AFRL-RI-RS-TR-2019-192

CONTINUUM: FINDING SPACE AND TIME VULNERABILITIES IN JAVA PROGRAMS

NORTHEASTERN UNIVERSITY

OCTOBER 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-2019-192 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)*
OCTOBER 2019	FINAL TECHNICAL REPORT	APR 2015 – APR 2019

4. TITLE AND SUBTITLE

CONTINUUM: FINDING SPACE AND TIME VULNERABILITIES IN JAVA PROGRAMS

5a. CONTRACT NUMBER
FA8750-15-2-0084

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

William Robertson, Engin Kirda, Christopher Kruegel, Giovanni Vigna, Manuel Egele

5d. PROJECT NUMBER
STAC

5e. TASK NUMBER
NE

5f. WORK UNIT NUMBER
ST

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northeastern University
360 Huntington Ave.
Boston, MA 02115

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-2019-192

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 Continuum project aimed to develop novel techniques to preemptively identify AC vulnerabilities in both the time and space domains. As part of this effort, a prototype implementation of these techniques was developed as part of a scalable, integrated platform that could be used by both expert and non-expert analysts on real software authored for the Java Virtual Machine (JVM). Continuum led to the development of new vulnerability detection methods that have been empirically shown to discover zero-day bugs in widely used software. We conclude the report by laying out promising directions for future research.

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	22	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

1.0 SUMMARY

Software continues to be plagued by vulnerabilities that allow attackers to violate basic software security properties. These vulnerabilities take myriad forms, for instance failures to enforcement or safety that can lead to arbitrary code execution (integrity violations) or failures to prevent sensitive data from being released to unauthorized principals (confidentiality violations). The third traditional security property, availability, is not exempt from this issue. However, denial-of-service (DoS) as a vulnerability class tends to be viewed as simplistic, noisy, and easy—in principle— to defend against. This view, however, is simplistic, as availability vulnerabilities and exploits against them can take sophisticated forms. Algorithmic complexity (AC) vulnerabilities are one such form, where a small adversarial input induces worst-case behavior in the processing of that input, resulting in a denial of service.[1]

The goal of the Continuum project was to develop novel techniques to preemptively identify AC vulnerabilities in both the time and space domains. As part of this effort, a prototype implementation of these techniques was developed as part of a scalable, integrated platform that could be used by both expert and non-expert analysts on real software authored for the Java Virtual Machine (JVM). In both of these respects, this four-year effort was successful. Continuum has led to the development of new vulnerability detection methods that have been empirically shown to discover zero-day bugs in widely-used software, and the resulting platform continues to be developed and enhanced today.

In this report, we first describe the current state of the Continuum analysis platform and its dynamic analysis component HotFuzz. We then describe other major scientific results that have sprung from Continuum research efforts: Rampart, DIFUZE, and HACRS. Finally, we conclude and lay out promising directions for future work in automated vulnerability discovery.

2.0 INTRODUCTION

The Continuum platform is composed of two major components: a static analyzer based on the angr [6] platform, and a dynamic analyzer called HotFuzz. Continuum's static analyzer builds on angr to support integrated analysis of Java bytecode and all native code architectures already supported by angr. Facts derived from static taint and more sophisticated context-sensitive data flow analysis is used to target HotFuzz towards *potentially* vulnerable paths of an analysis target, e.g., by ruling out paths that cannot be influenced by adversarial input. Program-specific static analyses have also been developed that incorporate static cost models in the numeric domain. These analyses propagate symbolic expressions that represent upper bounds on dynamic resource usage, further refining the set of suspicious paths to analyze.

HotFuzz attempts to dynamically maximize resource consumption using novel optimization-guided fuzzing techniques, and uses a set of empirically tuned heuristics to report potential vulnerabilities to the analyst. Human intuition can also be incorporated into the analysis by providing inputs that expand the platform's coverage of difficult-to-exercise code paths. The results then feed into further partially concretized static analyses, producing a powerful iterative analysis capable of handling large, real-world program artifacts.

[1]Strictly speaking, it is sufficient for attacks to cause "bad" behavior; it need not be worst-case *per se*.

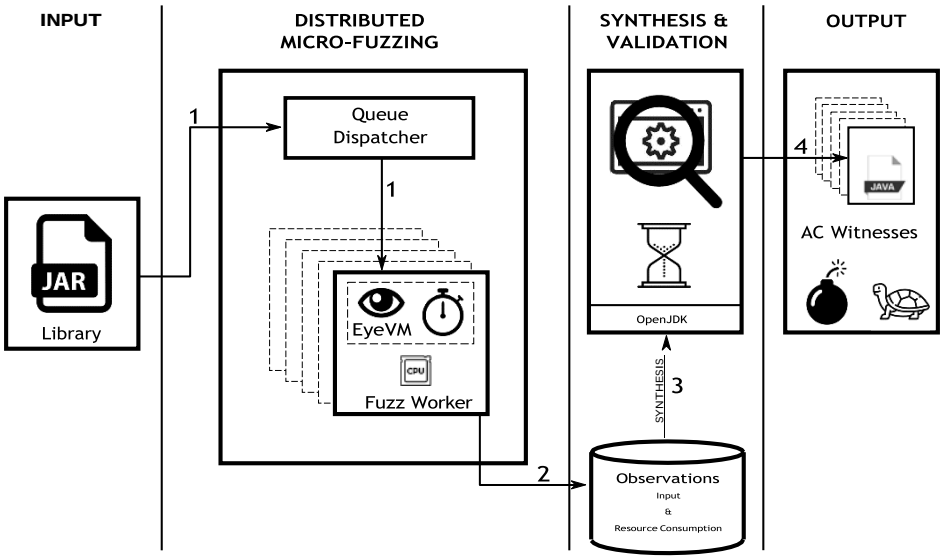

Figure 1: Architectural overview of the HotFuzz testing procedure. In the first phase, individual μFuzz instances micro-fuzz each method comprising a library under test. Resource consumption is maximized using genetic optimization over inputs seeded using either IVI or SRI. In the second phase, test cases flagged by the first phase are synthesized into Java programs. These programs are executed in an unmodified JVM in order to replicate the abnormal resource consumption observed in the first phase. Programs that do not do so are rejected as false positives, while HotFuzz reports those that do as validated AC vulnerability witnesses to a human analyst.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

