1. When the source code is well-understood by the analyst
2. For performing small modifications to the program logic

4.3.1 Advantages

1. Allows the analyst to test hypotheses regarding program behavior by monitoring and dumping internal program state
2. Allows for transformation of program into a simulator of its own behavior
3. Facilitates easy creation of POCs (e.g. turning the client program into a “malicious” client)

4.3.2 Disadvantages

1. Requires understanding of source code layout and semantics

4.4 Simulation

Simulation is the process of modeling a possibly simplified subset of program behavior in a script to determine the likely results of exploiting a potential attack vector. The simulation workflow requires a custom script/program written for the user to interact with, and is meant to model only a single aspect of the program. This differs from the scripting component of the program interaction workflow, which uses scripts to interact with the actual target program. Simulation is also distinct from the patching workflow, which alters or augments the target program rather than simplifying it.

The goal of simulation is to simplify the target program's execution to the essentials relevant to an attack vector. This quickly determines if it is even possible for an attack vector to produce the desired effect on the target program. The simulation often makes many assumptions and simplifications in the attacker's favor since, if the attack vector is not effective in this most favorable world, it will certainly not be effective against the actual program.

This workflow is useful:

1. For modeling side-channels in space, when possible sizes and their distribution are already known
2. For modeling side-channels in time, when simulating a "perfect" environment without extra noise to see if an attack vector is even possible

4.4.1 Advantages

1. Simplifying assumptions allow quick development of prototype simulations
2. Simulations can definitively rule out POIs, providing a rare shortcut to negative answers on SC problems
3. Can easily be scaled (e.g. by running longer)
4. Combines well with more active methods (patching, profiling, etc.)

4.4.2 Disadvantages

1. Requires upfront work to create the simulation code/program
2. Results may not be useful (i.e. a simulation showing a POI is vulnerable may not help develop a POC to exploit the POI)
3. Incorrectly chosen or overly broad assumptions may invalidate simulation results

4.5 Debugging

Debugging is a bottom-up dynamic program analysis process for identifying, locating, and investigating programming errors in an application. A tool called a “debugger” wraps around the program and allows fine-grained runtime introspection. Unlike profiling, which aggregates data about the program’s interaction with the environment, debugging allows actual program states to be observed and manipulated through debugger-specific functionality, such as breakpoints.

State information gathered at each breakpoint allows the analyst to iteratively change program inputs until the desired state is reached. This technique creates a feedback loop between the analyst and the program, similar to how an automated fuzzer mutates inputs based on dynamic code coverage information.

Using a debugger helps an operator gain a precise understanding of a program's state at a given point in execution when provided with a given set of inputs. This assists in discovering side- channel and algorithmic complexity POIs, as well as in determining the reachability of POIs. As an example, consider a program that compares user input to a secret password character-by- character. The analyst would place a breakpoint at the comparison instruction, most likely inside a loop, causing the program to hit the breakpoint once for each character in the password, thus leaking the length of this secret.

This workflow is useful:

1. When modifying the program state at a specific point in execution to alter the program's control path. This is useful for artificially generating worst-case conditions without knowing the worst-case inputs

2. When micro-level observations of the program one instruction and/or source code line at a time are needed to explore a specific POI or hypothesis about program behavior

3. For observing runtime behavior and program state to add insight to static code review. For example, a worst-case complexity that is hard to spot via manual code review may become apparent when the program is run

4.5.1 Advantages

1. Enables analysis of code reachability
2. Enables inspection of internal program state

4.5.2 Disadvantages

1. Limited to observations of behavior induced by known inputs
2. Manual, error-prone, and time-consuming
3. Requires some pre-existing knowledge of the source code layout of the program

4.6 Decompilation

Decompilation is a form of reverse engineering that translates compiled code to a more abstract representation for analysts or automated tools to consume. It is a bottom-up static workflow that aims to recover a source code's individual files representing the target program from its compiled binary. This workflow thus allows for more rapid and thorough analysis of the target program by analysts, as source code is much more semantically accessible to humans than compiled binaries. Recovering a source representation of the target program also presents an opportunity to create modified versions of the program for various analytic purposes (see the Patching workflow).

