HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing

Anonymized for submission

***Abstract*—Fifteen billion devices run Java and many of them are connected to the Internet. As this ecosystem continues to grow, it remains an important task to discover the unknown security threats these devices face. Fuzz testing repeatedly runs software on random inputs in order to trigger unexpected program behaviors, such as crashes or timeouts, and has historically revealed serious security vulnerabilities. Contemporary fuzz testing techniques focus on identifying memory corruption vulnerabilities that allow adversaries to achieve remote code execution. Meanwhile, algorithmic complexity (AC) vulnerabilities, which are a common attack vector for denial-of-service attacks, remain an understudied threat.**

In this paper, we present HotFuzz, a framework for automatically discovering AC vulnerabilities in Java libraries. HotFuzz uses micro-fuzzing, a genetic algorithm that evolves arbitrary Java objects in order to trigger the worst-case performance for a method under test. We define Small Recursive Instantiation (SRI) which provides seed inputs to micro-fuzzing represented as Java objects. After micro-fuzzing, HotFuzz synthesizes test cases that triggered AC vulnerabilities into Java programs and monitors their execution in order to reproduce vulnerabilities outside the analysis framework. HotFuzz outputs those programs that exhibit high CPU utilization as witnesses for AC vulnerabilities in a Java library.

We evaluate HotFuzz over the Java Runtime Environment (JRE), the 100 most popular Java libraries on Maven, and challenges contained in the DARPA Space and Time Analysis for Cyber-Security (STAC) program. We compare the effectiveness of using seed inputs derived using SRI against using empty values. In this evaluation, we verified known AC vulnerabilities, discovered previously unknown AC vulnerabilities that we responsibly reported to vendors, and received confirmation from both IBM and Oracle. Our results demonstrate micro-fuzzing finds AC vulnerabilities in real-world software, and that micro-fuzzing with SRI derived seed inputs complements using empty seed inputs.

I. Introduction

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 enforce memory 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[1] behavior in the processing of that input, resulting in a denial of service. While in the textbook example against a hash table an adversary inserts values with colliding keys to degrade the complexity of lookup operations from an expected $O(1)$ to $O(n)$, the category of AC vulnerabilities is by no means hypothetical. Recent examples of algorithmic complexity vulnerabilities include denial of service issues in Go's elliptic curve cryptography implementation [4], an AC vulnerability that manifests through amplification of API requests against Netflix' internal infrastructure [13] triggered by external requests, and a denial of service situation in the Linux kernel's handling of TCP packets [3]. This vulnerability was considered serious enough that it was embargoed until OS vendors and large Linux users such as cloud providers and content delivery networks could develop and deploy patches. While these particular vulnerabilities involved unintended CPU time complexity, AC vulnerabilities can also manifest in the spatial domain for resources such as memory, storage, or network bandwidth.

While discovering AC vulnerabilities is notoriously challenging, program analysis seems like a natural basis for developing solutions that tackle such issues. In fact, prior research has started to explore program analysis techniques for finding AC vulnerabilities in software. Most of this work is based on manual or static analysis that scales to real world code bases, but focuses on detecting known sources of AC vulnerabilities, such as triggering DoS in programs that use regular expressions [29], [50], [55] or deserialize objects from strings [19].

Fuzz testing, where an external program feeds random input to a program under test until it crashes or times out, has historically revealed serious bugs that enabled Remote Code-Execution (RCE) in software such as OS Kernels, Mobile Devices, and Web Browsers. Recent work has also started adapting existing state-of-the-art fuzz testers such as AFL [57] and libFuzzer [7] to detect AC vulnerabilities in software. These include favoring inputs that maximize the length of an input's execution path through a program's Control Flow Graph (CFG) [42], incorporating multi-dimensional feedback so that fuzzing breaks out of local maxima [33], and augmenting AFL with symbolic execution to maximize a Java program's resource consumption [39].

Prior work for detecting AC vulnerabilities through fuzz testing re-purposes existing fuzz testers which were explicitly designed to detect crashing inputs as opposed to maximizing the resource utilization of the program under test. By doing so, they suffer the same limitations fuzz testers have at achieving code coverage. That is, a fuzzer can only reason about code that is actually executed, and potential vulnerabilities guarded by complex constraints on a program's inputs are commonly beyond reach for these systems. Prior work also limits the evaluation of new techniques to relatively small code bases with known performance problems. While restricting fuzzing to relatively smaller code bases simplifies comparing fuzzing

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

strategies, it also limits understanding the techniques that can discover previously unknown bugs hiding in real world software. In addition, measuring resource consumption by collecting statistics over the edges that an execution visits in a program's Control Flow Graph (CFG) ignores potential AC vulnerabilities caused by resource intensive code found in short execution paths. For example, a program that contains a loop that iterates a few times, but invokes a resource intensive system call every iteration, will appear less expensive compared to a program that contains a loop with more iterations, but does little work in each iteration.

In order to avoid reasoning about programs as a whole, this paper proposes *micro-fuzzing* (a concept analogous to micro-execution [22]) as a novel technique to balance coverage with in-depth exploration of a Java program's run-time resource usage. Fuzzing typically considers a whole program under test starting from its entry point with AFL, or a manually defined test harness around a library method with libFuzzer. In contrast, micro-fuzzing considers every method in a program or library as an entrypoint and attempts to automatically execute each one in isolation. This property is analogous to micro-execution, which supports executing code starting from arbitrary addresses in a program without requiring any manual effort. To this end, micro-fuzzing constructs program states represented as method inputs, directly invokes methods on those states, and measures the amount of resources those states consume using model specific registers available on the host machine.

This approach departs from traditional fuzzing methodologies that execute whole programs or functions on flat bitmaps as input and struggle to cover deeper paths in those program. Prior work attempts to cover deeper paths by augmenting fuzzing with symbolic execution [51], altering the program under test [41], or enriching the fuzzer's input domain with more expressive grammars over flat bitmaps [12]. In contrast to these approaches, micro-fuzzing invokes individual methods in a program with arbitrary state, that may include objects, and uses an evolutionary algorithm to evolve this state to detect an AC vulnerability in the method. Analogous to libFuzzer [7], micro-fuzzing attempts to overcome the coverage limitation in whole-program fuzzing by focusing on individual methods instead. However, in contrast to libFuzzer, micro-fuzzing neither optimizes for coverage or memory corruption, nor does it require instrumentation or human intervention to setup scaffolding to transform the flat bitmaps given by the fuzzing engine, into valid inputs for the method under test.

We implement micro-fuzzing for Java programs in HotFuzz, which uses a genetic algorithm to evolve method inputs with the goal to maximize method execution time. To generate initial populations of inputs, we devised two different strategies. The Identity Value Instantiation (IVI) strategy creates inputs by assigning each actual parameter the identity element of the parameter's domain (e.g., 0 for numeric types or "" for strings). In contrast, Small Recursive Instantiation (SRI) assigns parameters small values chosen at random from the parameter's domain. Irrespective of how inputs are instantiated, HotFuzz leverages the EyeVM, an instrumented JVM that provides run-time measurements at method-level granularity. If micro-fuzzing creates an input that causes the method under test's execution time to exceed a threshold, HotFuzz marks the method as potentially vulnerable to an AC attack. To validate potential AC vulnerabilities, HotFuzz synthesizes Java programs that invoke flagged methods on the suspect inputs and monitors their end-to-end execution in an unmodified JVM that mirrors a production environment. Those programs that exceed a timeout are included in HotFuzz's output corpus. Each program in this corpus represents a witness of a potential AC vulnerability in the library under test that a human operator can either confirm or reject.