HotFuzz adopts a dynamic testing approach to detecting AC vulnerabilities, where the testing procedure consists of two phases: *(i)* micro-fuzzing, and *(ii)* witness synthesis and validation. In the first phase, a Java library under test is submitted for *micro-fuzzing*, a novel approach to scale AC vulnerability detection. In this process, the library is decomposed into individual methods, where each method is considered a distinct entry-point for testing by a *μFuzz* instance. As opposed to traditional fuzzing, where the goal is to provide inputs that crash a test program, here each μFuzz instance attempts to maximize the resource consumption of individual methods under test using genetic optimization over its inputs. To that end, seed inputs for each method under test are generated using one of two instantiation strategies: *identity value instantiation* (IVI) and *small recursive instantiation* (SRI). Method-level resource consumption when executed on these inputs is measured using a specially instrumented Java virtual machine we call the *EyeVM*. If optimization eventually produces an execution that is measured to exceed a pre-defined threshold, then that test case is forwarded to the second phase of the testing procedure.

The purpose of the second phase is to validate whether test cases found during micro-fuzzing represent actual vulnerabilities when executed in a real Java run-time environment, as differences between the fuzzing and realistic execution environments can lead to false positives. This validation is achieved through *witness synthesis* where, for each test case discovered by the first phase, a program is generated that invokes the method under test with the associated inputs that produce abnormal resource usage. If the behavior with respect to resource utilization that was observed during micro-fuzzing is replicated, then the synthesized test case is flagged as a witness of the vul-

nerability that can then be examined by a human analyst. Figure 1 depicts a graphical overview of the two phases. In the following, we motivate and de- scribe the design of each component of the testing procedure in detail.

3.1 Micro-Fuzzing

Micro-fuzzing represents a drastically different approach to vulnerability detection than traditional automated whole-program fuzzing. In the latter case, inputs are generated for an entire program either randomly, through mutation of seed inputs, or incorporating feedback from introspection on execution. Whole-program fuzzing has the significant benefit that any abnormal behavior—i.e., crashes—that is observed should be considered as a real vulnerability as by definition all path constraints on the input are satisfied (up to the determinism of the execution). However, whole-program fuzzing also has the well-known drawback that full coverage of the test artifact is difficult to achieve. Thus, an important measure of a traditional fuzzer's efficacy is its ability to efficiently cover paths in a test artifact.

Micro fuzzing strikes a different trade-off between coverage and path satisfiability. Inspired by the concept of micro-execution [2], micro-fuzzing constructs realistic intermediate program states and directly executes individual methods on these states. Thus, coverage is trivial to achieve by simply enumerating all methods comprising the test artifact, while the difficulty lies instead in ensuring that constructed states used as method inputs are feasible in practice.[2] In our problem setting, where we aim to preemptively warn developers against insecure usage of AC-vulnerable methods or conservatively defend against powerful adversaries, we believe micro-fuzzing represents an interesting and useful point in the design space.

A second major departure from traditional fuzzing is the criteria used to identify vulnerabilities. Typical fuzzers use abnormal termination as a signal that a vulnerability might have been found. In our case, vulnerabilities are represented not by crashes but rather by excessive space or time consumption. Thus, coverage is not the sole metric that must be maximized in our case. Instead, HotFuzz must measure and optimize space and/or time usage and attempt to maximize these metrics *in addition* to coverage. Conceptually speaking, resource measurement is a straightforward matter of adding an API to toggle resource usage recording and to associate measurements with Java methods. In practice, this involves non-trivial engineering, the details of which are described in subsequent sections. In the following, we describe how HotFuzz optimizes resource consumption during micro fuzzing given dynamic measurements provided by the EyeVM.

3.1.1 Resource Consumption Optimization.

HotFuzz's fuzzing component, called μFuzz, is responsible for optimizing the resource consumption of methods under test. To do so, μFuzz uses genetic optimization to evolve an initial set of seed inputs over multiple generations until abnormal resource consumption is detected. Genetic optimization has been extensively used in traditional fuzzers, with two notable differences. First, as already discussed, traditional fuzzers optimize an objective function that solely considers path coverage (or some proxy thereof), whereas in our setting we are concerned in addition with resource consumption. Second, traditional fuzzers treat inputs as flat bitmaps when genetic optimization (as opposed to more general mutation) is applied.

[2] We note that in the traditional fuzzing case, a similar problem exists in that while crashes indicate the presence of an availability vulnerability, they do not necessarily represent exploitable opportunities for control-flow hijacking.

Recall that genetic algorithms require defining crossover and mutation operators on members of the population of inputs. New generations are created by performing crossover between members in prior generations. Additionally, in each generation, some random subset of the population undergoes mutation with small probability. Since μFuzz operates on Java values rather than flat bitmaps, we must define new crossover and mutation operators specific to this domain, as bitmap-specific operators do not directly translate to arbitrary Java values.

Java Value Crossover. Genetic algorithms create new members of a population by "crossing" existing members. When individual inputs are represented as bitmaps, a standard approach is single-point crossover: a single offset into two bitmaps is selected at random, and two new bitmaps are produced by exchanging the content to the right of the offset from both parents. Single-point crossover does not directly apply to inputs comprised of Java objects, but can be adapted in the following way. Let X_0, X_1 represent two existing inputs from the overall population. To produce two new inputs, perform single-point crossover for each corresponding pair of values (x_0, x_1) (X_0, X_1) using

$$(x_0', x_1') = \begin{cases} C(x_0, x_1) & \text{if } (x_0, x_1) \text{ are primitives} \\ (C^L(x_0, x_1), C^R(x_0, x_1)) & \text{if } (x_0, x_1) \text{ are objects.} \end{cases}$$

Here, C performs one-point crossover directly on primitive values, while C_L and C_R recursively perform cross-over on the members of (x_0, x_1) and select the left and right offspring, respectively.