4.7 Use Case

Decompilation is the only form of static analysis used by the STAC Control Team. It is used on its own to help analysts gain a semantic understanding of how a program works and to input into other workflows. Semantic understanding aids in the search for locations of potential SC or AC vulnerabilities as well as determining how program inputs reach vulnerable code. This latter information supplements other workflows with capabilities not readily available in their associated tooling.

Decompilation is useful when source code is unavailable, and the operator requires an in-depth understanding of one or more of the program's features. Decompilation is less useful when the compiled code decompiles to a source that is not easily understood because of either intentional obfuscation or because the scale of the decompiled source exceeds the limits of human consumption and comprehension.

In summary, decompilation workflow is useful in two ways:

1. When source code is unavailable
2. To complement "black-box" challenge analysis by manually tracing an input value's execution path. This is useful for seeing the bounds placed on the control flow path that the input traverses and determining if hitting these bounds could cause information leaks

4.7.1 Advantages

1. Provides a "ground truth" understanding for other workflows
2. Provides a baseline understanding of program flow and function for analysts Can be performed as deep or shallow as desired

4.7.2 Disadvantage

Fully manual, therefore hard to scale and coordinate between analysts and prone to human error: Time-consuming

5 BASICLLE

The Baselining Algorithmic and Side-channel Issues with Challenges Live Linux Environment, or BASICLLE (pronounced “basically”), is a custom Linux distribution created to package AC and SC vulnerability discovery tools in a convenient ISO package.

The concept was initially tested as a virtual machine, then converted to an ISO for easy distribution on DVDs or flash drives, which is useful for bringing to a live Engagement or location with restricted internet access.

BASICLLE was established early in the program’s lifecycle and revised several times over the course of the program. BASICLLE training materials were utilized by new staff assigned to the BASIC team.

5.1 Training

The BASIC team created a set of documentation covering how to use the tools installed on BASICLLE, along with the workflows developed by the BASIC team, to find and exploit AC and SC vulnerabilities in Java programs.
The training material was intended to provide a technically skilled individual (without extensive knowledge of AC/SC exploits) proficiency in a variety of reverse engineering techniques used by the BASIC team over the course of the STAC program.

These training documents were included as part of the BASICLLE package.

6 TOOLS

The suite of tools utilized by the BASIC team and documented in the BASICLLE training materials over the life of the STAC program was extensive. A summary and brief description of those tools follows.

Note that not all tools were used for the full duration of the program, as for example, a shift in focus from Java 7 to Java 8 around Engagement 5 meant some tools became obsolete due to a lack of support for the newer version of Java.

Likewise, some tools used very early in the program may have been dropped before they were included in the BASICLLE training materials, whether due to underperformance, or the availability of stronger alternatives.

The list should prove suitable for providing a snapshot of the types of utilities utilized by the BASIC team.

Note that this list should not be taken as an authoritative or immutable list of 'perfect tools', but rather, the tools that best fit the needs of the team based on the challenges encountered during the STAC program. The training documentation created by the BASIC team encourages analysts to explore these and other tools to find the best fit for the job at hand.

6.1 ACSploit

An offshoot of work performed by the BASIC team, ACSploit is a tool (located on a GitHub repo [2]) to generate worst-case inputs for commonly used algorithms. These worst-case inputs cause the target program to utilize a large amount of resources (e.g. time or memory).

The team has used ACSploit to discover weaknesses in several open source software programs, with all information responsibly disclosed to the authors/maintainers. ACSploit grew organically as a way to quickly triage well known algorithms that repeatedly appeared in the STAC challenge problems. ACSploit does not find AC vulnerabilities like TA1 performer technology, rather it tests for the existence of a vulnerability by generating an input known to trigger it, should it exist. It is more akin to exploit frameworks like Metasploit that generate exploits for known vulnerabilities, rather than find vulnerabilities in programs.

6.1.1 Presentations

ACSploit was presented at the Black Hat Asia conference [1]. The tool was demonstrated by the Principal Investigator Scott Tenaglia, who performed a live demonstration, and presented background information on the BASIC team's work in creating the tool.