We evaluate HotFuzz by micro-fuzzing the Java Runtime Environment (JRE), challenges provided by the DARPA Space and Time Analysis for Cyber-Security (STAC) program, and the 100 most popular libraries available on Maven, a popular repository for hosting Java program dependencies. We identify 2 intentional (in STAC) and 126 unintentional (in the JRE and Maven libraries) AC vulnerabilities.

In summary, this paper makes the following contributions:

- We introduce micro-fuzzing as a novel and efficient technique for identifying AC vulnerabilities in Java programs (see Section III-A).
- We devise two (IVI and SRI) strategies to generate seed inputs for micro-fuzzing (see Section III-A2b).
- We propose the combination of IVI and SRI with micro-fuzzing to detect AC vulnerabilities in Java programs.
- We design and evaluate HotFuzz, an implementation of our micro-fuzzing approach, on the Java Runtime Environment (JRE), challenges developed during the DARPA STAC program, and the 100 most popular libraries available on Maven. Our evaluation results yield previously unknown AC vulnerabilities in real-world software, including 20 in the JRE,106 across 67 Maven libraries, including the widely used org.json library, "solve" 2 challenges from the STAC program, and include confirmations from IBM and Oracle. (see Section V).

II. Background and Threat Model

In this section, we briefly describe algorithmic complexity (AC) vulnerabilities, different approaches to potentially detect such vulnerabilities, the threat model we assume, and the high-level design goals of this work.

A. AC Vulnerabilities

AC vulnerabilities arise in programs whenever an adversary can provide inputs that cause the program to exceed desired (or required) bounds in either the spatial or temporal domains. One can define an AC vulnerability in terms of asymptotic complexity (e.g., an input of size n causes a method to store $O(n^3)$ bytes to the filesystem instead of the expected $O(n)$), in terms of a concrete function of the input (e.g., an input of size n causes a method to exceed the intended maximum $150n$ seconds of wall clock execution time), or in other more qualitative senses (e.g., "the program hangs for several minutes"). However, in each case there is a definition, explicit or otherwise, of what constitutes an acceptable resource consumption threshold.

In this work, we do not assume that an explicit definition for this threshold exists and instead rely on domain knowledge and manual filtering of AC witnesses in order to label those that should be considered as true vulnerabilities. We believe that this is a realistic assumption and pragmatic method for vulnerability identification that avoids pitfalls resulting from

attempting to automatically understand intended resource consumption bounds, or from focusing exclusively on asymptotic complexity when in practice, as the old adage goes, "constants matter."

B. AC Detection Approaches

Software vulnerability detection in general can be roughly categorized as a static analysis, dynamic testing, or some combination of the two. Static analysis has been proposed to analyze a given piece of code for its worst case execution time behavior. While finding an upper bound to program execution time is certainly valuable, conservative approximations in static analysis systems commonly result in a high number of false positives. Furthermore, even manual interpretation of static analysis results in this domain can be challenging as it is often unclear whether a large worst-case execution time results from a property of the code or rather the approximation in the analysis. Additionally, static analyses for timing analysis commonly work best for well structured code that is written with such analysis in mind (e.g., code in a real-time operating system). The real-world generic code bases in our focus (e.g., Java's Runtime Environment), have not been engineered with such a focus and quickly reach the scalability limits of static timing analyzers.

Dynamic testing, in particular fuzz testing, has emerged as a particularly effective vulnerability detection approach that runs on the same production code deployed to servers and end users and can therefore run in parallel with the software development lifecycle [37], [49], [57]. Most existing fuzzers focus on detecting memory corruption exploits leading to program crashes, whereas AC exploits usually do not involve abnormal program termination. Furthermore, fuzzers aim to maximize coverage of a program under test which is fundamentally at odds with AC vulnerability detection since exploits usually involve forcing program execution to repeatedly invoke the same code either recursively or in a loop. As such, traditional fuzzers are not suitable for our problem domain. However, traditional fuzzers have the enticing property that crashing inputs are by definition witnesses of a software bug, and frequently these inputs can easily be used to reproduce a crash. Our work aims to fill the gap in existing fuzzers, by using scalable dynamic testing techniques to automatically discover potential AC vulnerabilities and produce witnesses that can be evaluated by developers or operators.

C. Optimization

The goal of identifying AC vulnerabilities boils down to a simple to posit yet challenging to answer optimization question. "What are concrete input values that make a given method under test consume the most resources?" One possible approach to tackle such optimization problems is with the help of genetic algorithms. A genetic algorithm emulates the process of evolution to derive approximations for a given optimization problem. To this end, a genetic algorithm will start with an initial population of individuals and over the duration of multiple generations repeatedly perform three essential steps: i) Mutation, ii) Crossover, and iii) Selection. In each generation, a small number of individuals in the population undergo mutation. Furthermore, each generation will see a large number of crossover events where two individuals combine to form offspring. Finally, individuals in the resulting population get evaluated for their fitness, and the individuals with the highest fitness are selected to form the population for the next generation. The algorithm stops after either a fixed number of generations, or when the overall fitness of subsequent populations no longer improves. In our scenario where we seek to identify AC vulnerabilities in Java methods, individuals correspond to the actual parameter values that are passed to a method under test. Furthermore, assessing fitness of a given individual can be accomplished by measuring the method's resource consumption while processing the individual (see Section IV-A). While mutation and crossover are straightforward to define on populations whose individuals can be represented as sequences of binary data, the individuals in our setting are tuples of well-formed Java objects. As such, mutation and crossover operators must work on arbitrary Java classes, as opposed to flat binary data (see Section III-A1).

D. Threat Model

In this work, we assume the following adversarial capabilities. An attacker either has access to the source code of a targeted program and its dependencies, or a compiled artifact that can be tested offline. Using this code, the attacker can employ arbitrary techniques to discover AC vulnerabilities exposed by the program, either in the program itself or by any library functionality invoked by the program. Furthermore, we assume that these vulnerabilities can be triggered by untrusted input.

E. Design Goals

The goal of our work is to discover AC vulnerabilities in Java code so that they can be patched before attackers have the opportunity to exploit them. In particular, we aim for an analysis that is automated and efficient such that it can run continuously in parallel with the development lifecycle on production artifacts. This gives developers insight into potential vulnerabilities hiding in their applications without altering their development workflow.

III. HotFuzz Overview

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 entrypoint for testing by a μ*Fuzz* instance. As opposed to traditional fuzzing, where the goal is to provide inputs that crash a program under test, here each μFuzz instance attempts to maximize the resource consumption of individual methods under test using genetic optimization over the method's 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.

Differences between the micro-fuzzing and realistic execution environments can lead to false positives. 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