Java Object Mutation. Mutation operators for traditional fuzzers rely on heuristics to derive new generations; mutating members of the existing population through random or semi-controlled bit flips. In contrast, micro fuzzing requires mutating arbitrary Java values, and thus bitmap- specific techniques do not directly apply. Instead, μFuzz mutates Java objects using the following procedure. For a given Java object x with attributes $\{a_0, a_1, \ldots, a_n\}$, choose one of its attributes a_i at random. Then we define the mutation operator M as

$$a_i' = \begin{cases} M_{\text{flip\_bit}}(a_i) & \text{if } a_i \text{ is a numeric value,} \\ M_{\text{insert\_char}}(a_i) & \text{if } a_i \text{ is a string or array value,} \\ M_{\text{delete\_char}}(a_i) & \text{if } a_i \text{ is a string or array value,} \\ M_{\text{replace\_char}}(a_i) & \text{if } a_i \text{ is a string or array value,} \\ M_{\text{swap\_chars}}(a_i) & \text{if } a_i \text{ is a string or array value,} \\ M_{\text{mutate\_attr}}(a_i) & \text{if } a_i \text{ is an object.} \end{cases}$$

Each mutation sub-operator above operates on a random element or pair of elements. For example, $M_{\text{flip\_bit}}$ selects a random bit in a numeric element and flips it, while $M_{\text{swap\_chars}}$ randomly selects two elements of a string or array and swaps them. The other sub-operators are defined in an intuitively similar manner.

3.1.2 Seed Generation.

Given suitable crossover and mutation operators, all that remains to apply standard genetic optimization is the definition of a procedure to generate seed inputs. We define two such procedures that we describe below. Identity Value Instantiation (IVI), and Small Recursive Instantiation (SRI).

Identity Value Instantiation. Recent work has proposed guidelines for evaluating new fuzz testing techniques [3]. One of these guidelines is to compare any proposed strategy for constructing seed inputs for fuzz testing with "empty" seed inputs. Since empty bitmaps do not directly translate to our input domain, we define IVI as an equivalent strategy for Java values. The term "identity value" is derived from the definition of an identity element for an additive group.

In particular, IVI is defined as

$$I(T) = \begin{cases} 0 & \text{if } T \text{ is a numeric type,} \\ \text{""} & \text{if } T \text{ is a string,} \\ \{\} & \text{if } T \text{ is an array,} \\ T_{\text{random}}\,(I(T_0), \ldots, I(T_n)) & \text{if } T \text{ is a class.} \end{cases}$$

That is, $I(T)$ selects the identity element for all primitive types, while for classes I is recursively applied to all parameter types T_i of a randomly selected constructor for T. Thus, for a given method under test M, $I(M)$ is defined as I applied to each of M's parameter types.

Small Recursive Instantiation. In addition to IVI, we define a complementary seed input generation procedure called Small Recursive Instantiation (SRI). In contrast to IVI, SRI generates random values for each method parameter. However, experience dictates that selecting uniformly random values from the entire range of possible values for a given type is not the most productive approach to input generation. Therefore, we configure SRI with a spread parameter α that limits the range of values from which SRI will sample. Thus, SRI is defined as

$$S(T, \alpha) = \begin{cases} R_{\text{num}}(-\alpha, \alpha) & \text{if } T \text{ is a numeric type,} \\ \{R_{\text{char}}\}^{R_{\text{num}}(0,\alpha)} & \text{if } T \text{ is a string,} \\ \{S(T, \alpha)\}^{R_{\text{num}}(0,\alpha)} & \text{if } T \text{ is an array,} \\ T_{\text{random}}\,(S(T_0, \alpha), \ldots, S(T_n, \alpha)) & \text{if } T \text{ is a class.} \end{cases}$$

In the above, $R_{\text{num}}(x, y)$ selects a value on the range $[x, y)$ uniformly at random, while R_{char} produces a character chosen uniformly at random. Similarly to I, for a given method under test M we define $S(M)$ as S applied to each of M's parameter types. We note that SRI with $\alpha = 0$ is in fact equivalent to IVI, and thus IVI can be considered a special case of SRI.

Seed Population Instantiation. Given the above definitions for IVI and SRI, the parameter α, and the method under test M, μFuzz instantiates a full seed population of size n as follows. First, $\frac{n}{4}$ inputs are constructed using IVI, i.e., $S(M, 0)$. Another set of inputs are constructed as $\{S(M, i)\}\ \forall_4 0 < i \leq \frac{n}{2}$. Finally, the last $\frac{n}{2}$ inputs are constructed using $S(M, \alpha)$. We note that defining the population with a range of values for α allows μFuzz to test M on varying inputs, increasing the chances that interesting resource consumption behaviors are exposed during testing.

3.2 Witness Synthesis

Test cases exhibiting abnormal resource consumption are forwarded from the micro-fuzzing phase of the testing procedure to the second phase: witness synthesis and validation. The rationale be-

hind this phase is to reproduce the behavior observed during fuzzing in a realistic execution environment using a production JVM in order to avoid any false positives introduced due to the measurement instrumentation.

In principle, one could simply interpret any execution that exceeds the configured timeout as evidence of a vulnerability. In practice, this is an insufficient criterion since the method under test could simply be blocked on I/O, sleeping, or performing some other benign activity. An additional consideration is that because the EyeVM operates in interpreted mode by default, a test case that exceeds the timeout in this phase might not do so when the JIT is enabled.

Therefore, validation of suspected vulnerabilities in a realistic environment is necessary. To that end, given an abnormal method invocation $M(v_0, \ldots, v_n)$, a self-contained Java program is synthesized that invokes M using the Java Reflection API. The program is packaged with any necessary library dependencies and is then executed in a standard JVM with JIT enabled. Instead of using JVM instrumentation, the wall clock execution time of the entire program is measured. If the execution was both CPU-bound as measured by the operating system and the elapsed wall clock time exceeds a configured timeout, the synthesized program is considered a witness for a legitimate AC vulnerability and recorded in serialized form in a database. The resulting corpus of AC vulnerability witnesses are reported to a human analyst for manual examination.

3.3 Implementation

Our prototype implementation of HotFuzz consists of 5487 lines of Java code and 1007 lines of C++ code in the JVM.

