HotFuzz: Discovering Temporal and Spatial Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing

WILLIAM BLAIR, Boston University, United States
ANDREA MAMBRETTI, SAJJAD ARSHAD, MICHAEL WEISSBACHER,
WILLIAM ROBERTSON, and ENGIN KIRDA, Northeastern University, United States
MANUEL EGELE, Boston University, United States

Fuzz testing repeatedly assails software with random inputs in order to trigger unexpected program behaviors, such as crashes or timeouts, and has historically revealed serious security vulnerabilities. In this article, we present HotFuzz, a framework for automatically discovering Algorithmic Complexity (AC) time and space 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) as a technique to derive seed inputs represented as Java objects to micro-fuzzing. 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 fuzzing framework. HotFuzz outputs those programs that exhibit high resource 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 Cybersecurity (STAC) program. We evaluate SRI's effectiveness by comparing the performance of micro-fuzzing with SRI, measured by the number of AC vulnerabilities detected, to simply using empty values as seed inputs. 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 that micro-fuzzing finds AC vulnerabilities in real-world software, and that micro-fuzzing with SRI-derived seed inputs outperforms using empty values in both the temporal and spatial domains.

CCS Concepts: • **Security and privacy → Denial-of-service attacks**; **Vulnerability scanners**; **Software security engineering;**

Additional Key Words and Phrases: Fuzz testing, micro-fuzzing, dynamic analysis, algorithmic complexity, denial-of-service

A preliminary version of this paper appeared at NDSS'20 [18].

Authors' addresses: W. Blair and M. Egele, Boston University, 111 Cummington Mall, Boston, MA, 02215, United States; emails: {wdblair, megele}@bu.edu; A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, and E. Kirda, Northeastern University, 360 Huntington Ave, Boston, MA, 02115, United States; emails: mbr@ccs.neu.edu, arshad@iseclab.org, {mw, wkr, ek}@ccs.neu.edu.

2471-2566/2022/07-ART33 $15.00
https://doi.org/10.1145/3532184

ACM Reference format:
William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2022. HotFuzz: Discovering Temporal and Spatial Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. *ACM Trans. Priv. Secur.* 25, 4, Article 33 (July 2022), 35 pages.
https://doi.org/10.1145/3532184

1 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 AC vulnerabilities include denial of service issues in Go's elliptic curve cryptography implementation [7], an AC vulnerability that manifests through amplification of API requests against Netflix' internal infrastructure triggered by external requests [17], and a denial of service vulnerability in the Linux kernel's handling of TCP packets [5]. The vulnerability in the Linux kernel 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. For example, a recent vulnerability was discovered in Huawei devices that permits unauthenticated remote adversaries to leak memory by simply sending crafted messages to victim devices [9]. On desktop systems, attackers could exhaust kernel memory by taking advantage of an oversight in Shiftfs' fault handling code [11]. In some cases, attackers can even cause AntiVirus software to consume excessive memory and freeze a system by simply storing a specially crafted file [10]. While discovering AC vulnerabilities is notoriously challenging, program analysis seems like a natural basis for developing solutions to 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 worst-case performance of commonly used data structures [27], regular expression engines [42, 70, 79], or serialization APIs [29]. Fuzz testing, where a fuzzer feeds random inputs to a program under test until the program either crashes or times out, historically revealed serious bugs that permit **Remote Code-Execution** (**RCE**) exploits in widely used software such as operating system kernels, mobile devices, and web browsers. While the original fuzzers demonstrated widespread reliability issues across UNIX utilities [52], modern fuzzers like AFL [81], and libFuzzer [13] have become important components of the software development lifecycle because of their ease of use and the speed at which they discover software faults. Recent work has adapted these state-of-the-art fuzzers to automatically slow down programs with known

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

performance problems. These approaches include favoring inputs that maximize the length of an input's execution in a program's **control flow graph** (**CFG**) [59], incorporating multi-dimensional feedback that provides AFL with more visibility into the portions of the CFG each test case executes the most [47], augmenting AFL with symbolic execution in order to maximize a Java program's resource consumption [41, 54], or using a genetic algorithm over programs in a **domain specific language** (**DSL**) in order to trigger a Java method's worst-case runtime [76]. These recent advances demonstrate that modern fuzzers can automatically slow down programs such as sorting routines, hash table operations, and common UNIX utilities. Furthermore, identifying allocation functions via static analysis in order to incorporate memory usage into a program's CFG [77] have been shown to outperform state-of-the-art fuzzers for detecting AC space vulnerabilities that crash programs by exhausting available memory. These recent developments present exciting new directions for fuzz testing beyond detecting memory corruption bugs. However, these approaches do not reconcile the traditional fuzzing objective function of maximizing code coverage (breadth) with the opposing goal of maximizing a given program or individual method's runtime (depth). Indeed, these tools are evaluated by the slowdown or memory usage they can achieve for a given program or function, as opposed to the amount of code they successfully cover. Achieving high code coverage on any program under test is a notoriously difficult task because common program patterns like comparing the input to magic values or checksum tests are difficult to bypass using fuzzing alone, although program transformation tricks like splitting each comparison into a series of one-byte comparisons [46] or simply removing them from the program [58] can improve coverage. Augmenting fuzzing with advanced techniques like taint analysis [62] or symbolic execution [55, 71] helps overcome these fuzzing roadblocks, and RedQueen [14] showed how advanced tracing hardware can emulate these more heavyweight techniques by providing a fuzzer with enough information to establish correspondence between program inputs and internal program state. Prior work has successfully shown fuzz testing can reproduce known AC vulnerabilities in software, and research continues to produce innovative ways to maximize code coverage. What is missing in fuzzing for AC vulnerabilities are techniques to automatically sanitize a program's execution for AC vulnerabilities, analogous to how modern fuzzers rely on sanitizers to detect memory corruption bugs [67]. Current fuzzing approaches in general lack the ability to automatically fuzz programs at the method level without the need for manually defined test harnesses. Recent work has started automating the creation of test harnesses for existing fuzzers with the goal of creating enough valid states to successfully invoke individual methods either through statically analyzing library control flow graphs [39] or by obtaining valid method calls by examining the client code that uses a library method [16]. While this represents exciting new directions for automatically generating test harnesses, they nonetheless require access to high-quality code that correctly uses the method under test. This article proposes *micro-fuzzing* (a concept analogous to micro-execution [32]) as a novel technique to automatically construct test harnesses that allow a fuzzer to invoke methods and sanitize their execution for AC vulnerabilities in both time and space. Both AFL and libFuzzer can fuzz individual methods, but only after an analyst manually defines a test harness that transforms a bitstream into the types required to call a method. For AFL this involves defining a C program that reads the bitstream from standard input, whereas libFuzzer passes the bitstream to a specific function that it expects will call the method under test with the appropriate types derived from the bitstream. In contrast, micro-fuzzing takes whole programs or libraries as input and attempts to automatically construct a test harness for every function contained in the input. Observe that this approach is analogous to micro-execution [32], which executes arbitrary machine code by using a virtual machine as a test harness that provides state on-demand in order to run the code under test. To this end, micro-fuzzing constructs test harnesses represented as function inputs, directly invokes functions on those inputs, and measures the amount of time and

space each input consumes using a combination of model specific registers available on the host machine and an altered execution environment. This alleviates the need to define test harnesses manually and supports fuzzing whole programs and libraries by considering every function within them as a possible entrypoint. Furthermore, we sanitize every function's execution so that once its observed runtime or memory usage crosses configured thresholds, we kill the micro-fuzzing process and highlight the function as vulnerable. This sanitization highlights functions with potential AC vulnerabilities out of all the functions micro-fuzzing automatically executes, as opposed to measuring a fuzzer's ability to automatically slow-down individual programs or functions.