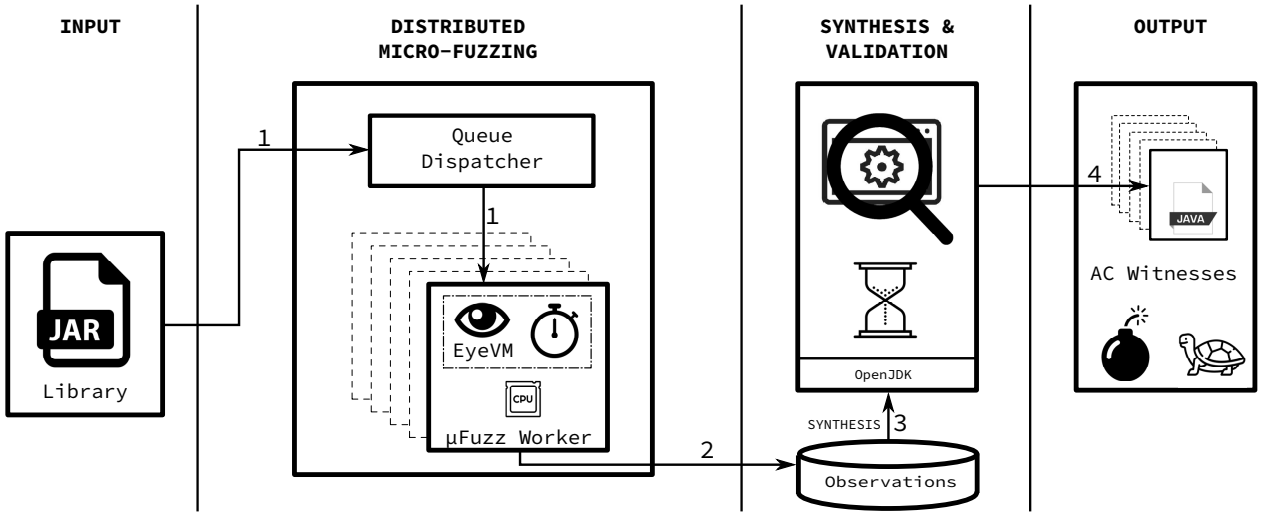


Fig. 1: Architectural overview of the HotFuzz testing procedure. In the first phase, individual μFuzz instances micro-fuzz each method comprising a library under test. Resource consumption is maximized using genetic optimization over inputs seeded using either IVI or SRI. In the second phase, test cases flagged as potential AC vulnerabilities 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 fail to do so are rejected as false positives, while HotFuzz reports those that do as validated AC vulnerability witnesses to a human analyst.

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 vulnerability that can then be examined by a human analyst. Otherwise, we discard the synthesized test case as a false positive.

Figure 1 depicts a graphical overview of the two phases. In the following, we motivate and describe the design of each component of the testing procedure in detail.

A. 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 bug as by definition all the constraints on the execution path that terminates in the bug 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 [22], micro-fuzzing constructs realistic intermediate program states, defined as Java objects, and directly executes individual methods on these states. Thus, we can cover all methods by simply enumerating all the methods that comprise a 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 pre-emptively 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. In this work, we consider the program's state as the inputs given to the methods we micro-fuzz. Modeling implicit parameters, such as files, static variables, or environment variables are outside the scope of this work.

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 resource 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 methods to the existing Reflection API in Java that toggles resource usage recording and associates measurements with Java methods. In practice, this involves non-trivial engineering, the details of which we present in Section IV. In the following, we describe how HotFuzz optimizes resource consumption during micro-fuzzing given dynamic measurements provided by the EyeVM, our instrumented JVM that provides run-time measurements at method-level granularity.

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

[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.

inputs over multiple generations until it detects abnormal resource consumption. Traditional fuzzers use evolutionary algorithms extensively, but in this work we present a genetic optimization approach to fuzzing that departs from prior work in two important ways. 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. Prior work for detecting AC vulnerabilities through fuzz testing either record resource consumptions using a combination of program instrumentation, CPU utilization, or counting executed instructions. In contrast, we record resource consumption using an altered execution environment (the EyeVM) and require no modification to the library under test. Second, traditional fuzzers treat inputs as flat bitmaps when genetic optimization (as opposed to more general mutation) is applied. 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, which can belong to arbitrary Java classes.

a) 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 and $(x_0, x_1)_0 = x_0$ and $(x_0, x_1)_1 = x_1$. To produce two new inputs, perform single-point crossover for each corresponding pair of values $(x_0, x_1) \in (X_0, X_1)$ 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 and produces the offspring as a pair. When x_0 and x_1 are objects, C_L and C_R recursively perform cross-over on every member attribute in (x_0, x_1) and select the left and right offspring, respectively. For example, consider a simple Java class `List` that implements a singly linked list. The `List` class consists of an `integer` attribute `hd` and a `List` attribute `tl`. Crossing an instance of `List` $\vec{x}$ with another instance $\vec{y}$ constructs two new lists $\vec{x'}$ and $\vec{y'}$ given by

$$\vec{x'} = C_L(\vec{x}, \vec{y}) = (hd := C(\vec{x}.hd, \vec{y}.hd)_0, tl := C_L(\vec{x}.tl, \vec{y}.tl))$$
$$\vec{y'} = C_R(\vec{x}, \vec{y}) = (hd := C(\vec{x}.hd, \vec{y}.hd)_1, tl := C_R(\vec{x}.tl, \vec{y}.tl))$$

b) Java Value 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 uniformly 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 the attribute a_i chosen from the object x. 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. In our current implementation, we only consider arrays of primitive types. The other sub-operators are defined in an intuitively similar manner. $M_{\text{mutate\_attr}}$ recursively applies the mutation operator M to the chosen attribute a_i when a_i is an object. After we obtain the mutated attribute a_i', we produce the mutated object x' by replacing a_i with a_i' in x.

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).

a) Identity Value Instantiation: Recent work has proposed guidelines for evaluating new fuzz testing techniques [30]. 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

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

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.

b) 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. For example, starting with large random numbers as seed inputs may waste time executing benign methods that simply allocate large empty data structures like Lists or Sets. For example, creating a List with the `ArrayList(int capacity)` constructor and passing it an initial capacity of `1<<30` takes over 1 second and requires over `4GB` of RAM. For this reason, we configure SRI with a

spread parameter α that limits the range of values from which SRI will sample. Thus, SRI is defined as

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

In the above, $R_{\mathsf{num}}(x,y)$ 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.

B. 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 behind this phase is to reproduce the behavior during fuzzing in a realistic execution environment using a real 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 during the first micro-fuzzing stage (see Section IV-B), a test case that exceeds the timeout in the first phase might not do so during validation when 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 a combination of the Reflection API and the `GSON` library. 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.

Recall HotFuzz takes compiled whole programs and libraries as input. Therefore, the witnesses contained in its final output corpus do not point out the source of any vulnerabilities in a program's source code. However, the EyeVM can trace the execution of any Java program running on it (see Section IV-A2). Given a witness of an AC vulnerability, we can trace its execution in the EyeVM in order to gain insight into the underlying causes of the problem in the program or library. In Section V, we use this technique to profile where an exploit found by HotFuzz spends most of its execution time.

IV. Implementation

In this section, we describe our prototype implementation of HotFuzz and discuss the relevant design decisions. Our prototype implementation consists of 5,487 lines of Java code, 1,007 lines of C++ code in the JVM, and 288 lines of Python code for validating witnesses detected by micro-fuzzing.

A. 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 micro-fuzzing procedure is compatible with any program or library that runs on the OpenJDK. 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 the `getRuntime()` method which we added to the existing `Executable` class in the OpenJDK. The `Executable` class allows μFuzz to obtain a Java object that represents the method under test and access our analysis data through our API. This API includes three methods to control and record our analysis: `setMethodUnderTest`, `clearAnalysis`, and `getRuntime`.

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*, described in Section IV-A1, and *tracing*, described in Section IV-A2. 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 the measurement mode to record the method under test's execution time during micro-fuzzing, while the tracing mode allows for manual analysis of the suspected AC vulnerabilities produced by HotFuzz.

1) EyeVM Measurement Mode: Commodity JVMs do not provide a convenient mechanism for recording method execution times. Prior work has made use of bytecode rewriting [32] 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 [46] or JVM Tool Interface [10] could be used, but 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 interprets Java programs represented in a bytecode instruction set documented by the JVM Specification [9]. During start up, the HotSpot VM generates a Template Table and allocates a slot for every instruction given in the JVM instruction set. Each slot contains a buffer of instructions in the host machine's instruction set architecture that interprets the slot's bytecode. The Template Interpreter inside the HotSpot VM interprets Java programs by fetching the Java instruction given at the Bytecode Pointer (BCP), finding the instruction's slot in the Template Interpreter's table, and jumping to that address in memory. The HotSpot VM interprets whole Java programs by performing this fetch, decode, execute procedure starting from the program's entrypoint which is given by a method called `main` in one of the program's classes.