3.3.1 EyeVM.

The OpenJDK includes the HotSpot VM, an implementation of the Java Virtual Machine (JVM), and the libraries and toolchain that support the development and execution of Java programs. The EyeVM is a fork of the OpenJDK that includes a modified HotSpot VM for recording resource measurements. By modifying the HotSpot VM directly, our fuzzing procedure is compatible with any library that runs on it. The EyeVM exposes its resource usage measurement capabilities to analysis tools using the Java Native Interface (JNI) framework. In particular, a fuzzer running on the EyeVM can obtain the execution time of a given method under test by invoking `java.lang.reflect.Executable.getTimeConsumed()` on a corresponding reflective method handle.

We chose to instrument the JVM directly because it allows us to analyze programs without altering them through bytecode instrumentation. This enables us to micro-fuzz a library without modifying it in any way. It also limits the amount of overhead introduced by recording resource measurements.

The EyeVM can operate in two distinct modes to support our resource consumption analysis: *measurement* and *tracing*. In measurement mode, the EyeVM records program execution time with method-level granularity, while tracing mode records method-level execution traces of programs running on the EyeVM. HotFuzz utilizes measurement mode to record the method under test's execution time during micro fuzzing, while tracing mode allows for manual analysis of suspected AC vulnerabilities.

EyeVM Measurement Mode. Commodity JVMs do not provide a convenient mechanism for recording method execution times. Prior work has made use of bytecode rewriting [4] for this purpose.

However, this approach requires modifying the test artifact, and produced non-trivial measurement perturbation in our testing. Alternatively, an external interface such as the Serviceability Agent [5] or JVM Tool Interface [8] could be used. However, these approaches introduce costly overhead due to context switching every time the JVM invokes a method. Therefore, we chose to collect resource measurements by instrumenting the HotSpot VM directly.

The HotSpot VM source code contains an Assembler class that defines an API for authoring assembly programs. The HotSpot VM uses this API to implement an interpreter for every instruction set architecture that the JVM supports. This allows the JVM to generate interpreters at runtime. We instrumented the JVM interpreter by inserting instructions using the Assembler API in the JVM source. To measure execution time, we modify method entry and exit to store elapsed time in thread-local data structures that analysis tools can query after a method returns. We implement this instrumentation using the same Assembler API the JVM uses to generate bytecode interpreters. When saving the measured execution time for the method under test, we use the Assembler API to compute the address to the measurement field for the running thread, and generate a `mov` instruction to perform the store. Our current implementation is limited to the `x86-64` platform. However, this technique can be applied to any architecture supported by the HotSpot VM.

Unfortunately, instrumenting the JVM such that *every* method invocation and return records that method's execution time introduces significant overhead. That is, analyzing a single method also results in recording measurements for every method it invokes in turn. This is both unnecessary and adds noise to the results due to both the need to perform an additional measurement for each method as well as the adverse effects on the cache due to the presence of the corresponding measurement field. Thus, our implementation avoids this overhead by restricting instrumentation to a single method under test that μFuzz can change on demand.

In particular, μFuzz stores the current method under test inside thread local data. During method entry and exit, the interpreter compares the current method to the thread's method under test. If these differ, the interpreter simply jumps over our instrumentation code. Therefore, any method call outside our analysis incurs at most one comparison and a short jump.

To measure the execution time of a method under test, we utilize the Read Time-Stamp Counter (`rdtsc`) instruction available on the `x86` architecture. `rdtsc` is a model specific register that increments after each CPU clock cycle. Every time the interpreter invokes a method, our instrumentation stores the latest value of `rdtsc` into the calling thread and increments a depth counter. If the same method enters again in a recursive call, the depth counter is incremented. If the method under test calls another method, it simply skips over our analysis code. Each time the method under test returns, the depth counter is decremented. If the depth counter is equal to zero, `rdtsc` is invoked and the computed difference between the current value and the old value stored inside the thread. We note that the measured execution time for the method under test consequently includes its own execution time and the execution time of all methods it invokes internally. This result is stored in- side the method under test's internal JVM data structure located in its class's constant pool. After
μFuzz invokes a method, it can query the last observed execution time for the method through the EyeVM's measurement API. Listing 1 presents a complete implementation of this instrumentation. These stubs are added to the appropriate sections in the JVM source code that generate and clean up stack frames for invoked methods.

The `java` executable used to run every Java program loads the HotSpot VM as a shared library into its process address space in order to run the JVM. Thus, EyeVM can export arbitrary symbols to support the JNI interface exposed to analysis tools. Currently, the EyeVM defines functions that

allow a process to configure the method under test, to poll its most recent execution time, and to clear its stored execution time. The EyeVM then simply uses JNI to bind methods on a given Java class to the native EyeVM functions that support our analysis. In particular, HotFuzz adds three methods to the `java.lang.reflect.Executable` class to support our analysis: `setMethodUnderTest`, `clearAnalysis`, and `getRuntime`.

EyeVM Tracing Mode. In addition to measuring method execution times, EyeVM allows an analyst to trace the execution of Java programs with method-level granularity. Traces provide valuable insight into programs under test and is used herein to evaluate HotFuzz detection capabilities. Each event comprising a trace represents either a method invocation or return. Invocation events carry all parameters the method is invoked on.

In principle, traces could be generated either by instrumenting the bytecode of the program under analysis, or through an external tool interface like the JVMTI. The former approach requires modifying the program under analysis, and the latter requires context switching between the JVM and an external agent that needs to be compiled and loaded into the JVM. As both of these approaches introduce significant overhead, we (as for measurement mode) opt instead for JVM-based instrumentation. That is, modifying the JVM directly to trace program execution is a simpler approach that does not require any modification on the program under analysis and only requires knowledge of internal JVM data structures.

EyeVM's tracing mode is implemented by instrumenting the bytecode interpreter generated at run-time by the HotSpot VM. Recall that the JVM executes bytecode within a generated assembly interpreter in the host machine's instruction set architecture. In order to generate program traces that record all methods invoked by the program under analysis, stubs are added to locations in the assembler interpreter that invoke and return from methods. We note that these are the same locations that are instrumented to implement measurement mode.