6.2 Audria

Audria is a tool for reading various program statistics from the /proc filesystem. Audria can monitor the following statistics:

1. current and average CPU usage
2. virtual memory usage (peak, resident set, swap etc.)
3. IO load (current/total read/writes with and without buffers)
4. time spent in user/system mode
5. page faults
6. and other information

Audria can monitor a single process, a group of processes, or all currently running processes. It can watch existing processes or spawn processes and watch them directly from startup. Audria is designed to run in batch mode at very short intervals and thus allows very detailed inspections of the resource usage of a process. Audria is a useful complement to a debugger.

6.3 Byteman

Byteman is a bytecode manipulation tool that makes it simple to manipulate Java applications at load time and/or runtime. Byteman does not rewrite or alter the original program code, so it can even modify core Java classes such as String, Thread, etc. Byteman uses an Event Condition Action (ECA) rule language based on Java. ECA rules are used to specify where, when, and how the source Java code of the target application should be modified.

Byteman provides the following four main features:

1. execution tracing of specific code paths and displaying application and JVM state
2. subverting normal execution by changing application or JVM state, making unscheduled method calls, or forcing an unexpected return or throw
3. orchestrating the timing of independent application threads
4. monitoring and gathering statistics on application and JVM performance

The Byteman core engine is a general-purpose code injection program capable of injecting inline Java code into almost any location reachable during execution of a Java method. Byteman rule conditions and actions can employ all the normal Java built-in operations in order to test and modify program state. They can also invoke application and JVM methods that are in scope at the injection point.

6.4 BytecodeCounter

BytecodeCounter is a JVM agent library that counts the number of Java bytecode instructions executed by a Java application on a per-method basis.

6.5 BytecodeViewer

BytecodeViewer is a GUI application for viewing and editing decompiled Java code, decompiled bytecode, and disassembled bytecode. BV comes with five decompilers: JD-GUI/Core, Procyon, CFR, Fernflower, and Krakatau.

BytecodeViewer also provides:

1. DEX2JAR conversion
2. JAR2DEX conversion
3. APK decompilation
4. Smali / Baksmali integration
5. Search utilities
6. Plugin system

6.6 Debug JDK

The Debug JDK is a debug build of the Java Development Kit (JDK) with certain additional debug features enabled.

6.7 Eclipse Debugger

The Java IDE Eclipse can debug a Java application using source code decompiled from a JAR with a tool such as JD-GUI.

6.8 JD-GUI

JD-GUI is a standalone graphical utility that decompiles ".class" files and displays the reconstructed Java source code.

6.9 JProfiler

JProfiler is a Java profiler that measures and displays detailed performance information, including CPU usage, memory usage, and thread performance.

JProfiler is a commercial product that requires a purchased license after the trial period ends.

6.10 Jython

Jython is an implementation of Python in pure Java that allows the rapid scripting capabilities of Python to be used on Java classes. Jython allows Python code to interact with and inspect Java classes and objects. Jython works directly on class files and Java bytecode so there is no need for decompilation.

6.11 Luyten

Luyten is a standalone graphical utility that decompiles ".class" files and displays the reconstructed Java source code. It is a GUI built on top of the Procyon Java decompiler.

Luyten is especially useful when dealing with Java 8 code, as JD-GUI does not fully support all Java 8 bytecode.

6.12 Libmaap

Early in the program, Libmaap was a set of open source programs and 'glue code' used to analyze binaries from the Google Play marketplace, specifically to identify the most common libraries used by the most popular apps on the marketplace.

Usage of this tool was discontinued however, after a decision was made by the team to avoid targeting live Android code for vulnerabilities.

6.13 Soot

Soot is a Java manipulation and optimization framework. Soot provides intermediate representation languages for analysis, instrumentation, optimization, and visualization of Java and Android applications.

Soot's intermediate representations are:

- Baf: a streamlined representation of bytecode that is simple to manipulate
- Jimple: a typed 3-address intermediate representation suitable for optimization
- Shimple: an SSA variation of Jimple
- Grimp: an aggregated version of Jimple suitable for decompilation and code inspection

There is an Eclipse plugin for Soot to streamline transformations.