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 and space consumption. Java provides an ideal platform for evaluating micro-fuzzing because of its wide use across different domains in industry and the JVM's support for introspection allows HotFuzz to automatically generate test harnesses, represented as valid Java objects, for individual methods dynamically at runtime. To generate initial populations of inputs, we devise 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. We use IVI for the sole purpose of providing a baseline for measuring the effectiveness of using SRI to generate seed inputs for micro-fuzzing, based on recent recommendations for evaluating new fuzz testing techniques [43]. Irrespective of how inputs are instantiated, HotFuzz leverages the EyeVM, an instrumented JVM that provides run-time and memory measurements at method-level granularity. If micro-fuzzing creates an input that causes the method under test's execution time or memory consumption 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 or exhaust the JVM's memory are included in HotFuzz's output corpus. Every program contained in the output corpus represents a witness of a potential AC vulnerability in the library under test that a human operator can either confirm or reject. Sanitizing method execution for AC vulnerabilities based on resource thresholds mimics the sanitizers used by modern fuzzers that kill a process whenever an integrity violation occurs at runtime, but it also introduces false positives into our results given that it is difficult to configure a proper timeout that detects only true positives. In our evaluation, we show that the number of bugs detected by our sanitizer is concise enough to permit manual analysis of the results when micro-fuzzing for both AC time and space bugs. We evaluate HotFuzz by micro-fuzzing the **Java Runtime Environment** (**JRE**), challenges provided by the DARPA **Space and Time Analysis for Cybersecurity** (**STAC**) program, and the 100 most popular libraries available on Maven, a popular repository for hosting Java program dependencies. We identify 13 intentional (in STAC) and 217 unintentional (in the JRE and Maven libraries) AC vulnerabilities.

In summary, this article makes the following contributions:

— We introduce micro-fuzzing as a novel and efficient technique for identifying AC time and space vulnerabilities in Java programs (see Section 3.1).
— We devise two strategies (IVI and SRI) to generate seed inputs for micro-fuzzing (see Section 3.1.2).
— We propose the combination of IVI and SRI with micro-fuzzing to detect AC time and space vulnerabilities in Java programs.
— We design and evaluate HotFuzz, an implementation of our micro-fuzzing approach, on the 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 52 (26 Time and 26 Space) in the JRE, 165 (132 Time and 33 Space) across 47 Maven libraries, including the widely used org.json and ASM libraries in addition to the Clojure runtime, "solve" 13 challenges (five Time and eight Space) from the STAC program, and include confirmations from IBM and Oracle. In addition, micro-fuzzing with SRI-derived seed inputs outperforms IVI-derived seed inputs, measured by the number of AC witnesses detected, when micro-fuzzing with respect to both time and space (see Section 5).

2 BACKGROUND AND THREAT MODEL

In this section, we briefly describe AC vulnerabilities, different approaches that detect such vulnerabilities, the threat model we assume, and the high-level design goals of this work.

2.1 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 temporal or spatial 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 assume an explicit definition of this threshold independent of a given program under analysis and 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." We define an AC witness to be any input that causes a specific method under test's resource consumption to exceed a configured threshold. In this work, we consider AC witnesses that consume excessive time as well as memory. We consider any method that has an AC witness to contain an AC vulnerability. We recognize that this definition of an AC vulnerability based on observing a method's resource consumption exceeding some threshold will inevitably cause some false positives, since the chosen threshold may not be appropriate for a given method under test. Section 3 presents a strategy for minimizing false positives by automatically reproducing AC vulnerabilities in a production environment outside our fuzzing framework. This step may fail to remove all false positives, and in our evaluation given in Section 5, we show that the output of this validation stage is concise enough to allow an analyst to manually triage the results. Since we make no assumption about the methods we test in our analysis, we argue that output that consists of less than four hundred concise test cases after micro-fuzzing a hundred libraries is reasonable for a human analyst to consume.

2.2 AC Detection

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., the Java 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 continuously in parallel with the development lifecycle [51, 69]. State-of-the-art fuzzers detect bugs by automatically executing a program under test instrumented with sanitizers until the program either crashes or times out. A sanitized program crashes immediately after it violates an invariant enforced by the sanitizer, such as writing past the boundary of a buffer located on the stack or reading from previously freed memory. Once a fuzzer generates a test case that crashes a given sanitized program under test, the test case is a witness to a memory corruption bug in the original program. Since memory corruption bugs may be extended into exploits that achieve Remote Code Execution or Information Disclosure, fuzzers offer an effective and automated approach to software vulnerability detection. When source code is not available, a fuzzer can still attempt to crash the program under test in either an emulated or virtualized environment. Fuzz testing's utility for detecting memory corruption bugs in programs is well known, and current research explores how to maximize both the amount of code a fuzzer can execute and the number of bugs a fuzzer can find. Unfortunately, defining a sanitizer that crashes a process after an AC vulnerability occurs is not as straightforward as detecting memory integrity violations. This is in part because what constitutes an AC vulnerability heavily depends on the program's domain. For example, a test case that slows down a program by several milliseconds may be considered an AC vulnerability for a low latency financial trading application and benign for a web service that processes requests asynchronously. In this work, we sanitize for AC vulnerabilities in HotFuzz with respect to both time and space. HotFuzz sanitizes for AC time vulnerabilities by killing a process after a method's runtime exceeds a configured threshold. HotFuzz can also sanitize for AC space vulnerabilities by killing a process after a method's memory consumption exceeds a separate threshold or throws an out of memory exception. Like sanitizers that detect memory corruption bugs, this allows us to save only those test cases that exhibit problematic behavior. The drawback is that we do not have absolute certainty that our test cases are actual bugs in the original program and risk highlighting test cases as false positives. Building a fuzzing analysis that does not introduce any false positives is notoriously difficult, and fuzzers that detect memory corruption bugs are not immune to this problem. For example, Aschermann et al. [14] point out that previous evaluations erroneously report crashing inputs that exhaust the fuzzer's available memory as bugs in the original program under test. Furthermore, sanitizers point out many different sources of bugs including stack-based overflows, use after free, use after return, and heap-based overflows. While the presence of any of these bugs is problematic, triaging is still required to understand the problem given in a test case.

2.3 Fuzzing AC

SlowFuzz [59] and PerfFuzz [47] adapt two state-of-the-art fuzzers, libFuzzer, and AFL, respectively, and demonstrate the capability to automatically slow down individual programs or methods implemented in C/C++. Parallel developments also showed frameworks built on top of AFL can successfully slow down programs in interpreted languages as well [54]. Memlock [77] extends the PerfFuzz approach with static analysis to provide the fuzzer visibility into how individual execution paths consume memory in order to better generate inputs that exhaust memory and crash a program under test.

HotFuzz departs from these previous works by automatically creating test harnesses during micro-fuzzing, and sanitizing methods' execution for AC vulnerabilities. For example, PerfFuzz relies on fine grained coverage information (e.g., AFL's approximate edge coverage [81] and the *performance map*) to derive inputs that slow a program down. In contrast, HotFuzz relies on course-grained resource measurements to detect AC vulnerabilities across whole libraries. The former is far more detailed, and could likely allow HotFuzz to cover more library code and detect additional AC vulnerabilities. However, after implementing AFL's approximate edge coverage in HotFuzz, we observed a 55× decline in throughput while micro-fuzzing the JRE. For this reason, we used course grained measurements in order to obtain an efficient evaluation. In addition to quickly micro-fuzzing large libraries, HotFuzz does not require an analyst to manually define a test harness in order to fuzz individual methods contained in a library. This key feature differentiates micro-fuzzing found in HotFuzz from how AFL or libFuzzer fuzz individual methods. Since AFL and LibFuzzer only consider test cases consisting of finite bitstreams, one can fuzz an individual method with AFL by defining a test harness that transforms a bitstream read from `stdin` into function inputs, and with libFuzzer an analyst implements a `C` function that takes the bitstream as input and must transform it into the types needed to invoke a function. Observe that this must be done manually, whereas HotFuzz examines the type signature of the method under test and attempts to generate the test harness automatically. To reproduce our evaluation using existing tools, we would need to manually define approximately 360,000 individual test harnesses for the artifacts contained in our evaluation. Alternatively, a single test harness that receives a finite bitstream from the fuzzer could be made around each library. In this setting, a prefix in the bitstream selects the method, and the remainder of the bits derive the method's input which could be passed to constructors. This approach would fail to generate objects that trigger security issues with corrupted attributes that normal constructors would never produce (see Section 5.2.7). The input bits could also represent serialized objects, which may require providing the fuzzer seed inputs either manually or by authoring input generators [56]. In our early prototypes, we observed that traditional bitstream mutation strategies quickly corrupted Java objects' strict binary format in memory and caused the JVM to throw exceptions. It is also difficult to statically determine how many input bits will be needed to randomly create a method's input. This implies HotFuzz's instantiation strategies may require generating new bits from the finite bitstream passed to the test harness. This complicates propagating bitstream mutations to the generated Java objects. In contrast, HotFuzz directly alters Java objects that all share a common ancestry with a corpus of automatically-derived seed objects. Finally, the test harness would also need to restrict sanitizers to the chosen method under test and implement timeouts that prevent false positives from slowing micro-fuzzing (see Section 5). Prior works such as SlowFuzz and PerfFuzz both explore how fuzzers can automatically slow down individual programs and methods. Understanding what techniques work best to slow down code is necessary to understand how to design a fuzzer to detect AC time vulnerabilities. SlowFuzz observed that using the number of executed instructions as a test case's fitness in libFuzzer's genetic algorithm can be used to slow down code with known performance problems, such as sorting routines and hash table implementations. PerfFuzz went a step further and showed how incorporating a *performance map* that tracks the most visited edges in a program's CFG helps a fuzzer further slow down programs. Memlock adapts the fuzzer's goal of maximizing code coverage to consuming more memory in order to detect AC space vulnerabilities. These approaches take important steps needed to understand what techniques allow fuzzers to automatically slow down arbitrary code in order to spot AC vulnerabilities in programs. At the same time, they lack three important properties for being used to detect unknown AC vulnerabilities. First, they require manually defined test harnesses in order to fuzz individual functions. Second, these fuzzing engines only consider bitstreams as input to the programs under test and miss the opportunity

to evolve the high-level classes of the function's domain in the fuzzer's genetic algorithm. Third, these tools are evaluated primarily by how they successfully slow down code or cause code to consume memory, as opposed to sanitizing individual method executions for AC vulnerabilities.