However, while performance overhead is an important factor, program execution tracing can nevertheless be effectively implemented in the C++ run-time portion of the JVM as opposed to generating inline assembly as in the measurement case. Then, during interpreter generation, all that is added to the generated code are invocations of the C++ tracing functions.

When a program under test reaches a trace recording point, the JVM is executing the generated bytecode interpreter implemented in x86-64 assembly. Directly calling a C++ function will lead to a JVM crash, as the machine layout of the bytecode interpreter differs from the Application Binary Interface (ABI) expected by the C++ run-time. Fortunately, the JVM provides a convenient mechanism to call methods defined in the C++ run-time using the `call_VM` method available in the Assembler API. `call_VM` requires that parameters to the C++ function are passed using general purpose registers. This facility is used to pass a pointer to the object that represents the method we wish to trace, a value that denotes whether the event represents an invocation or return, and a pointer to the parameters passed to the method under test. All of this information is accessible from the current interpreter frame when tracing an event. Additionally, before calling the C++ stub, the Top of the Stack State (`ToSState`) the JVM uses to understand where the top of the JVM stack is located must be saved. Thus, before switching to the C++ run-time, the `ToSState` is pushed onto the machine stack. The C++ tracing function is then called. Finally, after the function returns, the `ToSState` is restored back to its original value.

The trace event handler itself collects the name of the method under test and its parameters. The

name of the method is obtained from the method object the JVM passes to the stub. The parameters passed to the method under test are collected by accessing the stub parameters in similar fashion. The JVM's `SignatureIterator` class allows the tracing function to iterate over the parameter types specified in the method under test's signature, and, therefore, ensures that the correct parameter types are recorded. For each parameter passed to a method, both its type and value are saved. All of this information is batch-streamed to a trace file.

μFuzz. Micro fuzzing is implemented using a RabbitMQ message broker and a collection of μFuzz instances. Each μFuzz instance runs inside the EyeVM in measurement mode, consumes methods as jobs from a queue, and micro-fuzzes each method within its own process. Over time, micro- fuzzing methods in the same process might introduce side effects that prevent future jobs from succeeding. For example, a method that starts an applet could restrict the JVM's security policy and prevent μFuzz from performing benign operations required to fuzz future methods. This occurs because once a runner VM restricts its security policy it cannot be relaxed. To prevent this and similar issues from affecting future micro-fuzzing jobs, the following probe was added to every μFuzz instance. Prior to fuzzing each method received from the job queue, μFuzz probes the environment to ensure basic operations are allowed. If this probing results in a security exception, the μFuzz process is killed and a new one spawned in its place.

In order to prevent noise from perturbing the outcome of our experiments, we configure each μFuzz in the following way. Every μFuzz instance runs within the EyeVM in interpreted mode in order to maintain consistent run-time measurements for methods under test. Before micro-fuzzing a method, the μFuzz worker thread binds itself to an available CPU core. Each time μFuzz successfully invokes the method under test, it submits a test case for storage in the results database. Every test case generated by μFuzz consists of the input given to the method under test and the number of clock cycles it consumed when invoked on the input.

Exceptions are interpreted as a signal that an input is malformed, and therefore all such test cases are discarded. Ignoring input that causes the method under test to throw an exception restricts μFuzz's search space to that of valid inputs while it attempts to maximize resource consumption.

4.0 RESULTS AND DISCUSSION

We have extensively evaluated the HotFuzz component of the Continuum platform. In the following, we present two case studies involving the detection of zero-day vulnerabilities in real-world software as well as detection of STAC challenge program vulnerabilities.

4.1 Case Study: Detecting AC Vulnerabilities with IVI-based Inputs

Our evaluation revealed the surprising fact that 6 methods in the JRE contain AC vulnerabilities exploitable by simply passing empty values as inputs. Figure 2 shows the 6 utilities and APIs that contain AC vulnerabilities that an adversary can exploit.

The presence of AC vulnerabilities in internal APIs that handle low-level functionality such as bytecode rewriting and manifest files has the potential to create critical attack surfaces for DoS in any program that utilizes them.

4.2 Case Study: Arithmetic DoS in Java Math

As a part of our evaluation, HotFuzz detected 2 AC vulnerabilities inside the JRE's Math package. To the best of our knowledge, no prior CVEs document these vulnerabilities. We developed proof-of-concept exploits for these vulnerabilities and verified them across three different implementations of the JRE from Oracle, IBM, and Google. The vulnerable methods and classes provide abstractions called `BigDecimal` and `BigInteger` for performing arbitrary precision arithmetic in Java. Any Java program that performs arithmetic over instances of `BigDecimal` derived from user input may be vulnerable to AC exploits, provided an attacker can influence the value of the number's exponent when represented in scientific notation.

A successful exploit of `BigDecimal.add` in Oracle's JDK (versions 9 and 10) can run for over an hour even when Just-in-Time (JIT) compilation is enabled. On IBM's J9 platform, the exploit ran for 4 and a half months, as measured by the `time` utility, before crashing. When we exploit the vulnerability on the Android 8.0 Runtime (ART), execution can take over 20 hours before it ends with an exception when run inside an x86 Android emulator.

We reported our findings to all three vendors and received varying responses. IBM assigned a CVE [1] for our findings. Oracle decided to communicate the potential performance impact an exploit could have on its users in an upcoming security notice. Google argued that it does not fall within the definition of a security vulnerability for their platform.

HotFuzz automatically constructs valid instances of `BigDecimal` and `BigInteger` that substantially slow down methods in both classes. For example, simply incrementing `1e2147483648` by 1 takes over an hour to compute on Oracle's JDK even with Just-in-Time (JIT) Compilation enabled. HotFuzz finds these vulnerabilities without any domain-specific knowledge about the Java Math library or the semantics of its classes; HotFuzz derived all instances required to invoke methods by starting from the `BigDecimal` constructors given in the JRE.