During execution the Template Interpreter also relies heavily on functionality provided by HotSpot's C++ runtime.

The JVM developers define an Assembler API in the HotSpot source code that allows them to author C++ methods that, when executed, generate the native executable code required for each slot in the Template Interpreter. This allows a developer to implement the functionality for a given bytecode instruction, such as `iadd`, by writing a C++ method m. When the HotSpot VM starts up, it invokes m, and m emits as output native code in the host machine's instruction set architecture that interprets the `iadd` bytecode. HotSpot saves this native code to the appropriate slot so it can use it later to interpret `iadd` bytecode instructions. The API available to developers who author these methods naturally resembles the host's instruction set architecture. For example, if the two arguments to an `iadd` instruction reside in memory, a developer can call methods that will emit code to compute their sum. Instead of using this API to emit code that interprets JVM bytecodes, we use it to emit code that efficiently records methods' resource utilization for our analysis.

We instrument the JVM interpreter by augmenting relevant slots in the Template Interpreter using the same API that the JVM developers use to define the Template Interpreter. To measure execution time, we modify method entry and exit to store the method's elapsed time, measured by the `RDTSC` Model-Specific Register (MSR) available on the `x86` architecture, into thread-local data structures that analysis tools can query after a method returns. We limit our current implementation to the `x86-64` platform, but this technique can be applied to any architecture supported by the HotSpot VM. In addition, we could modify the Template Interpreter further to record additional resources, such as memory or disk consumption.

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.

Every time the interpreter invokes a method, our instrumentation stores the latest value of `rdtsc` into an attribute T_{start} in the calling thread and increments a depth counter T_{depth}. If the same method enters again in a recursive call, we increment T_{depth}. If the method under test calls another method, it simply skips over our analysis code. Each time the method under test returns, we decrement T_{depth}. If T_{depth} is equal to zero, `rdtsc` is invoked and the computed difference between the current value and T_{start} is stored inside the calling thread. Observe that the measured execution time for the method under test consequently includes its own execution time and the execution time of all the methods it invokes. This result is stored inside the method under test's internal JVM data structure located in its class's constant pool. The Assembler API available in the JVM sources supports all the functionality needed to implement these measurements, including computing the offsets of C++ attributes, manipulating machine registers, and storing values to memory.

Every time the JVM invokes a method, the Template Interpreter sets up a new stack frame for the called method. Likewise, when a method returns, the Interpreter removes the stack frame. The code that implements this logic is defined using the same Assembler API that defines the functionality for each JVM bytecode instruction. To record our resource measurements, we insert our relevant code snippets that start the measurement whenever a new stack frame is made, and save the measurement after the frame is removed.

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`.

2) 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 (see Section V). 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 the program under test, or through an external tool interface like the JVMTI. 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 does not require any modification of 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 Template Interpreter in the host machine's instruction set architecture. In order to generate program traces that record all methods invoked by the program under test, stubs are added to the locations in the Template 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.

To trace a program under test, we define a trace recording point as when the program either invokes a method or returns from one. When a method under test reaches a trace recording point the JVM is executing in the generated Template Interpreter represented 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. The JVM maintains an Operand Stack that it uses to hold inputs to methods and bytecode instructions. Internally, a special variable called the Top of the Stack State (ToSState) allows the JVM to check where the top of the Operand Stack is located. Before calling our C++ stub to trace an event, we push the current ToSState onto the machine stack. Next, we call our C++ tracing function. After the tracing function returns, we pop the ToSState off the machine stack and restore it to its original value.

The trace event stub itself collects the name of every invoked method or constructor, 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. Values of primitive types are represented as literals, whereas objects are represented by their internal ID in the JVM. Within the trace file, one can find the origin of a given ID from the object's constructor in the trace. All of this information is streamed to a trace file one event at a time.

B. μFuzz

Micro-fuzzing is implemented using a 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 running VM restricts its security policy, it cannot be loosened. To prevent this and similar issues from affecting future micro-fuzzing jobs, we add the following probe 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 is spawned in its place. Traditional fuzzers avoid these problems by forking before each test case so it can run in a fresh state. For a simple Java program that loops indefinitely, the JVM runs 16 Operating System threads. Constantly forking such a heavily multi-threaded environment on every test case introduces unnecessary complexity into our experiments.

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. If μFuzz runs with JIT enabled, our instrumentation no longer profiles the method under test, but rather the JVM's response to fuzzing the method under test. A JIT enabled JVM responds by compiling the bytecode that implements the method under test into equivalent native code in the host machine's instruction set architecture and executes it in a separate code cache. This causes the method under test's runtime to change dramatically during micro-fuzzing and skew our results. For this reason, we run μFuzz in the EyeVM in interpreted mode to ensure deterministic behavior.

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.

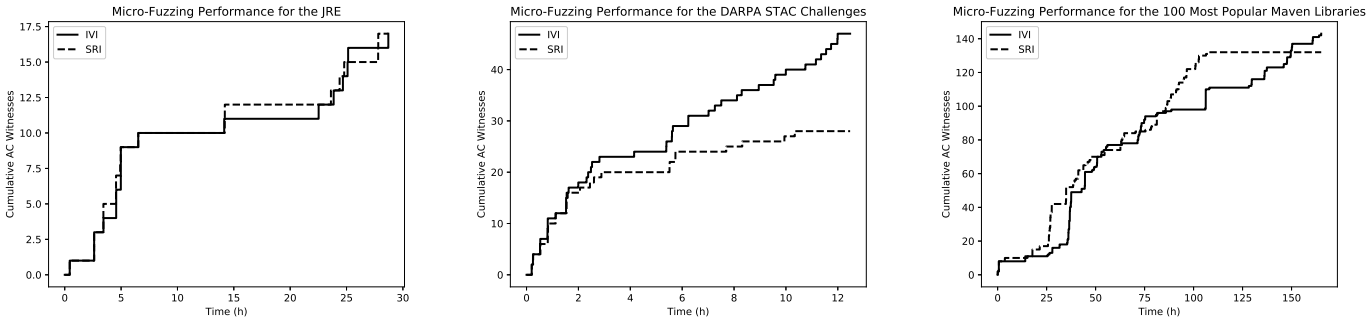
Exceptions are interpreted as a signal that an input is malformed, and therefore all such test cases are discarded. Ignoring input that causes the method under test to throw an exception restricts μFuzz's search space to that of valid inputs while it attempts to maximize resource consumption. In a different context, these test cases could be considered a potential attack vector for triggering DoS, but not due to an AC vulnerability.

V. Evaluation

In this section, we describe an evaluation of our prototype implementation of HotFuzz. This evaluation focuses on the testing procedure's efficiency in finding AC vulnerabilities in Java code, and additionally considers the effect of seed input instantiation strategy on micro-fuzzing efficiency. In particular, we define the performance of micro-fuzzing as the number of AC vulnerabilities it detects in a test artifact over time, and consider one strategy to outperform another if it detects more AC vulnerabilities given the same time period. In accordance with recently proposed guidelines for evaluating new fuzz testing techniques [30], we evaluate both "empty seed values" (IVI-based seeds) as well as SRI-based seeds.

We evaluate HotFuzz over the Java Runtime Environment (JRE), all challenge programs developed by red teams in DARPA's Space and Time Analysis for Cyber-Security (STAC) program, and the 100 most popular libraries available on Maven. This set of evaluation artifacts presents the opportunity to detect previously unknown vulnerabilities in real-world software as well as to validate HotFuzz on programs for which we have ground truth for AC vulnerabilities.