2.4 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 may 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 4.1). 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 3.1.1).

2.5 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. An adversary can achieve DoS attacks on programs and services that utilize vulnerable libraries by taking the information they learn about a library through offline testing and developing exploits that trigger the AC vulnerabilities contained in library methods used by a victim program. For example, an adversary could take the test cases produced by our evaluation (see Section 5) and attempt to reproduce their behavior on programs that utilize the methods. Determining whether an adversary can transform these test cases into working AC exploits on victim programs is outside the scope of this work.

2.6 Design Goals

The goal of our work is to discover AC vulnerabilities in Java code with respect to both time and space 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 software development lifecycle on production artifacts. This gives developers insight into potential vulnerabilities hiding in their applications without altering their development workflow.

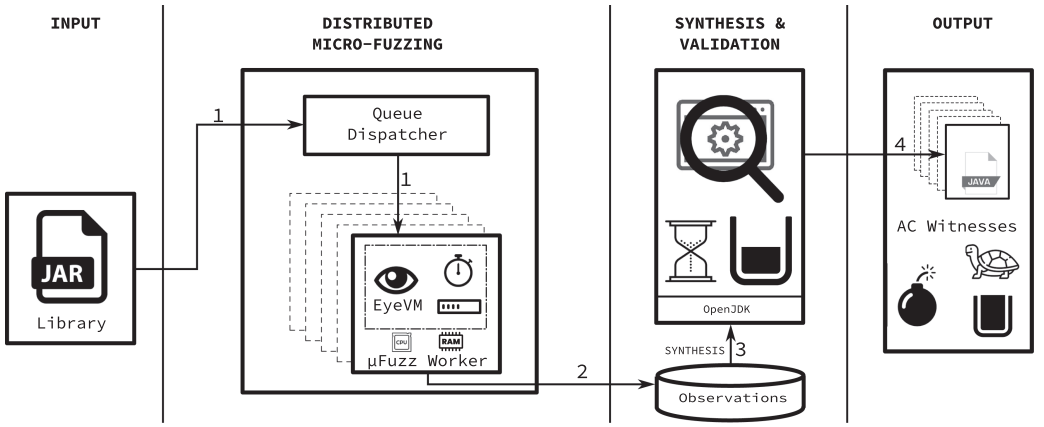


Fig. 1. Architectural overview of the HotFuzz testing procedure. In the first phase, individual μFuzz instances micro-fuzz each method comprising a library under test. Resource consumption is maximized using genetic optimization over inputs seeded using either IVI or SRI. In the second phase, test cases 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. HotFuzz reports those programs that pass the Synthesis and Validation stage as AC vulnerability witnesses to a human analyst.

3 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 *µFuzz* 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: IVI and SRI. Method-level resource consumption when executed on these inputs is measured using a specially-instrumented Java Virtual Machine (JVM) 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, and therefore reduce the number of false positives in our final results. 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.

3.1 Micro-Fuzzing

Micro-fuzzing represents a drastically different approach to vulnerability detection than traditional automated whole-program fuzzing. In the latter case, inputs are generated for an entire program either randomly, through mutation of seed inputs, or incorporating feedback from introspection on execution. Whole-program fuzzing has the significant benefit that any abnormal behavior—i.e., crashes—that is observed should be considered as a real 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 tradeoff between coverage and path satisfiability. Inspired by the concept of micro-execution [32], 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 at preemptively warning 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 that complements whole program fuzzing approaches. 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 balance between maximizing a method's resource utilization *in addition* to coverage. Conceptually speaking, implementing 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 4. 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 and memory consumption measurements at method-level granularity.

3.1.1 Resource Consumption Optimization. HotFuzz's fuzzing component, called μFuzz, is responsible for optimizing the resource consumption of methods under test. To do so, μFuzz uses genetic optimization to evolve an initial set of seed inputs over multiple generations until 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 consumption using a combination of program instrumentation, CPU utilization, counting executed instructions, or measuring memory allocated by designated functions. In contrast, we record resource consumption using an altered execution environment (the EyeVM) and require no modification to the library under test. Second, traditional

[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.

fuzzers treat inputs as bitstreams 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 a small probability. Since μFuzz operates on Java objects rather than bitstreams, we must define new crossover and mutation operators specific to this domain as bitstream-specific operators do not directly translate to arbitrary Java objects, which can belong to arbitrary Java classes.

Java Value Crossover. Genetic algorithms create new members of a population by "crossing" existing members. When individual inputs are represented as bitstreams, a standard approach is a single-point crossover: a single offset into two bitstreams is selected at random, and two new bitstreams 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 a 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

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

In this example, we show how HotFuzz crosses over a `List` that holds `integers`, but if the type of value stored in the `hd` attribute were a complex class T, the crossover operator would recursively apply crossover to every attribute stored in T.

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 bitstream-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

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

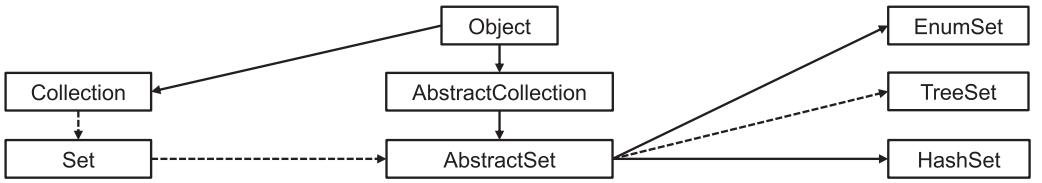


Fig. 2. A random walk over a CHG finds a concrete class for the `Collection` interface.

Each mutation sub-operator above operates on the attribute a_i chosen from the object x. For example, $M_{\text{flip\_bit}}$ selects a bit at random 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. When an attribute is a class, as opposed to a primitive type or a string or array, mutation utilizes the $M_{\text{mutate\_attr}}$ operator. $M_{\text{mutate\_attr}}$ 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.

3.1.2 Seed Generation. Given suitable crossover and mutation operators, all that remains to apply standard genetic optimization is the definition of a procedure to generate seed inputs. First, we define a **Class Hierarchy Graph** (**CHG**) that allows HotFuzz to find the concrete classes that implement any abstract classes and interfaces needed to invoke a method under test. In order to instantiate each concrete class, we define two procedures that we describe below: IVI, and SRI.

Class Hierarchy Graph. Every object in an object-oriented language like Java is assigned a corresponding class that allows the runtime environment to understand the attributes contained within the object and the methods available that may be called on the object. While micro-fuzzing a given method under test, HotFuzz may have to instantiate an abstract class or interface given in the method's signature. These classes are difficult to instantiate because they provide no constructors and the concrete classes available to HotFuzz depends heavily on the classes available in a program's environment. For this reason, we propose constructing a CHG as a pre-processing step to make all the concrete classes that inherit a given abstract class or interface available to HotFuzz for instantiating. To support constructing the CHG, we only require a binary relation S such that $A\ S\ B$ holds whenever a class B either inherits A directly or through multiple intermediate classes. Note that we can represent the CHG as the transitive closure S^* which encodes all the classes reachable from a given class. This allows HotFuzz to consult S^* in order to obtain a suitable concrete class B that can be instantiated and used to represent any abstract class or interface A. For example, Figure 2 provides a simplified version of S^* that shows several concrete implementations of a `Collection` reachable from both abstract classes and interfaces. If a given method under test requires an instance of a `Collection`, HotFuzz performs a random walk starting from the `Collection` node until it encounters a concrete class. A random walk over the CHG helps ensure that seed inputs for the method under test contain a diverse number of classes. In this example, the random walk is illustrated by the dotted line and shows how HotFuzz selects `TreeSet` as the concrete class to represent the requested `Collection` interface. In our prototype implementation of HotFuzz, we avoid explicitly constructing S^* by utilizing Java's ability to dynamically query relationships between classes at runtime. The details of our implementation can be found in Section 4.2 and the benefits obtained by micro-fuzzing using a CHG can be found in Section 5.2.1.