The underlying issue in the JRE source code that introduces this vulnerability stems from how it handles numbers expressed in scientific notation. Every number in scientific notation is expressed as a coefficient multiplied by ten raised to the power of an exponent. The performance of arithmetic over these numbers in the JRE is sensitive to the difference between two numbers' exponents. This makes addition over two numbers with equal exponents, such as `1e2147483648` and `2e1e2147483648`, return immediately, whereas adding `1e2147483648` to `1e1`, can run for over an hour on Oracle's JVM.

After observing this result, we surveyed popular libraries that use `BigDecimal` internally, and developed proof of concepts that exploit this vulnerability as shown in Figure 3. We found that several general-purpose programming languages hosted on the JVM are vulnerable to this attack along with `org.json`, a popular JSON parsing library.

Developers face numerous security threats when they validate input values given as strings. The vulnerabilities we discussed in this section are especially problematic because malicious input is perfectly valid, albeit very large, floating point numbers. If a program performs any arithmetic over a `BigDecimal` object derived from user input, then it must take care to prevent the user from providing arbitrary numbers in scientific notation. Likewise, these results show that developers must be careful when converting between these two classes, as interpreting certain floating-point numbers as integers could suddenly halt their application. This complicates any input validation that accepts numbers given as strings. Our results reveal that failure to implement such validation correctly could allow remote adversaries to slow victim programs to a halt.

4.3 Case Study: DARPA STAC Challenges

The DARPA Space and Time Analysis for Cybersecurity (STAC) program contains a set of challenge programs developed in Java that test the ability of program analysis tools to detect AC vulnerabilities. In this case, study, we measure HotFuzz's ability to automatically detect AC vulnerabilities found in these challenges. We began by feeding the challenges into HotFuzz, which produced a set of test cases that exploit AC vulnerabilities in the challenges.

However, these test cases on their own do not answer the question of whether a given challenge is vulnerable to an AC attack, because challenges are whole programs that receive input from sources such as standard input or network sockets, and HotFuzz detects AC vulnerabilities at the method level. Therefore, given a challenge that is vulnerable to an AC attack, we need a way to determine whether one of its methods that HotFuzz marks as vulnerable is relevant to the intended vulnerability.

The STAC challenges provide ground truth for evaluating HotFuzz in the form of a proof-of-concept exploit. We define the following procedure for measuring the number of challenges HotFuzz can solve automatically. We start by executing each challenge that contains an intended vulnerability in the EyeVM, and execute the challenge's exploit on the running challenge. This produces a trace of every method invoked in the challenge during a successful exploit. If HotFuzz marks a method M as vulnerable in the output for challenge C, and M appears in the trace for C, we count the challenge as confirmed.

When we conducted this experiment on all the challenges contained in the STAC program vulnerable to AC attacks, we found that HotFuzz automatically marked 7 out of 49 challenges as vulnerable. This result demonstrates micro-fuzzing's effectiveness for detecting AC vulnerabilities in real-world software. The challenges given in the DARPA STAC program are large programs that utilize well-known, complex library dependencies such as Spring and various Apache Commons libraries. The scale of these programs renders applying any program analysis difficult. While a fully-automated analysis that solves all these challenges remains infeasible in general, micro-fuzzing provides a lightweight approach for detecting real security problems without the hard requirement of manual human analysis.

4.4 Other Scientific Results

In addition to the core analysis platform, the Continuum program has resulted in other high-impact research published in top scientific venues. In this section, we summarize several key publications.

4.4.1 Rampart. To degrade the performance of web servers, a common practice is to launch distributed denial-of-service attacks (DDoS) that flood the target system with numerous requests. Specifically, among other attacks, attackers might command thousands of computers (or more) to send attack traffic, or they might spoof the victim's IP address to launch reflected attacks. Fortunately, for defenders, these attacks incur comparatively high cost for the attackers (e.g., acquiring a large-size botnet to mount the attack) and they can often already be detected by state-of-the-art network- level defense mechanisms.

Unfortunately, sophisticated DoS attacks have gained significant traction recently. In these attacks, attackers use low-bandwidth, highly targeted, and application-specific traffic to overwhelm a target system. Different from traditional DDoS attacks that rely on flooding a victim system with

an extensive amount of traffic, sophisticated DoS attacks require less resources and utilize a lower volume of intensive requests to attack the victim system's availability. Specifically, attacker's target expensive or slow execution paths of the victim system. For example, an intensive attack might request the system to calculate computationally expensive hashes millions of times by specifying an unusually high iteration count for the bcrypt function. Particularly problematic is that sophisticated DoS attacks are difficult to detect by state-of-the-art defenses, such as source address filtering or trace back mechanisms, because those defenses were designed to mitigate large-scale network-layer DDoS attacks.

In the course of the Continuum project, we have designed and implemented a defense mechanism, Rampart, to protect a web application's back end from sophisticated DoS attacks. Rampart aims to mitigate attacks that overwhelm the available CPU resources (CPU time) of a web server through low-rate application- layer attack traffic, which we call CPU-exhaustion DoS attacks. Therefore, we design Rampart to accurately and efficiently detect and stop suspicious intensive attacks that may cause CPU exhaustion, and to be capable to block future attacks, without negatively affecting the application's availability for legitimate users.

Developing such, a defense is challenging. First, attack requests can blend in well with normal requests: Similar to requests sent by legitimate users, they also arrive at a low rate. Moreover, attack requests are generally well- formed, and, thus, do not cause the application to crash or throw an exception except for possibly resource exhaustion exceptions (e.g., a stack overflow exception). In turn, it is difficult to differentiate these two kinds of requests, i.e., it is non-trivial to block only attack requests without also incorrectly blocking legitimate requests. Since a legitimate request can be mistakenly labeled as suspicious, the defense system has to quickly detect and revoke any false positive filter that blocks legitimate requests, to not reduce the application's availability unnecessarily.

To address these challenges, we leverage statistical methods and fine-grained context-sensitive program pro- filing, which allows us to accurately detect and attribute CPU-exhaustion DoS attacks. Specifically, Rampart actively monitors all requests to precisely model the re- source usage of a web application at the function-level. It then dynamically builds and updates statistical execution models of each function by monitoring the runtime of the function called under different contexts. Upon arrival of a new request, the request is then constantly checked against the statistical models to detect suspicious deviation in execution time at runtime. Rampart lowers the priority of a request that it labeled as suspicious by aborting or temporarily suspending the application in- stance that is serving it, depending on the server's load. To prevent pollution attacks against the statistical models, Rampart collects only profiling measurements of normal requests that do not cause CPU-exhaustion DoS and that do not deviate much from the norm observed in the past. It also enforces a rate limit by network address.