7 STAC TOOLS EVALUATION

At the end of 2018, the team was asked to perform an analysis of the tools developed by the TA1 performers. This included tools from Iowa State, Northwestern, Vanderbilt, Utah, Draper, UC Boulder, University of Maryland, and GrammaTech.

There were considerable difficulties building and running many of the tools, as the majority were custom programs built for specific runtime environments that did not always translate cleanly to an easy package for the BASIC team to test. However, this is to be expected of prototype solutions undergoing active development in an ongoing DARPA program. Full results can be viewed in the submitted report file [3].

8 RESULTS AND DISCUSSION

As the control team for the STAC program, the BASIC team was used as a basis for comparison with the automated tools and techniques developed by the TA1 performers.

Full reports for each engagement were submitted by the TA4 performer. A summary of top-level results is included here.

Note that aggregate scores were provided for engagements 2 and 4, as 1 and 3 were live versions of the same engagement questions, and the results were consolidated by TA4, the performer in charge of the engagement organization, before they consolidated the testing on the future engagements (from 5 to 7).

Terms used in the reports

- TPR: True Positive Rate, the ratio of the number of correct vulnerable response to total number of vulnerable questions answered
- TNR: True Negative Rate the ratio of the number of correct non-vulnerable responses to the total number of non-vulnerable questions answered

8.1 Engagement 2

Two Six Labs answered all of the E2 questions with a 71% accuracy. Two Six Labs had an 86% accuracy and a 70% accuracy for SC Space and SC Time vulnerabilities respectively.

8.2 Engagement 4

Two Six Labs answered 39 questions with 90% accuracy. In E4, Two Six Labs had the second highest accuracy and the highest TPR against the set of all questions. Two Six Labs' accuracy increased by 16 percentage points from E2 and 23 percentage points from E3.

8.3 Engagement 5

Two Six Labs answered 28 questions during the engagement with the second highest take-home accuracy of 71% and the second highest collaborative accuracy of 89%, despite being limited on which questions they could collaborate on. Two Six was one of two teams to achieve a 100% accuracy in cases where they changed answers between the take-home and collaborative engagements.

During the take-home engagement the team had the most difficulty with AC Space questions and "No" responses, achieving accuracies of 56% and 50% respectively. Following the collaborative engagement their accuracy on answered questions across all

categories was ≥ 75% and their "No" accuracy was 100%. After the live engagement the team tied for 2nd on True Positive Rate and tied for 1st on True Positive Rate for both the answered questions and all questions segmentations.

8.4 Engagement 6

Two Six Labs outperformed all the research teams during the take-home engagement with an 89% accuracy and 38 of the 43 questions answered. As a result of this performance the team was excluded from the live collaborative engagement to allow the research teams to attempt to match the control team's performance by working together. Two Six Labs tied for the highest TPR (75%) and had the highest TNR (100%). Against the segmentation of all questions (unanswered questions treated as incorrect) Two Six Labs outperformed the research teams with the highest total accuracy of 79% and the highest TPR of 71%. The team's TNR in the all questions segmentation decreased from 100% to the third highest of 85%.