Identity Value Instantiation. Recent work has proposed guidelines for evaluating new fuzz testing techniques [43]. One of these guidelines is to compare any proposed strategy for constructing

seed inputs for fuzz testing with "empty" seed inputs. Intuitively, empty seed inputs represent the simplest possible seed selection strategy. Since empty bitstreams 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. While seed inputs produced by IVI will all be identical modulo consulting the CHG, micro-fuzzing will lead to a diverse population with time as the genetic algorithm applies mutation at random to introduce new features, and crossover leads to related offspring.

Small Recursive Instantiation. In addition to IVI, we define a complementary seed input generation procedure called 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 `4 GB` 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

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

In the above, $R_{\text{num}}(x, y)$ selects a value at random from the range $[x, y)$, while R_{char} produces a character obtained 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. We note that HotFuzz can be configured with different random number distributions to alter the behavior of R.

3.2 Witness Synthesis

Test cases exhibiting abnormal resource consumption are forwarded from the micro-fuzzing phase of the testing procedure to the second phase: witness synthesis and validation. The rationale behind this phase is to reproduce the behavior during fuzzing in a realistic execution environment using a real JVM in order to avoid false positives introduced due to the measurement instrumentation. In principle, one could simply interpret any execution that exceeds the configured timeout or allocates memory beyond the configured threshold 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, performing some other benign activity, or the micro-fuzzing environment simply exhausted its heap. An

additional consideration is that because the EyeVM operates in interpreted mode during the first micro-fuzzing stage (see Section 4.3), a test case that exceeds the timeout or memory threshold in the first phase might not do so during validation when **Just-in-Time** (**JIT**) compilation 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 by using a combination of the Reflection API and the Google `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 time vulnerability and recorded in serialized form in a database. For AC space vulnerabilities, we limit the size of the Java heap to the configured memory threshold and confirm a witness if the synthesized program throws an out of memory exception. 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 root cause 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 4.1.2). 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 5, we use this technique to discover the root cause for several AC bugs detected by HotFuzz. We acknowledge that filtering false positives may hide true positives that could be discovered through additional micro-fuzzing. Since HotFuzz stores all test cases in a database, an analyst can always use results from the first stage to inform future micro-fuzzing campaigns.

4 IMPLEMENTATION

In this section, we describe our prototype implementation of HotFuzz and discuss the relevant design decisions. Our prototype implementation consists of 5,824 lines of Java code, 1,007 lines of C++ code in the JVM, and 300 lines of Python code for validating AC witnesses detected by micro-fuzzing.

4.1 EyeVM

The OpenJDK includes the HotSpot VM, an implementation of the JVM, and the libraries and toolchains 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 analysis data through our API. This API includes four methods to control and record our analysis: `setMethodUnderTest`, `clearAnalysis`, `getRuntime`, and `getMemory`. 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 4.1.1, and *tracing*, described in Section 4.1.2. In measurement mode, the EyeVM records program execution time with method-level granularity, while the 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 resource usage during micro-fuzzing, while the tracing mode allows for manual analysis of the suspected AC vulnerabilities produced by HotFuzz.

4.1.1 EyeVM Measurement Mode. Commodity JVMs do not provide a convenient mechanism for recording method execution times or the amount of memory consumed by an individual method. Prior work has made use of bytecode rewriting [45] for this purpose. However, this approach requires modifying the test artifact, and produced non-trivial measurement perturbation in our testing. Furthermore, accurately measuring resource usage by correctly rewriting the bytecode of the large code bases given in our evaluation, which includes the entire JRE along with popular Maven libraries, represents a significant undertaking that deviates from our research goal of detecting AC vulnerabilities. Alternatively, an external interface such as the Serviceability Agent [65] or JVM Tool Interface [2] could be used, but these approaches introduce costly overhead due to the context switching incurred whenever 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 [1]. During start-up, the HotSpot VM generates a Template Table and allocates a slot in this table 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, and 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 heavily relies on functionality provided by HotSpot's C++ runtime. The HotSpot source code contains an Assembler API that allows JVM developers 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. One can think of this Assembler API as a C++ library that resembles an assembler like GNU `as`. For example, if the two arguments to an `iadd` instruction reside in memory, a developer can call methods on this API to load the values into registers, add them together, and store the result on the JVM's operand stack. We use this API 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 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 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

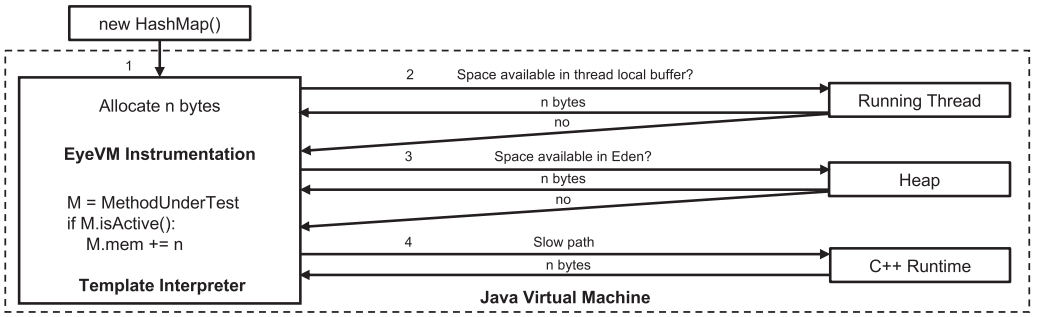


Fig. 3. The different ways the JVM may allocate an object, and how the EyeVM tracks memory usage.

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, the EyeVM invokes `RDTSC` 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' constant pool. In addition to tracking the number of clock cycles spent executing the method under test, we also instrument all of the memory allocation routines within the JVM to track the method under test's memory usage. Figure 3 breaks down the different strategies the JVM may take in allocating the bytes that back a new object. The fastest way to allocate memory is to reserve space within a buffer located within the running thread. If this thread local buffer is full, or if too many bytes are requested, the JVM attempts to reserve space within the heap maintained by its generational garbage collector. Within this heap, new objects are born into the Eden space reserved for objects in their first generation. If this allocation fails, the JVM takes the slow path which causes the Template Interpreter to switch to the JVM's C++ runtime to allocate the object. In order to accurately track the method under test's memory consumption, the EyeVM instruments each of these three allocation paths in order to increment a counter stored within the method object. This counter cumulatively records all memory allocated by the method under test. The EyeVM does not track deallocation performed by the JVM's garbage collector since our primary goal is to measure the amount of memory a method under test will consume when given specific inputs. In our experiments, we did not see false positives caused by garbage collector activity since every witness is validated in a production environment before being presented to an analyst. Whenever a program allocates memory in the EyeVM, our instrumentation first checks to see whether the method under test is active on the call stack. If so, it records the allocation and updates the corresponding object that represents the method under test. This is a necessary

performance optimization, as we found simultaneously tracking the memory consumption of all methods quickly slowed down our experiments. Furthermore, heap profiling tools like JProfiler [73] allow analysts to explore the entire Java heap of a running program, but they are difficult to use for quickly obtaining memory measurements just for the method under test within micro-fuzzing's genetic algorithm. 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 in memory. Every time the JVM invokes a method, the Template Interpreter sets up a new stack frame for the method which the interpreter removes after the method returns. The code that implements this logic is defined using the same Assembler API that implements each JVM bytecode instruction. To record our resource measurements, we insert relevant code snippets into the Template Interpreter that run every time the EyeVM adds or removes a stack frame. The `java` executable that runs every Java program begins by loading the HotSpot VM as a shared library into its process address space in order to run the JVM. Thus, the EyeVM can export arbitrary symbols to expose a JNI interface to analysis tools implemented in Java. Currently, the EyeVM defines functions that allow a process to configure the method under test, poll the method's most recent resource consumption, and clear the method's stored analysis data. The EyeVM then simply uses the JNI to bind the methods we added to the `Executable` Java class to the native EyeVM functions that support our analysis.