Rampart can deploy filters to prevent future suspicious requests from over-consuming the server's CPU time. It employs an exploratory algorithm to tackle the problems of false positive requests and false positive filters. Specifically, when a true positive attack request is detected, a filtering rule is deployed to block similar suspicious requests, which might include legitimate requests (false positives). Rampart dynamically removes the deployed filter once the attack ends, to recover service for any legitimate users who might have been affected by the filter. Similarly, a false positive filter might be created if a legitimate request was incorrectly identified as suspicious. To not negatively impact an application's availability for future legitimate requests, Rampart periodically evaluates (explores) all generated filter policies and deactivates false positive filters. In turn, this algorithm allows Rampart to rapidly and intelligently discover false positive rules,

while simultaneously thwarting true attacks.

We designed Rampart as a general defense against CPU-exhaustion DoS attacks. Importantly, to be protected by Rampart, it is not necessary to modify a web application or its source code in any way. To emphasize the practicality of Rampart, we implemented a prototype of Rampart for PHP, which remains the most popular server-side programming language today. Moreover, we thoroughly evaluated our prototype implementation, and found that it incurs negligible performance overhead of less than an additional 3 ms for processing a request, i.e., roughly 0.1% of the median website load time.

Finally, we demonstrated that Rampart could effectively preserve the availability and performance of real world, non-trivial web applications when they are victim of CPU-exhaustion DoS attacks. We focused on two of the most popular open-source content management systems: Drupal and WordPress. For example, when launching known attacks without Rampart's protection, the average CPU usage increased from 32.21% to 95.05% for attacks on Drupal and from 42.21% to 94.14% for attacks on WordPress. However, if protected by Rampart, then the average CPU usage remained comparatively stable at no more than 39.62% for Drupal and 51.40% for WordPress.

4.4.2 HACRS. Traditionally, vulnerability discovery has been a heavily manual task. Expert security researchers spend significant time analyzing software, understanding how it works, and painstakingly sifting it for bugs. Even though human analysts take advantage of tools to automate some of the tasks involved in the analysis process, the amount of software to be analyzed grows at an overwhelming pace. As this growth reached the scalability limits of manual analysis, the research community has turned its attention to automated program analysis, with the goal of identifying and fixing software issues on a large scale. This push has been met with significant success, culminating thus far in the DARPA Cyber Grand Challenge (CGC), a cyber-security competition in which seven finalist teams pitted completely autonomous systems, utilizing automated program analysis techniques, against each other for almost four million dollars in prize money.

Despite the success of these systems, the underlying approaches suffer from a number of limitations. These limitations became evident when some of the CGC autonomous systems participated in a follow-up vulnerability analysis competition (the DEF CON CTF) that included human teams. The autonomous systems could not easily understand the logic underlying certain applications, and, as a result, they could not easily produce inputs that drive them to specific (insecure) states. However, when humans could provide "suggestions" of inputs to the automated analysis process the results were surprisingly good.

This experience suggested a shift in the current vulnerability analysis paradigm, from the existing tool-assisted human-centered paradigm to a new human-assisted tool-centered paradigm. Systems that follow this paradigm would be able to leverage humans (with different level of expertise) for specific well-defined tasks (e.g., tasks that require an understanding of the application is underlying logic), while taking care of orchestrating the overall vulnerability analysis process.

This shift is somewhat similar to introduction of the assembly line in manufacturing, which allowed groups of relatively unskilled workers to produce systems (such as cars) that had, until then, remained the exclusive domain of specially trained engineers. Conceptually, an assembly line "shaves off" small, easy tasks that can be carried out by a large group of people, in loose collaboration, to accomplish a complex goal.

As part of the Continuum project, we explored the application of this idea to vulnerability analysis. More precisely, we developed an approach that leverages task lets that can be dispatched to human analysts by an autonomous program analysis system, such as those used in the Cyber Grand Challenge, to help it surmount inherent drawbacks of modern program analysis techniques. We explored the question of how much our "program analysis assembly line" empowers humans, otherwise unskilled in the field, to contribute to program analysis, and we evaluated the improvement that external human assistance can bring to the effectiveness of automated vulnerability analysis. Our results are significant: by incorporating human assistance into an open-source Cyber Reasoning System, we were able to boost the number of identified bugs in our dataset by 55%, from 36 bugs (in 85 binaries) using fully-automated techniques to 56 bugs through the use of non-expert human assistance.

4.4.3 DIFUZE. Device drivers are an essential part in modern Unix-like systems to handle operations on physical devices, from hard disks and printers to digital cameras and Bluetooth speakers. The surge of new hardware, particularly on mobile devices, introduces an explosive growth of device drivers in system kernels. Many such drivers are provided by third-party developers, which are susceptible to security vulnerabilities and lack proper vetting. Unfortunately, the complex input data structures for device drivers render traditional analysis tools, such as fuzz testing, less effective, and so far, research on kernel driver security is comparatively sparse.

DIFUZE represents a novel combination of techniques to enable interface-aware fuzzing, facilitating the dynamic exploration of the `ioctl` interface provided by device drivers on UNIX-like systems. DIFUZE performs an automated static analysis of kernel driver code to recover their specific ioctl interface, including valid commands and associated data structures. It uses this recovered interface to generate inputs to ioctl calls, which can be dispatched to the kernel from userspace programs. These inputs match the commands and structures used by the driver, enabling efficient and deeper exploration of the ioctls. The recovered interface allows the fuzzer to make meaningful choices when mutating the data: i.e., typed fields like pointers, enums, and integers should not be handled as simply a sequence of bytes. DIFUZE stresses assumptions made by the drivers in question and exposes serious security vulnerabilities.