For the real-world software evaluation, we selected the JRE as it provides basic functionality utilized by every Java program. Given Java's widespread deployment across domains that range from embedded devices to high performance servers, any unknown algorithmic complexity vulnerabilities in the JRE present significant security concerns to programs that utilize those methods. For this reason, we evaluate HotFuzz over



(a) (b) (c)

Fig. 2: Measuring HotFuzz's performance while micro-fuzzing the Java Runtime Environment (JRE) (Figure 2a), all of the challenges given in the DARPA Space and Time Analysis for Cyber Security (STAC) Program (Figure 2b), and the 100 Most Popular Libraries found in Maven [8] (Figure 2c) using both IVI and SRI-derived seed inputs.

all methods in the JRE in order to measure its ability to detect unknown AC vulnerabilities in production software. Specifically, we consider JRE 1.8.0_181 from Java 8 as a library under test in our evaluation.

In addition to the JRE, Java programs frequently rely on libraries available through the popular Maven Repository, which as of 2019 hosts 15.1 million artifacts. In order to understand how vulnerable Maven's libraries are to AC attacks, we evaluate HotFuzz over the repository's 100 most popular libraries. A library's popularity on Maven is defined by the number of artifacts that include it as a dependency. For every Maven library we consider in our evaluation, we micro-fuzz every method contained in the library, and exclude the methods given in its dependencies.

a) Findings Summary: In conducting our evaluation, HotFuzz detected previously unknown AC vulnerabilities in the JRE, Maven Libraries, and discovered both intended and unintended AC vulnerabilities in STAC program challenges. Section V-A documents the experimental setup used to obtain these findings. Section V-B summarizes the impact of input seed generation strategy on micro-fuzzing performance, and provides several detailed case studies of micro-fuzzing results for the JRE, STAC challenges, and Maven libraries.

A. Experimental Set Up

We implement HotFuzz as a distributed system running within an on-premise Kubernetes cluster. The cluster consists of 64 CPUs and 256 GB of RAM across 6 Dell PowerEdge R720 Servers with Intel Xeon 2.4 GHz Processors. To micro-fuzz a given library under test, we deploy a set of μFuzz instances onto the cluster that consume individual methods from a work queue.

For each individual method under test, each μFuzz instance creates an initial population of inputs to the method, and runs a genetic algorithm that searches for inputs that cause the method under test to consume the most execution time. Recall that HotFuzz makes no assumptions about the code it fuzzes, and therefore it is critical to configure timeouts at each step of this process in order for the whole system to complete for all methods in a library under test. We configure the genetic algorithm inside each μFuzz instance with the recommended initial parameters for genetic algorithm experiments [20]. We use the same parameters for every method under test and do not tune these parameters for specific methods. We argue that this provides a generic approach to detecting AC vulnerabilities in arbitrary Java methods. Table I enumerates all of the parameters used to perform the evaluation.

To micro-fuzz each library under test, we created a pair of fuzzing jobs with identical parameters for each method contained in the library. Each pair consisted of a job that used the IVI seed input generation strategy, and the other used the SRI strategy with the α parameter listed in Table I. The parameters $(\psi, \lambda, \omega, \gamma)$ configured timeouts to ensure HotFuzz ran within a manageable time frame independent of the code under test, $(\pi, \chi, \tau, \epsilon, \nu)$ configured the genetic algorithm (GA) within HotFuzz, and σ configured the timeout used in the witness validation stage. Observe that we configured σ, the time required to confirm a witness as an AC vulnerability, to be half of ω, the time needed to detect a witness. Our intuition behind this choice is that a given test case will run much faster with JIT enabled than in our interpreted analysis environment, and hence the runtime required to confirm a witness is lower than the time required to detect it.

Given the definition of SRI presented in Section III-A2b, we use the following procedure to construct the initial population for a method under test M configured to use SRI in our evaluation. Given the parameter α, and the method under test M, μFuzz instantiates a full seed population of size π as follows. First, $\frac{\pi}{4}$ inputs are constructed using IVI, i.e., $S(M, 0)$. Another set of inputs are constructed as $\{S(M, i)\} \forall 0 < i \leq \frac{\pi}{4}$. Finally, the last $\frac{\pi}{2}$ inputs are constructed using $S(M, \alpha)$. We note that defining the population with a range of values for α allows μFuzz to explore inputs for M starting from the smallest possible set of values, while also including the full range of values allowed by the α parameter. Intuitively, we expect a method under test M to consume the fewest resources on the smallest input, and so we include this initial range so that micro-fuzzing always includes the smallest range of values.

The libraries under test that we consider for our evaluation are all 80 engagement articles given in the STAC program, and every public method contained in a public class found in the JRE and the 100 most popular libraries available on Maven. For the latter, we consider these public library classes and methods as the interface the library reveals to programs that utilize it. Therefore, this provides an ideal attack surface for us to micro-fuzz for AC vulnerabilities in its implementation. We emphasize that HotFuzz only requires as input the JAR files that make up a given library under test. HotFuzz does not require analysts to construct test harnesses around individual

TABLE I: The parameters given to every μFuzz instance. Multiple timeouts prevent HotFuzz from stopping because of individual methods that may be too complex to micro-fuzz efficiently.

Parameter	Definition	Value
α	The maximum value SRI will assign to a primitive type when constructing an object	256
ψ	The maximum amount of time to create the initial population	5s
λ	The time that may elapse betwen measuring the fitness of two method inputs	5s
ω	The amount of time required for a method to run in order to generate an AC witness	10s
γ	The wall clock time limit for the GA to evaluate the method under test.	60s
π	The size of the initial population	100
χ	The probability two parents produce offsprint in a given generation	0.5
τ	The probability an individual mutates in a generation	0.01
ϵ	The percent of the most fit individuals that carry on to the next generation	0.5
ν	The number of generations to run the GA	100
σ	The amount of time required for an AC witness to run outside the analysis framework in JIT mode in order to confirm it	5s

artifacts. Instead, HotFuzz automatically generates the test harness for every method it micro-fuzzes. This is in contrast to established fuzzing techniques [42], [33], [39] which require human effort to set up scaffolding for each fuzz target. To reproduce our evaluation, this would have to be done for approximately 400,000 individual methods.

B. Experimental Results

In our evaluation, HotFuzz detected 20 previously unknown AC vulnerabilities in the Java 8 JRE, detects both intended and unintended vulnerabilities in challenges from the STAC program, and detects 106 AC vulnerabilities in 67 libraries from the 100 most popular libraries found on Maven. Table II breaks down both the total wall-clock time HotFuzz spent micro-fuzzing the JRE, STAC engagement challenges, and Maven libraries for each seeding strategy and reports its throughput measured by the average test cases produced per hour. We define a test case to be a single input generated by HotFuzz for a method under test. Overall, micro-fuzzing with SRI derived seed inputs required more time to micro-fuzz the artifacts in our evaluation, but also produced more test cases overall.

1) Impact of Seed Input Generation Strategy: Table II presents the results of micro-fuzzing the JRE, all the challenges contained in the DARPA STAC program, and the 100 most popular libraries available on Maven using both IVI and SRI-derived seed inputs. Overall, micro-fuzzing with both strategies managed to invoke 24% of the methods contained in the JRE, 12% of the methods given in the STAC program, and 28% of the methods in the 100 most popular Maven libraries. As the results indicate, neither seeding strategy is categorically superior to the other. For example, when considering the results over the JRE, each strategy identifies 5 vulnerabilities that the other does not find, while another 12 vulnerabilities are identified by both strategies. While each strategy also finds vulnerabilities in the STAC artifacts that the other strategy cannot find, IVI seems to outperform SRI in terms of number of vulnerabilities found.