4.1.2 EyeVM Tracing Mode. In addition to measuring method execution times, the EyeVM allows an analyst to trace the execution of Java programs with method-level granularity. Tracing provides valuable insight into programs under test and is used herein to evaluate HotFuzz's ability to detect AC vulnerabilities (see Section 5). Each event given in a trace represents either a method invocation or return. Invocation events carry all parameters passed to the method as input. In principle, traces could be generated either by instrumenting the bytecode of 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. The 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 in this state 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. The `call_VM` method requires that parameters passed to the C++ function are contained within 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 holds 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 a 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 tracing records the correct parameter types. 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. All of this information is streamed to a trace file one event at a time.

4.2 Class Hierarchy Graph (CHG)

We implement the CHG by using the `Class.isAssignableFrom` method available in the Java runtime as the relation S presented in Section 3.1.2. This method returns true if the input class can be considered a subclass of the class on which the method was invoked. This allows us to explicitly construct the transitive closure S^* which we store in a `redis` database running alongside HotFuzz. In order to ensure fast retrieval of the CHG during micro-fuzzing, we store every concrete class B that is assignable from a class A under A's slot in the database. In practice, it is sufficient to store a fraction of S^* in the database since HotFuzz only queries classes that are assignable from abstract classes or interfaces. Before micro-fuzzing begins, we populate S^* by enumerating all pairs of classes A and B, and store B under A in the CHG if B is concrete and B is assignable from A. During micro-fuzzing, HotFuzz needs a way to traverse S^* from the information stored in the `redis` database. Given an abstract class or interface A, HotFuzz computes the number of classes stored under A using the `LLEN` command. HotFuzz then computes a random value $i \xleftarrow{R} [0, LLEN(A))$ and then attempts to instantiate the class given at `LINDEX(i)`. If HotFuzz is unable to instantiate the class given at index i, a new index is computed at random which repeats the process until HotFuzz successfully instantiates a class, or exhausts the available classes.

4.3 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 running each test case so that the fuzzer can fall back to its original state once the test case finishes,

crashes, or times out. 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 and very quickly destabilizes our experiments. We configure each μFuzz instance in the following way to prevent non-determinism present in the JVM from introducing unnecessary noise into our experiments. Every μFuzz instance runs within the EyeVM in an interpreted mode in order to maintain consistent run-time measurements for methods under test. If μFuzz runs with JIT enabled, our measurement instrumentation no longer profiles the method under test, but rather the JVM's response to fuzzing the method under test. A JVM with JIT enabled 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 would skew our results. For this reason, we run μFuzz in the EyeVM in interpreted mode to ensure consistent measurements. Upon receiving a method under test, μFuzz queries the CPUs available by obtaining the process' CPU affinity mask with `sched_getaffinity`. μFuzz then calls `sched_setaffinity` to pin the thread running the method under test to the lowest CPU given in the affinity mask. This confines the method under test to a single CPU for the duration of micro-fuzzing and also requires that every μFuzz instance have access to at least two CPUs, one for the method under test, and the remainder for the JVM's own threads. 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 consumes when invoked on the input. μFuzz interprets exceptions 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.

5 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 libraries in both time and space, 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 detected in a test artifact over time, and consider one strategy to outperform another if the strategy detects more AC vulnerabilities given the same time budget. In accordance with recently proposed guidelines for evaluating new fuzz testing techniques [43], we evaluate our proposed seed selection strategy (SRI) by comparing the performance of micro-fuzzing with SRI-based seeds to micro-fuzzing with "empty seed values" (IVI-based seeds). To the best of our knowledge, including existing fuzzers in our evaluation that adapt AFL [41] and libFuzzer [61] to Java programs to detect AC vulnerabilities would require significant engineering effort both in terms of instrumentation and designing test harnesses around our artifacts. Furthermore, the results those tools have achieved on real-world code bases like the JRE or Java libraries appear limited to individual methods and find bugs that crash the method under test with a thrown exception, or individual challenges from the STAC program. For these reasons, we exclude those tools from our evaluation. We evaluate HotFuzz over the JRE, all challenge programs developed by red teams in the DARPA STAC program, and the 100 most popular libraries available on the Maven repository. 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 AC vulnerabilities in the JRE present significant security concerns to programs that utilize those methods. For this reason, we evaluate HotFuzz over 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 2022 hosts 28 million artifacts. While this repository provides a convenient way to download an application's library dependencies, it also introduces any unknown vulnerabilities hiding within them into an application. 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 contained in its dependencies.

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 5.1 documents the experimental setup used to obtain these findings. Section 5.2 summarizes how the seed input generation strategy impacts micro-fuzzing performance, and provides several detailed case studies of micro-fuzzing results for the JRE, STAC challenges, and Maven libraries.

5.1 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 consumes individual methods from a message broker. 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 either the most execution time or memory. Detecting both AC time and space vulnerabilities requires that we micro-fuzz each artifact in our evaluation separately in order to optimize for each domain. In addition to the method under test, every job submitted to HotFuzz requires configuration parameters given in Table 1. Recall that HotFuzz makes no assumptions about the code under test, and therefore it is critical to configure timeouts at each step of this process in order for the whole system to complete all methods in a library under test. For this reason, the parameters $(\psi, \lambda, \omega, \gamma)$ configured various timeouts that ensured HotFuzz ran end to end within a manageable time frame. This is important in order to prevent problems caused by fuzzing individual methods or calling specific constructors from halting the entire micro-fuzzing pipeline. We determined the values for these parameters empirically, and only added each parameter after we observed a need for each one in our evaluation. The parameters $(\pi, \chi, \tau, \epsilon, \nu)$ configured the **genetic algorithm (GA)** within HotFuzz. In our evaluation, we assigned these parameters the initial values recommended for genetic algorithm experiments [30]. Finally, σ configured the timeout used in the witness validation stage for confirming AC time witnesses. 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. We found that using one parameter θ to both detect and confirm AC space witnesses sufficed in the interpreted analysis and witness validation environments. We used the same parameters for every method under test and do not tune

Table 1. The Parameters Given to Every μFuzz Instance

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	5 s
λ	The time that may elapse between measuring the fitness of two method inputs	5 s
ω	The amount of time required for a method to run in order to generate an AC witness	10 s
γ	The wall clock time limit for the GA to evaluate the method under test	60 s
π	The size of the initial population	100
χ	The probability two parents produce offspring 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 length of time an AC witness must run for confirmation	5 s
θ	The memory a method must consume to generate and confirm an AC space witness	1 GB

Multiple timeouts prevent HotFuzz from stopping because of individual methods that may be too difficult to efficiently micro-fuzz.

these parameters for specific methods. We argue that this provides a generic approach to detecting AC vulnerabilities in arbitrary Java methods. To micro-fuzz each library under test for either AC time or space vulnerabilities, we created a pair of fuzzing jobs with identical parameters for each method contained in the library with the exception of the α parameter. Each pair consisted of one job that used the IVI seed input generation strategy, and the other used the SRI strategy with the α parameter which bounds the values used when constructing the seed inputs for micro-fuzzing. 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 the Maven repository. 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 potential AC time and space vulnerabilities.

Seed Input Strategy. Given the definition of SRI presented in Section 3.1.2, we use the following procedure to construct the initial seed inputs for every method under test M configured to use SRI in our evaluation. Given the parameter α, μFuzz instantiates a population of seed inputs of size π for M as follows. Let $\mathcal{N}(\mu, \sigma)$ be a normal random number distribution with mean μ and standard deviation σ. For every primitive type required to instantiate a given class, SRI obtains a value $X \leftarrow \mathcal{N}(0, \alpha/3)$. This allows micro-fuzzing to favor small values centered around 0 that exceed the chosen parameter α with a small probability. To be precise, configuring α to be three times the size of the standard deviation σ of our random number distribution $\mathcal{N}$ makes $Pr(|X| > \alpha) < 0.135\%$. This ensures that the primitive values we use to instantiate objects stay within the range $[-\alpha, \alpha]$ with high probability.

5.2 Experimental Results

In our evaluation, HotFuzz detected 52 (26 Time and 26 Space) previously unknown AC vulnerabilities in the Java 8 JRE, detects both intended and unintended vulnerabilities in challenges from the STAC program, and detects 165 (132 Time and 33 Space) AC vulnerabilities in 47 libraries from the 100 most popular libraries found on Maven. Section 5.2.1 describes how using the CHG improves coverage during micro-fuzzing. Section 5.2.2 reports both the total wall-clock time HotFuzz spent micro-fuzzing the artifacts given in our evaluation and reports micro-fuzzing's throughput measured by the average number of 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 detected more AC time and space vulnerabilities.