In our experiments, we analyzed seven modern mobile devices and found 36 vulnerabilities, of which 32 were previously unknown. Four vulnerabilities found by DIFUZE were patched during the course of our experiments (e.g., CVE-2017-0612), ranging in severity from flaws that crash the device in question causing Denial of Service (DoS) to bugs that can give the attacker complete control over the phone.

DIFUZE has been released as an open-source tool [7] in the hopes that it will be useful for other security researchers.

5.0 CONCLUSIONS

Work on the Continuum analysis platform and the security problems it targets continues. During the course of the program, numerous unpublished techniques were explored such as the TypeDB object synthesis database for automated Java test case generation or the path cost differential analysis for side-channel detection and characterization. These in particular continue to be investigated as future refinements of Continuum-based analyses.

While the STAC program focused on Java bytecode, our respective groups have substantial expertise in analyzing binary code. In particular, we are interested in more deeply incorporating the dynamic analyses we have developed as dynamic symbolic execution variants as part of the angr platform. This integration would likely greatly reduce the number of iterations between angr+java and HotFuzz required to discover future vulnerabilities, increasing the scalability of the platform.

A highly promising direction for the optimization-guided fuzzing efforts is the use of online AI/ML techniques to dynamically optimize fuzzing parameters in order to boost coverage and defect discovery rates. In preliminary work, we have applied this idea to AFL, using lightweight linear weighting to bias the application of individual mutation operators and different operator stack sizes in havoc mode. On our application test set, this has resulted in up to a 2x increase in edge coverage and unique crashes, with a median increase in the 1.5x range, compared to current unmodified AFL. Interestingly, the resulting learned "intensity" of mutation is heavily correlated with how constrained the target program's input format is. That is, for inputs that have many constraints such as an Office XML document, less mutation is more effective, while on other formats such as the TLS protocol, more mutation produces better coverage and crash results. In general, we believe that integrating lightweight machine learning with fuzz testing is likely to result in more significant improvements in the state-of-the-art in vulnerability discovery.

References

[1] *CVE-2018-1517*. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1517.

[2] Patrice Godefroid. "Micro execution". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 539–549.

[3] George Klees et al. "Evaluating Fuzz Testing". In: *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2123–2138. DOI: 10.1145/3243734.3243804. URL: http://doi.acm.org/10.1145/3243734.3243804.

[4] Eugene Kuleshov. "Using the ASM framework to implement common Java bytecode transformation patterns". In: *Aspect-Oriented Software Development* (2007).

[5] Kenneth B. Russell and Lars Bak. "The HotSpot Serviceability Agent: An Out-of-Process High-Level Debugger for a Java Virtual Machine". In: *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*. 2001, pp. 117–126. URL: http://www.usenix.org/publications/library/proceedings/jvm01/russell.html.

[6] The angr team. *angr*. URL: https://angr.io (visited on 2019).

[7] The DIFUZE team. *DIFUZE*. URL: https://github.com/ucsb-seclab/difuze (visited on 2019).

[8] *The JVM Tool Interface (JVM TI): How VM Agents Work*. https://www.oracle.com/technetwork/articles/javase/index-140680.html.

```
void Assembler::method_enter() {
    Label skip_measurement, increment_stack;
    push(rax);
    push(rbx);
    movptr(rax, Address(r15_thread, JavaThread::method_under_test_offset()));
    get_method(rbx);
    cmpptr(rax, rbx);
    jcc(Assembler::notEqual, skip_measurement);
    movl(rax, Address(r15_thread, JavaThread::analysis_depth_offset()));
    testl(rax, rax);
    jcc(Assembler::notZero, increment_stack);
    rdtsc();
    shlq(rdx, 32);
    orq(rdx, rax);
    movq(Address(r15_thread, JavaThread::analysis_started_offset()), rdx);
    bind(increment_stack);
    incrementl(Address(r15_thread, JavaThread::analysis_depth_offset()));
    bind(skip_measurement);
    pop(rbx);
    pop(rax);
}

void Assembler::method_exit() {
    Label skip_measurement;
    push(rax);
    push(rbx);
    movptr(rax, Address(r15_thread, JavaThread::method_under_test_offset()));
    get_method(rbx);
    cmpptr(rax, rbx);
    jcc(Assembler::notEqual, skip_measurement);
    decrementl(Address(r15_thread, JavaThread::analysis_depth_offset()));
    movl(rax, Address(r15_thread, JavaThread::analysis_depth_offset()));
    testl(rax, rax);
    jcc(Assembler::notZero, skip_measurement);
    rdtsc();
    shlq(rdx, 32);
    orq(rdx, rax);
    movq(rax, Address(r15_thread, JavaThread::analysis_started_offset()));
    subq(rdx, rax);
    movq(Address(rbx, Method::last_execution_time_offset()), rdx);
    bind(skip_measurement);
    pop(rbx);
    pop(rax);
}
```

Listing 1: The instrumentation in the EyeVM that records the method under test's execution time in clock cycles. We implement the instrumentation using the same Assembler API that defines the bytecode interpreters generated by the HotSpot VM at run-time.

```
classfile.Utility.replace("", "", "");

String xs[] = {"","","","","","",""};
m = new Manifest();
files = new File(new File(""),"");
m.addFiles(files, xs);

x = new DecimalFormat("");
x.toLocalizedPattern();

table = new LZWStringTable();
table.addCharString(0, 0);

byte y[] = {0, 0, 0};
il = new InstructionList(y);
ifi = new InstructionFinder(il);
ifi.search("");

int z[] = {0, 0, 0};
s = new SupplementaryCharacterData(y);
s.getValue(0);
```

Listing 2: Proof of concept exploits for AC vulnerabilities that require only IVI-based inputs to trigger.

```
clojure=> (inc (BigDecimal. "1e2147483647"))
clojure=> (dec (BigDecimal. "1e2147483647"))
groovy:000> 1e2147483647+1

scala> BigDecimal("1e2147483647")+1

JSONObject js = new JSONObject();
js.put("x", BigDecimal("1e2147483647"));
js.increment("x");
```

Listing 3: Proof of concept exploits that trigger inefficient arithmetic operations for the `BigDecimal` class in Clojure, Scala, Groovy, and the org.json library.