Initially, this result seems surprising. Recall that a fraction of the initial population created using SRI are identity values (i.e., $\alpha = 0$). Therefore, one would expect that the number of bugs detected by SRI-seeded micro-fuzzing to be a superset of the bugs detected by using IVI seeds. A closer analysis of the results pertaining to the micro-fuzzing of the STAC articles reveals that for *all* 21 bugs *confirmed* exclusively by IVI-based inputs, HotFuzz also derived inputs from the SRI population that were flagged as potential vulnerabilities in the first step. However, the resulting synthesized programs did not consume sufficiently many resources to cause a timeout in the second step of HotFuzz and were thus not confirmed. These results illustrate that the two seed generation strategies are *complementary* and that micro-fuzzing even if powered by a simple seeding strategy can expose serious availability vulnerabilities in widely-used software.

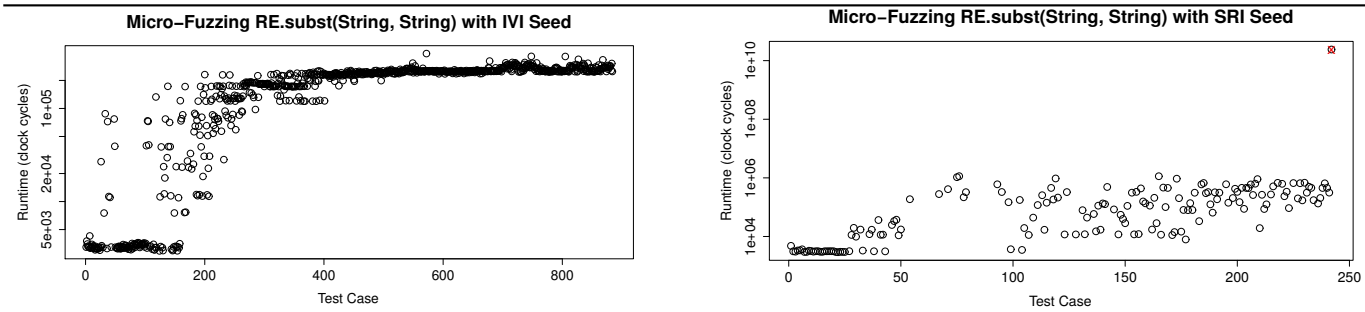
Figure 2 visually compares the performance of micro-fuzzing the JRE using both IVI and SRI-based seed inputs. From these results, we see that SRI inputs produce a marginal improvement over IVI inputs. That is, while both strategies eventually find the same number of vulnerabilities, SRI in aggregate finds them slightly faster.

Figure 3 provides a visual comparison between IVI-based and SRI-based micro-fuzzing on a regular expression method provided by the JRE. According to the documentation, the `RE.subst(String input, String sub)` method replaces all matches of the regular expression compiled by the `RE` instance on `input` with the value of `sub`. Figure 3a shows how micro-fuzzing this method using IVI-based seeds fails to arrive at a test case that demonstrates the vulnerability. In contrast, Figure 3b shows how using SRI-based seeds allows HotFuzz to detect the vulnerability. Additionally, we note that micro-fuzzing with SRI-derived seed inputs requires fewer test cases than micro-fuzzing with IVI-based seeds. When we traced the execution of the exploit found by HotFuzz in the EyeVM in tracing mode, we discovered that the method called the `StringCharacterIterator.isEnd` method from the `com.sun.org.apache.regexp.internal` package with alternating arguments indefinitely. We observed this exploit running for 12 days on a Debian system with Intel Xeon E5-2620 CPUs before stopping it. After reporting the issue to Oracle, they claimed it is not a security issue since the `RE.subst` method is protected and an attacker would have to perform a non-trivial amount of work to access it. That being said, the test case generated by HotFuzz is only 579 bytes in size and no method in the OpenJDK sources utilizes the `RE.subst` method outside of the OpenJDK test suite. This method appears to serve no purpose beyond providing a potential attack surface for DoS.

2) 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 6 shows the 6 utilities and APIs that contain AC vulnerabilities that an adversary can exploit. Upon disclosing our findings to Oracle they communicated that five of the six methods (lines 1-29) belong to internal APIs and that no path exists for malicious input to reach them. They recognized `DecimalFormat`'s

TABLE II: A comparison of fuzzing outcomes when evaluating HotFuzz on Java libraries using IVI and SRI seed inputs.

Library	Size	AC Witnesses Detected			AC Witnesses Confirmed			Methods Covered			Fuzzing Time (hrs)		Throughput (tests/hr)	
	No. Methods	Both	IVI	SRI	Both	IVI	SRI	Both	IVI	SRI	IVI	SRI	IVI	SRI
JRE	91632	12	5	5	10	5	5	20475	1051	617	33.9	30.4	2114869	2169852
DARPA STAC	67494	26	21	2	2	0	0	7928	275	255	12.0	12.5	1653175	1501118
Top 100 Maven Libraries	239777	73	37	30	49	30	27	65648	1030	1051	166.2	108.6	1748382	2705450



(a) IVI-derived inputs.

(b) SRI-derived inputs.

Fig. 3: Visualization of micro-fuzzing `com.sun.org.apache.regexp.internal.RE.subst(String input, String sub)` with IVI-based and SRI-based inputs. Micro-fuzzing with IVI-based inputs fails to detect a zero-day AC vulnerability in the JRE, while SRI-derived inputs detect the vulnerability correctly. We note that the y-axis for graph a is 5 orders of magnitude smaller than graph b. Graph b shows the test case that triggers the AC vulnerability in the upper-right-hand corner.

```
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");
```

Fig. 4: Proof of concept exploits that trigger inefficient arithmetic operations for the `BigDecimal` class in Clojure, Scala, Groovy, and the org.json library.

behavior (lines 31-34) as a functional bug that they will fix in an upcoming release. Oracle's assessment assumes that a malicious user will not influence the input of the public `DecimalFormat` constructor. Unless only string constants are passed to the constructor as input, this is a difficult invariant to always enforce.

3) 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 manually defined exploit on `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 considered this a Security-in-Depth issue and acknowledged our contribution in their Critical Patch Update Advisory [2]. 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 `1e0`, 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 4. We found

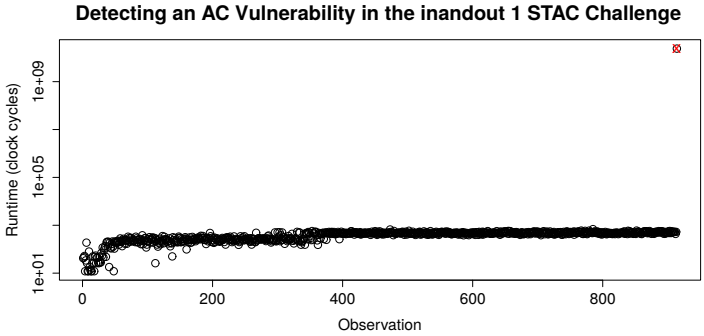


Fig. 5: Visualization of HotFuzz detecting an AC vulnerability found inside the inandout_1 challenge in the DARPA STAC Program. The test case that triggers the AC vulnerability in the challenge can be found in the upper-right-hand corner.

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) 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 corpus 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 proof-of-concept exploits on challenges with intended vulnerabilities. We define the following procedure to assess whether HotFuzz can detect an AC vulnerability automatically. We start by executing each challenge that contains an intended vulnerability in the EyeVM in tracing mode, 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 2 out of 37 challenges as vulnerable. One challenge, `inandout_1` provides a web service that allows users to order pizzas online. By running HotFuzz over this challenge, it identifies multiple methods with AC vulnerabilities in its code. Figure 5 visualizes HotFuzz detecting an AC vulnerability in the `PizzaParameters.subsequentEnergyOf2(int)` method found in the challenge. When we traced the execution of an exploit that achieved DoS against the pizza service, we observe the vulnerable method `subsequentEnergyOf2` identified by HotFuzz in the trace.