5.2.1 Improved Coverage with a Class Hierarchy Graph (CHG). Recall the class hierarchy graph presented in Section 3.1.2 which provides HotFuzz a way to obtain concrete classes that extend or implement abstract classes and interfaces that are required to invoke a given method under test. In our evaluation, we generated a CHG using all the classes available in the JRE, and provided this CHG to HotFuzz while micro-fuzzing every artifact in our evaluation. In principle, we could extend the CHG to include every class given in the DARPA STAC challenges and Maven libraries, but given the large number of classes found in these artifacts and their dependencies, we chose to restrict the CHG to the JRE. Over the course of our evaluation we found that using a CHG comprised of the JRE can provide considerable benefit to micro-fuzzing and enabled HotFuzz to cover more methods than it could without the CHG. In order to measure the coverage benefit provided by the CHG, we micro-fuzzed the JRE without setting up the CHG using IVI-derived seed inputs. We found that micro-fuzzing the JRE without the CHG covered only 20,434 methods compared to the 26,598 (+30.17%) methods covered by utilizing the CHG. Given that the JRE is filled with general-purpose utilities that are intended to be used by a wide range of Java programs, it should not be surprising that providing HotFuzz the ability to instantiate concrete classes from abstract classes and interfaces provides a noticeable improvement to HotFuzz's coverage.

5.2.2 Impact of Seed Input Generation Strategy. Table 2 presents the witnesses found when 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 with respect to both time and space. Table 3 provides micro-fuzzing statistics for both time and space including the methods covered by each seed input strategy, the wall clock time spent micro-fuzzing each artifact, and micro-fuzzing's throughput measured in test cases produced per hour. Overall, micro-fuzzing with both strategies for AC time vulnerabilities managed to invoke 29.17% of the methods contained in the JRE, 32.96% of the methods given in the STAC program, and 29.66% of the methods found in the 100 most popular Maven libraries. Micro-fuzzing for AC space vulnerabilities produced comparable coverage, with 25.44% of the methods given in the JRE, 29.49% of the methods given in the STAC program, and 27.28% of the methods in the 100 most popular Maven libraries. As the results indicate, neither seeding strategy is categorically superior to the other, although SRI does consistently outperform IVI on each of the artifacts included in our evaluation in both time and space. For example, when considering the results obtained by micro-fuzzing the JRE with respect to time, SRI identifies 13 vulnerabilities that IVI does not, compared to the eight vulnerabilities IVI finds that SRI misses. At the same time, both strategies are able to find another five vulnerabilities. The same pattern occurs when micro-fuzzing for AC space vulnerabilities, with SRI identifying 15 bugs over IVI, IVI finding three bugs that SRI misses, and both strategies detecting eight bugs. We observe the same pattern in both the STAC artifacts and Maven libraries included in our evaluation, although observe that both strategies were able to solve the same number of challenges in our experiment over the STAC articles when micro-fuzzing for AC time vulnerabilities. When micro-fuzzing the STAC articles for AC space vulnerabilities, SRI outperforms IVI by solving more challenges (see Case Study 5.2.5). These results indicate that SRI outperforms IVI as a seed input strategy, and overall the two approaches are *complementary*. Furthermore, their combined results show that relatively simple micro-fuzzing can expose serious availability vulnerabilities in widely-used software in both time and space. In some cases, IVI-based inputs detect vulnerabilities that SRI-based inputs miss either because only identity values trigger a bug or a bug is triggered by an object derived from an IVI seed. When using HotFuzz in practice, an analyst could combine the benefits of IVI and SRI by simply reserving some fraction

Table 2. AC Witnesses Detected by HotFuzz Using IVI and SRI-derived Seed Inputs with Respect to Both Time and Space

Library	Size	AC Witnesses Detected						AC Witnesses Confirmed					
		Time			Space			Time			Space		
	Methods	IVI	SRI	Both	IVI	SRI	Both	IVI	SRI	Both	IVI	SRI	Both
Java Runtime Environment	91,632	14	19	6	11	41	8	13	18	5	11	23	8
DARPA STAC	30,499	40	49	34	14	44	11	5	5	5	2	8	2
Top 100 Maven Libraries	239,777	84	102	46	23	42	6	78	97	43	11	27	5

of the population for inputs derived from each strategy. Figure 4 visually compares the performance of micro-fuzzing the JRE, STAC challenges, and the 100 most popular Maven libraries for AC time vulnerabilities, using both IVI and SRI-derived seed inputs. From these results, we see that SRI-derived seed inputs produce a marginal improvement over IVI inputs, as micro-fuzzing with SRI detects more AC time bugs over IVI seed inputs in each of our evaluation artifacts. Figure 5 provides the same comparison between IVI and SRI for detecting AC space vulnerabilities, with SRI again outperforming IVI as a seed selection strategy. Section 5.2.7 provides a case study for an AC space bug detected only by SRI in the runtime of the popular Clojure programming language. Figure 6 provides a visual comparison between IVI and SRI-based micro-fuzzing for AC time on a method provided by the JRE that works on regular expressions. According to the documentation, the `RE.split(String input)` method splits a given string `input` into an array of strings based on the regular expression boundaries expressed in the compiled regular expression instance `RE`. Figure 6(a) shows how micro-fuzzing this method using IVI-based seeds fails to arrive at a test case that demonstrates the vulnerability. In contrast, Figure 6(b) shows how using SRI-based seed inputs allows HotFuzz to detect the vulnerability. Additionally, we note that micro-fuzzing with SRI-derived seed inputs require 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 the PoC produced by HotFuzz run for seven days on a Debian system with Intel Xeon E5-2620 CPUs before stopping it. After disclosing our PoC to Oracle, they did not view the time AC vulnerability as a security issue since it belongs to an internal package that an adversary would be unlikely to reach in a production application. Our evaluation revealed a second AC vulnerability within the same `RE` package called `subst(String substituteIn, String substitution)` that substitutes every occurrence of the compiled regular expression in `substituteIn` with the string `substitution`. After tracing the PoC produced by HotFuzz, we observed that it has the same underlying problem as `RE.split`. That is, it appears to loop indefinitely checking the result of `StringCharacterIterator.isEnd`. We observed the PoC for `RE.subst` run for 12 days on the same Debian system on which we tested the `RE.split` PoC 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.

5.2.3 Case Study: Detecting AC Vulnerabilities with IVI-based Inputs. Our evaluation revealed the surprising fact that six methods in the JRE contain AC time vulnerabilities exploitable by simply passing empty values as input. Figure 7 shows the six utilities and APIs that contain AC time vulnerabilities that an adversary can exploit. Upon disclosing our findings to Oracle they

Table 3. Statistics from Evaluating HotFuzz Using IVI and SRI-derived Seed Inputs with Respect to Both Time and Space

Library	Methods Covered (Thousands)						Fuzzing Time (hr)				Throughput (M Tests/hr)			
	Time			Space			Time		Space		Time		Space	
	IVI	SRI	Both	IVI	SRI	Both	IVI	SRI	IVI	SRI	IVI	SRI	IVI	SRI
JRE	26.6	25.2	23.8	25.4	24.2	22.3	14.3	19.5	14.6	15.6	4.4	3.1	2.8	2.6
DARPA STAC	8.9	9.2	8.1	7.9	8.2	7.2	7.8	8.7	9.7	9.4	3.6	3.1	2.0	2.0
Top 100 Maven	69.6	68.7	66.9	63.1	61.8	59.5	52.9	55.5	62.6	73.7	5.9	5.6	4.0	3.1