8.5 Engagement 7

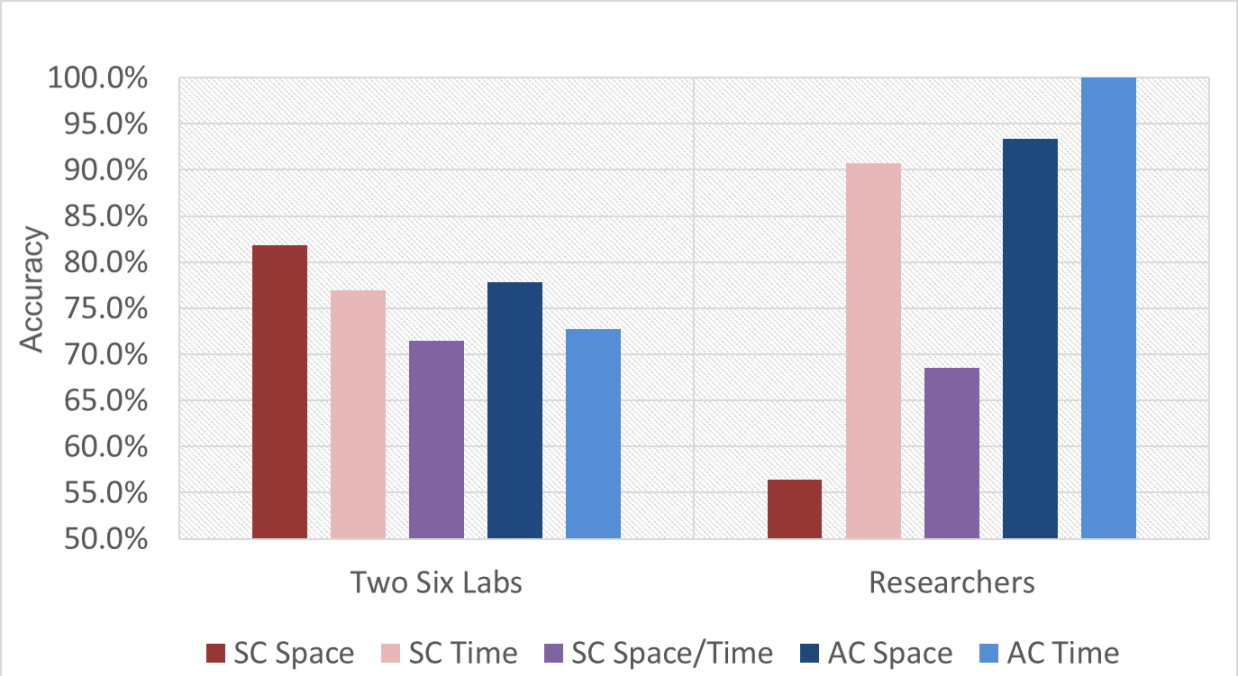

Figure 2 Percentage accuracy by question type for Researchers and Control Team (Image from Apogee Research Engagement 7 Report)

Table 1: Two Six Labs Engagement 7 Results

Question Type	**Number Attempted**	**Reviewed Number Correct**	**Reviewed Accuracy (%)**
SC Space	11	9	81
SC Time	13	10	77
SC Space/ Time	7	5	71
AC in Space	18	14	78
AC in Time	11	8	72
Total	**60**	**46**	**77**
Vulnerable/ Response	**Number Attempted**	**Reviewed Number Correct**	**Reviewed Accuracy (%)**
Vulnerable	30	16	**53**
Not Vulnerable	30	30	**100**
Yes Answer	17	16	**94**
No Answer	43	30	**70**

9 CONCLUSION

Two Six Labs worked as the control team for the STAC program, providing a baseline ‘state-of- the-art performance’ to judge the TA1 performers against.

Over the life of the program, the Two Six Labs BASIC team participated in seven challenge engagements conducted as a mix of live and take-home exercises. In addition, Two Six Labs was tasked with reviewing TA1 performer tools near the end of 2018.

To build a consistent and reliable method of approaching the engagements, the BASIC team developed the BASICLLE toolset, accompanying training documentation, and a lengthy series of workflows derived from their work during the challenge engagements.

Finally, during the program, Two Six Labs work produced a spinoff tool known as ‘ACSploit’, to trigger the worst-case in commonly used algorithms, the results of which were discussed publicly and released as open source software.

10 REFERENCES

[1] [Online]. Available: https://www.blackhat.com/asia-19/briefings/schedule/#acsploit-exploit-algorithmic-complexity-vulnerabilities-14426.

[2] [Online]. Available: https://github.com/twosixlabs/acsploit.

[3] T. S. Labs, "Two Six Labs TA1 Tool Review For DARPA STAC Program," 2018.

11 List of Symbols, Abbreviations, and Acronyms

Abbreviation	Meaning
AC	Algorithmic Complexity
BASIC	Baselining Algorithmic and Side-channel Issues with Challenges
BASICLLE	Baselining Algorithmic and Side-channel Issues with Challenges Live Linux Environment
CPU	Central Processing Unit
ISO	ISO Image
JDK	Java Developers Kit
JVM	Java Virtual Machine
POC	Proof of Concept
POI	Point of Interest (computational)
RAM	Random Access Memory
STAC	Space/Time Analysis for Cybersecurity
TNR	True Negative Rate
TPR	True Positive Rate