5) Case Study: Slow Parsing in org.json: Over the course of our evaluation HotFuzz detected a previously unknown AC vulnerability inside the popular org.json library. The vulnerable method, `JSONML.toJSONObject(String)` converts an XML document represented as a string into an equivalent JSONObject. This method is public, and instructions for its use in programs can be found in tutorials online [6]. Given the popularity of the org.json library on Maven, application developers may unknowingly expose themselves to DoS attacks by simply parsing XML strings into JSON.

Our experimental results obtained by micro-fuzzing the org.json library also demonstrated the utility of using SRI seed inputs over IVI seed inputs. While micro-fuzzing org.json in our evaluation, test cases evolved from IVI seed inputs failed to successfully invoke the `toJSONObject` method after 4,654 test cases. Meanwhile, the 96th SRI derived seed input successfully triggered the vulnerability. The second stage of our pipeline successfully validated this SRI test case represented as a 242 character string. After our evaluation completed, we took the PoC program generated by HotFuzz and observed it running for 60 hours on a Debian system with Intel Xeon E5-2620 CPUs in an unmodified Java environment with JIT enabled.

During our evaluation, HotFuzz started with no prior knowledge about org.json, JSON, or XML. Nonetheless, after simply passing the org.json library to HotFuzz as input and micro-fuzzing its methods using SRI derived seed inputs, we were able to uncover a serious AC vulnerability that exposes programs that depend on this library to potential DoS attacks. We have communicated our findings to the owners of the `JSON-java` project [5] who maintain the org.json library on Maven.

VI. Related Work

HotFuzz relates to previous work in four categories: *(i)* AC vulnerability analysis, *(ii)* test-case generation, *(iii)* fuzz testing, and *(iv)* resource analysis.