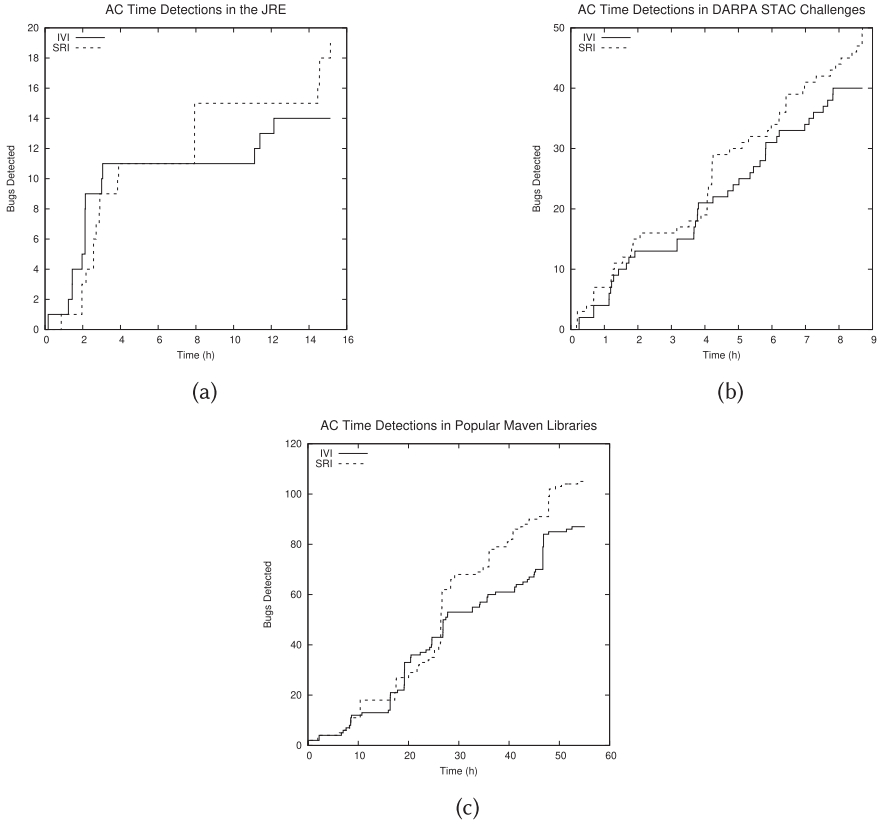


Fig. 4. Micro-fuzzing for AC time vulnerabilities with SRI-derived seed inputs outperforms IVI-derived seed inputs for the JRE (Graph (a)), the DARPA STAC program (Graph (b)), and the 100 most popular Maven libraries (Graph (c)).

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 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 programs exclusively pass string constants to the constructor as input, this is a difficult invariant to always enforce.

5.2.4 Case Study: Arithmetic DoS in Java Math. As a part of our evaluation, HotFuzz detected 5 AC time vulnerabilities inside the JRE's Math package. To the best of our knowledge, no prior

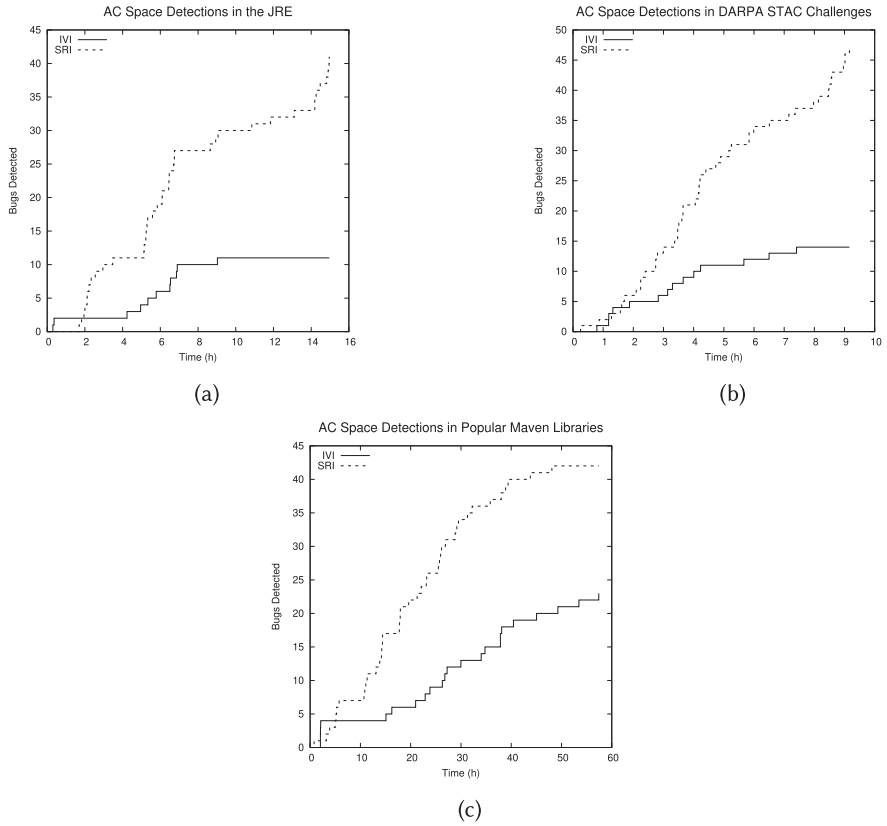
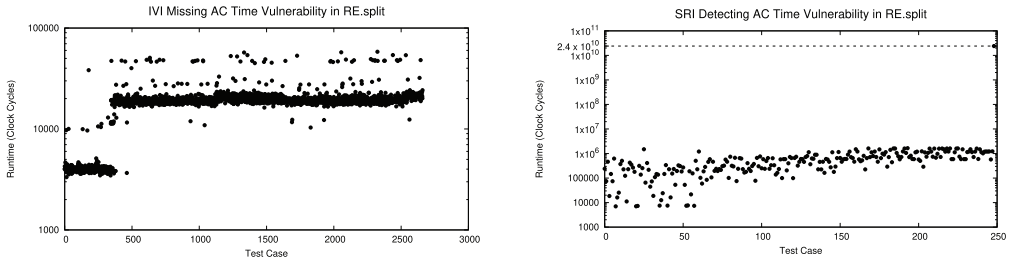


Fig. 5. Micro-fuzzing for AC space vulnerabilities with SRI-derived seed inputs outperforms IVI-derived seed inputs for the JRE (Graph (a)), the DARPA STAC program (Graph (b)), and the 100 most popular Maven libraries (Graph (c)).

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 JIT compilation is enabled. On IBM's J9 platform, the exploit ran for four 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 [4] for our findings. Oracle considered this a Security-in-Depth issue and acknowledged our contribution in their Critical Patch Update Advisory [8]. Google argued that it does not fall within the definition of a security vulnerability for the Android platform. HotFuzz automatically constructs valid instances of `BigDecimal` and `BigInteger` that substantially slow down methods in both classes. For example, simply incrementing `1e2147483647` by 1 takes over an hour to compute on Oracle's JDK even with JIT compilation enabled. HotFuzz finds these vulnerabilities without any domain-specific knowledge



(a) IVI-derived inputs (b) SRI-derived inputs

Fig. 6. Micro-fuzzing `com.sun.org.apache.regexp.internal.RE.split(String input)` with IVI-derived inputs fails to detect a zero-day AC vulnerability in the JRE, while SRI-derived inputs detect the vulnerability correctly (see the upper-right-hand corner of Graph (b)). Observe that the y-axis for Graph (a) is 5 orders of magnitude smaller than Graph (b).

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 `1e2147483647` and `2e2147483647`, return immediately, whereas adding `1e2147483647` to `1e0`, can run for over an hour on Oracle's JVM. The root cause of this performance overhead lies in how the JDK transforms numbers during its arithmetic operations [6]. For example, suppose a program uses the `BigDecimal` class to compute the sum $x_1 \times 10^{y_1} + x_2 \times 10^{y_2}$ where $y_1 \neq y_2$. Let x_{min} be the coefficient that belongs to the smaller exponent and x_{max} the coefficient that belongs to the larger exponent. The `add` method first computes the difference $|y_1 - y_2|$ and then defines an integer x_{scaled}, an instance of `BigInteger` which may represent an integer of arbitrary size, and directly computes $x_{scaled} = x_{max} \times 10^{|y_1 - y_2|}$. This allows the `add` method to complete the addition by returning a `BigDecimal` represented with a coefficient given by $x_{scaled} + x_{min}$ and an exponent given by the smaller of y_1 and y_2. Unfortunately, it also opens up the possibility of a significant performance penalty while computing x_{scaled} that an adversary could exploit to achieve DoS when the difference between y_1 and y_2 is large. In the PoC given above, `add` must compute $x_{scaled} = 1 \times 10^{2147483647}$ before simply adding it to 1 with an exponent of 0. Anecdotally, the EyeVM helped pinpoint this issue by tracing the execution of the PoC. When tracing $1 + 1 \times 10^{2147483647}$, the method `bigMultiplyPowerTen` started computing 1×10^{2147483647} but did not return before we manually stopped the PoC. This method appeared to be the source of the performance penalty because it was absent when tracing $2 \times 10^{2147483647} + 1 \times 10^{2147483647}$ which completed immediately. 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 8. We found that several general-purpose programming languages hosted on the JVM are vulnerable to this attack along with org.json, a popular JSON parsing library. Developers face numerous security threats when they validate input values given as strings. The vulnerabilities we discussed in this section are especially problematic because malicious input is perfectly valid, albeit very large, floating point numbers. If a program performs any arithmetic over a `BigDecimal` object derived from user input, then it must take care to prevent the user from providing arbitrary numbers in scientific notation. Likewise, these results show that developers must be careful when converting between these two classes, as interpreting certain

```
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. 7. Proof of concept exploits for AC vulnerabilities that require only IVI-based inputs to trigger.