A. AC Vulnerability Analysis

Prior work for detecting AC vulnerabilities in Java programs includes static analysis on popular libraries [55], [29], [34], object-graph engineering on Java's serialization facilities [19], and exploiting worst-case runtime of algorithms found in commercial grade networking equipment [18]. On the Android platform, Huang et al. [27] use a combination of static and dynamic analysis to detect AC vulnerabilities within Android's System Server. Further up the application stack, Pellegrino et

```
import com.sun.org.apache.bcel.internal.*;

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

import java.io.File;
import sun.tools.jar.Manifest;

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

import com.sun.imageio.plugins.common.*;

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

import com.sun.org.apache.bcel.internal.*;

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

import sun.text.SupplementaryCharacterData;

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

import java.text.DecimalFormat;

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

Fig. 6: Proof of concept exploits for AC vulnerabilities that require only IVI-based inputs to trigger.

al. [40] identify common implementation mistakes that make web services vulnerable to DoS attacks. Finally, Holland et al. [26] proposes a 2 stage analysis for finding AC vulnerabilities.

Prior work for detecting AC vulnerabilities is custom-tailored to specific domains (e.g., serialization, regular-expression engines, Android Services, or web applications) and therefore often requires human assistance. HotFuzz differs from these approaches in that it is generically applicable to any Java program without human intervention, intuition, or insight.

B. Test Case Generation

Program analysis tools can generate test cases that exercise specific execution paths in a program and demonstrate the presence of bugs. Several tools perform symbolic execution within the Java Pathfinder [24] platform [28], [35], [58] in order to increase code coverage in Java test suites. Symbolic execution has found serious security bugs when applied to whole programs [38], [15] and in under-constrained settings [43] similar to HotFuzz. Toffola et al. [53] introduced PerfSyn which uses combinatoric search to construct test programs that trigger performance bottlenecks in Java methods.

C. Fuzz Testing

State of the art fuzz testers [57], [7] combine instrumentation on a program under test to provide feedback to a genetic algorithm that mutates inputs in order to trigger a crash. Active research topics include deciding optimal fuzzing seeds [45] and techniques for improving a fuzz tester's code coverage [16], [54]. Prior work has seeded fuzzing by replaying sequences of kernel API calls [23], commands from Android apps to smart IoT Devices [17], input provided by human assistants [48]. Recent techniques for improving code coverage during fuzz testing include selective symbolic execution [51], control- and data-flow analysis on the program under test [44], reducing collisions in code coverage measurements [21], and altering the program under test [41]. Prior work applies existing fuzz testers to discover AC vulnerabilities in whole programs [42], [33], and in Java programs by combining fuzz testing with symbolic execution [39] or seeding black box fuzzing with information taken from program traces [36]. In contrast, HotFuzz micro-fuzzes individual methods and uses a genetic algorithm on individual Java objects in order to find inputs to these methods that demonstrate the presence of AC vulnerabilities. This departs from prior approaches that restrict fuzzing inputs to flat bitmaps.

D. Resource Analysis

Recent interest in AC and side-channel vulnerabilities increased the focus on resource analysis research. In this area, Proteus [56] presented by Xie et al. and Awadhutkar et al. [11] study sensitive paths through loops that might represent AC vulnerabilities. Meanwhile, Kothary [31], [47] investigates human-machine interaction to improve program analysis for finding critical paths and side channels. In Comb [25], Holland et al. investigate how to improve computation of all relevant program behaviors.

Other resource-oriented static analyses have also been proposed [14], [52]. This line of work is based on statically inferred properties of programs and their resource usage. In contrast, HotFuzz provides quantitative measurements of program behavior over concrete inputs in a dynamic, empirical fashion.

VII. Conclusion

In this work, we present HotFuzz, a fuzz tester that detects algorithmic complexity (AC) vulnerabilities in Java libraries through a novel approach called *micro-fuzzing*. HotFuzz uses genetic optimization of test artifact resource usage seeded by Java-specific Identity Value and Small Recursive Instantiation (IVI and SRI) techniques to search for inputs that demonstrate AC vulnerabilities in methods under test. We evaluate HotFuzz on the Java 8 Runtime Environment (JRE), challenge programs developed in the DARPA Space and Time Analysis for CyberSecurity (STAC) program, and the 100 most popular libraries on Maven. In conducting this evaluation, we discovered previously unknown AC vulnerabilities in production software, including 20 in the JRE, 106 in 67 Maven Libraries, as well as both known *and* unintended vulnerabilities in STAC evaluation artifacts. Our results demonstrate that the array of testing techniques introduced by HotFuzz are effective in finding AC vulnerabilities in real-world software.

References

[1] Blinded for anonymous review. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-XXXX-YYYY.

[2] Blinded for anonymous review. https://www.oracle.com/technetwork/security-advisory/cpuxxxXXXX-XXXXXX.html.

[3] Cve-2018-5390. https://nvd.nist.gov/vuln/detail/CVE-2018-5390#vulnCurrentDescriptionTitle.

[4] Cve-2019-6486. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6486.

[5] Json-java. https://stleary.github.io/JSON-java/.

[6] JSONML Tutorials Point. https://www.tutorialspoint.com/org_json/org_json_jsonml.htm.

[7] libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/libFuzzer.html.

[8] Maven Repository. https://mvnrepository.com/.

[9] The Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se8/html/index.html.

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

[11] AWADHUTKAR, P., SANTHANAM, G. R., HOLLAND, B., AND KOTHARI, S. Intelligence amplifying loop characterizations for detecting algorithmic complexity vulnerabilities. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)* (Dec. 2017), vol. 00, pp. 249–258.

[12] BASTANI, O., SHARMA, R., AIKEN, A., AND LIANG, P. Synthesizing program input grammars. In *ACM SIGPLAN Notices* (2017), vol. 52, ACM, pp. 95–110.

[13] BEHRENS, S., AND PAYNE, B. Starting the avalanche: Application ddos in microservice architectures.

[14] CARBONNEAUX, Q., HOFFMANN, J., REPS, T. W., AND SHAO, Z. Automated resource analysis with coq proof objects. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II* (2017), pp. 64–85.

[15] CHA, S. K., AVGERINOS, T., REBERT, A., AND BRUMLEY, D. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA* (2012), pp. 380–394.

[16] CHA, S. K., WOO, M., AND BRUMLEY, D. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015* (2015), pp. 725–741.

[17] CHEN, J., DIAO, W., ZHAO, Q., ZUO, C., LIN, Z., WANG, X., LAU, W. C., SUN, M., YANG, R., AND ZHANG, K. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018* (2018).

[18] CZUBAK, A., AND SZYMANEK, M. Algorithmic Complexity Vulnerability Analysis of a Stateful Firewall. In *Information Systems Architecture and Technology: Proceedings of 37th International Conference on Information Systems Architecture and Technology - ISAT 2016 - Part II* (2016), pp. 77–97.

[19] DIETRICH, J., JEZEK, K., RASHEED, S., TAHIR, A., AND POTANIN, A. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain* (2017), pp. 10:1–10:32.

[20] EIBEN, A. E., AND SMITH, J. E. *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, 2015.

[21] GAN, S., ZHANG, C., QIN, X., TU, X., LI, K., PEI, Z., AND CHEN, Z. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA* (2018), pp. 679–696.

[22] GODEFROID, P. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering* (2014), ACM, pp. 539–549.

[23] HAN, H., AND CHA, S. K. IMF: inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), pp. 2345–2358.

[24] HAVELUND, K. Java pathfinder, A translator from java to promela. In *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings* (1999), p. 152.

[25] HOLLAND, B., AWADHUTKAR, P., KOTHARI, S., TAMRAWI, A., AND MATHEWS, J. Comb: Computing relevant program behaviors. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (New York, NY, USA, 2018), ICSE '18, ACM, pp. 109–112.

[26] HOLLAND, B., SANTHANAM, G. R., AWADHUTKAR, P., AND KOTHARI, S. Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on* (2016), IEEE, pp. 79–84.

[27] HUANG, H., ZHU, S., CHEN, K., AND LIU, P. From System Services Freezing to System Server Shutdown in Android: All You Need Is a Loop in an App. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015* (2015), pp. 1236–1247.

[28] JAYARAMAN, K., HARVISON, D., GANESH, V., AND KIEZUN, A. jFuzz: A Concolic Whitebox Fuzzer for Java. In *First NASA Formal Methods Symposium - NFM 2009, Moffett Field, California, USA, April 6-8, 2009.* (2009), pp. 121–125.

[29] KIRRAGE, J., RATHNAYAKE, A., AND THIELECKE, H. Static analysis for regular expression denial-of-service attacks. In *Proceedings of the International Conference on Network and System Security (NSS)* (Madrid, Spain, June 2013).

[30] KLEES, G., RUEF, A., COOPER, B., WEI, S., AND HICKS, M. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), pp. 2123–2138.

[31] KOTHARI, S., TAMRAWI, A., AND MATHEWS, J. Human-machine resolution of invisible control flow? In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)* (May 2016), pp. 1–4.

[32] KULESHOV, E. Using the asm framework to implement common java bytecode transformation patterns. *Aspect-Oriented Software Development* (2007).

[33] LEMIEUX, C., PADHYE, R., SEN, K., AND SONG, D. Perffuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018* (2018), pp. 254–265.

[34] LIVSHITS, V. B., AND LAM, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005* (2005).

[35] LUCKOW, K. S., DIMJASEVIC, M., GIANNAKOPOULOU, D., HOWAR, F., ISBERNER, M., KAHSAI, T., RAKAMARIC, Z., AND RAMAN, V. JDart: A Dynamic Symbolic Analysis Framework. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings* (2016), pp. 442–459.

[36] LUO, Q., NAIR, A., GRECHANIK, M., AND POSHYVANYK, D. FOREPOST: finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering 22*, 1 (2017), 6–56.

[37] MICROSOFT. Microsoft security risk detection. https://www.microsoft.com/en-us/security-risk-detection/.

[38] MOLNAR, D., LI, X. C., AND WAGNER, D. A. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings* (2009), pp. 67–82.

[39] NOLLER, Y., KERSTEN, R., AND PASAREANU, C. S. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018* (2018), pp. 322–332.

[40] PELLEGRINO, G., BALZAROTTI, D., WINTER, S., AND SURI, N. In the compression hornet's nest: A security study of data compression in network services. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. (2015), pp. 801–816.

[41] PENG, H., SHOSHITAISHVILI, Y., AND PAYER, M. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA* (2018), pp. 697–710.

[42] PETSIOS, T., ZHAO, J., KEROMYTIS, A. D., AND JANA, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), pp. 2155–2168.

[43] RAMOS, D. A., AND ENGLER, D. R. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*. (2015), pp. 49–64.

[44] RAWAT, S., JAIN, V., KUMAR, A., COJOCAR, L., GIUFFRIDA, C., AND BOS, H. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017).

[45] REBERT, A., CHA, S. K., AVGERINOS, T., FOOTE, J., WARREN, D., GRIECO, G., AND BRUMLEY, D. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. (2014), pp. 861–875.

[46] RUSSELL, K. B., AND BAK, L. 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.

[47] SANTHANAM, G. R., HOLLAND, B., KOTHARI, S., AND RANADE, N. Human-on-the-loop automation for detecting software side-channel vulnerabilities. In *Information Systems Security* (Cham, 2017), R. K. Shyamasundar, V. Singh, and J. Vaidya, Eds., Springer International Publishing, pp. 209–230.

[48] SHOSHITAISHVILI, Y., WEISSBACHER, M., DRESEL, L., SALLS, C., WANG, R., KRUEGEL, C., AND VIGNA, G. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the 2017 ACM Conference on Computer and Communications Security* (2017), ACM.

[49] SOURCE, G. O. Oss-fuzz. https://github.com/google/oss-fuzz.

[50] STAICU, C., AND PRADEL, M. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. (2018), pp. 361–376.

[51] STEPHENS, N., GROSEN, J., SALLS, C., DUTCHER, A., WANG, R., CORBETTA, J., SHOSHITAISHVILI, Y., KRUEGEL, C., AND VIGNA, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. Internet Society.

[52] TIZPAZ-NIARI, S., ČERNÝ, P., CHANG, B.-Y. E., SANKARANARAYANAN, S., AND TRIVEDI, A. Discriminating traces with time. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2017), A. Legay and T. Margaria, Eds., Springer Berlin Heidelberg, pp. 21–37.

[53] TOFFOLA, L. D., PRADEL, M., AND GROSS, T. R. Synthesizing programs that expose performance bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018* (2018), pp. 314–326.

[54] WOO, M., CHA, S. K., GOTTLIEB, S., AND BRUMLEY, D. Scheduling Black-Box Mutational Fuzzing. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013* (2013), pp. 511–522.

[55] WÜSTHOLZ, V., OLIVO, O., HEULE, M. J. H., AND DILLIG, I. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II* (2017), pp. 3–20.

[56] XIE, X., CHEN, B., LIU, Y., LE, W., AND LI, X. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, ACM, pp. 61–72.

[57] ZALEWSKI, M. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[58] ZHU, H. JFuzz: A Tool for Automated Java Unit Testing Based on Data Mutation and Metamorphic Testing Methods. In *2015 Second International Conference on Trustworthy Systems and Their Applications, TSA 2015, Hualien, Taiwan, July 8-9, 2015* (2015), pp. 8–15.