```
clojure=> (inc (BigDecimal. "1e2147483647"))
clojure=> (dec (BigDecimal. "1e2147483647"))

scala> BigDecimal("1e2147483647")+1
```

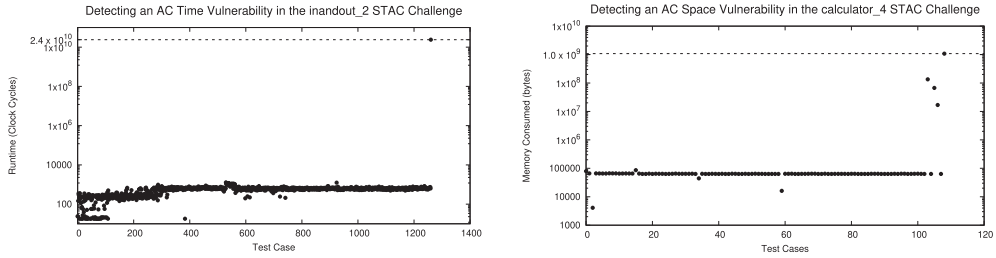
```
groovy:000> 1e2147483647+1

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

Fig. 8. Proof of concept exploits that trigger inefficient arithmetic operations for the `BigDecimal` class in Clojure, Scala, Groovy, and the org.json library.

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. After we disclosed this vulnerability to vendors, we observed that recent releases of the JRE provide mitigations for it. For example, in JRE build `1.8.0_242`, the PoCs we present in this section immediately end with an exception thrown by the `BigInteger` class. Instead of naively computing x_{scaled}, the new implementation first checks to see if x_{scaled} will exceed a threshold and, if so, aborts the inefficient computation with an exception before it can affect the availability of the overall process. While this defends against the original AC vulnerability, it also introduces a new opportunity for DoS by allowing an adversary to trigger exceptions that programs may fail to properly handle.

5.2.5 Case Study: DARPA STAC Challenges. The DARPA 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 contained 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



(a) AC Time (b) AC Space

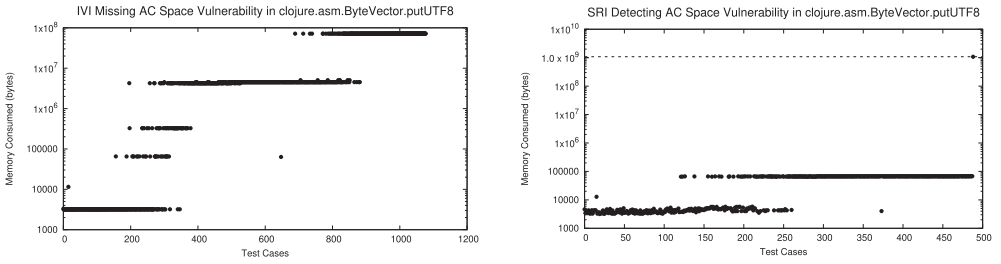
Fig. 9. HotFuzz detecting AC vulnerabilities in both time and space found in the inandout_2 and calculator_4 challenges from the DARPA STAC Program (see the upper-right-hand corner of Graphs (a) and (b)).

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 time and space attacks, we found that HotFuzz automatically marked 13 (five time and eight space) out of 61 (36 time and 25 space) challenges as vulnerable. One challenge, `inandout_2` provides a web service that allows users to order pizzas online. By running HotFuzz over this challenge, it identifies multiple methods with AC time vulnerabilities in its code. Figure 9 visualizes HotFuzz detecting an AC time vulnerability in the `subsequentEnergyOf2` method found within the `PizzaParameters` class in the challenge. When we traced the execution of an exploit that achieved DoS against the pizza service, we observed the vulnerable method `subsequentEnergyOf2` identified by HotFuzz in the exploit's trace. A second challenge, `calculator_4` evaluates arithmetic expressions submitted by users over a network. HotFuzz identifies an AC space vulnerability that successfully causes the `Modify.modify` method to consume significant memory while altering user data. Figure 9(b) visualizes how HotFuzz quickly evolves the initial population of 100 objects to find a set of inputs that consume excessive memory. When we traced the exploit for `calculator_4` which uses a pathological expression to balloon a calculator's memory consumption, we observed the `modify` method flagged by HotFuzz appear within the trace.

5.2.6 Case Study: Slow Parsing in org.json. Over the course of our evaluation, HotFuzz detected a previously unknown AC time vulnerability inside the popular org.json library. The org.json library is widely used, and is found as a dependency in Google's Closure compiler and the Spring framework. 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 [12]. 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. Over the course of 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 was completed, we took the PoC program generated by HotFuzz and observed it run for 120 hours on a Debian system with Intel Xeon E5-2620 CPUs in an unmodified Java environment with JIT enabled. The SRI strategy that produced this test case sampled primitive values uniformly at random from the interval $[0, \alpha)$.

Sampling from the normal distribution $\mathcal{N}(0, \alpha/3)$ detected the bug in the `JSONML` package, but the test case did not pass the witness validation stage. 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 time vulnerability that exposes programs that depend on this library to potential DoS attacks. We communicated our findings to the owners of the `JSON-java` project [3] who confirmed the behavior we observed as a bug. The developers immediately found the root cause of the infinite loop after debugging the method starting with the test case produced by HotFuzz. This test case opened an XML comment with the string `<!` which prompted a loop inside the `toJSONObject` method to check for special characters that represent the beginning (<) and end (>) of a comment until it reached the end of the original comment. The test case produced by HotFuzz caused a method in this loop, `nextMeta`, to always return the same character, and therefore prevented the loop from advancing. After fixing this bug, the developers included our test case in `org.json`'s test suite in order to prevent the issue from occurring in future releases. The test case that triggered this infinite loop is small (242 bytes) and demonstrates the potential micro-fuzzing has to uncover serious AC vulnerabilities hiding in popular Java libraries. After micro-fuzzing the `toJSONObject` method on the patched `org.json` library, we discovered six test cases that triggered AC vulnerabilities, but these were fixed in the latest release of `org.json`.

5.2.7 Case Study: Excessive Memory Consumption in the Clojure Runtime and ASM Library. Clojure is a popular functional programming language that provides a modern LISP dialect for contemporary computing environments including the JVM, Javascript, and the Microsoft **Common Language Runtime** (**CLR**). One way to deploy a Clojure program is to package its source files along with the Clojure runtime which can be obtained as a standalone JAR file. At production, the Clojure runtime parses and interprets the program within the JVM where it can take advantage of features like JIT compilation. The Clojure runtime contains an optimized implementation of a vector for holding arbitrary bytes called `ByteVector`. The `ByteVector` class holds both the array of bytes and an integer field that denotes the amount of bytes actually stored within the array, since a given vector may not utilize all the space that is available in the byte array. This optimization can cause trouble later on, if an adversary is somehow able to alter the `length` field to a value that is substantially larger than the length of the byte array stored in the object. While micro-fuzzing the Clojure runtime, we observed that using SRI-derived seed inputs created a small test case (1,608 bytes) that demonstrated a corrupted ByteVector could cause one of its methods to inadvertently allocate gigabytes worth of memory and trigger a DoS attack (Figure 10(b)). This demonstrates the utility of micro-fuzzing with SRI-derived seed inputs, as the corresponding run that utilized IVI inputs failed to discover the issue, even while using more test cases (Figure 10(a)). The method that demonstrates this vulnerability, `putUTF8`, attempts to place a UTF-8 string into a `ByteVector`'s byte array, but the test case produced by HotFuzz causes the method to get stuck growing the array to match the value given by the corrupted `length` field. A likely attack for this bug is that some applications may parse `ByteVector` objects through either the popular JSON format or a binary format protocol like Protobufs. If an adversary is able to submit their own corrupted `ByteVector` instances through such an interface, they could easily cause benign methods like `putUTF8` to consume significant memory and achieve a DoS attack. Indeed, this is how the PoC emitted by HotFuzz works, by parsing the test case given in JSON into a valid `ByteVector` object, and then calling the `putUTF8` method on the object. Furthermore, we found that the `ByteVector` implementation contained in the Clojure runtime is borrowed from the widely used ASM bytecode manipulation library [20]. We reproduced the PoC generated by HotFuzz on the latest release of ASM (9.2) which suggests this issue impacts not only projects using ASM, but also those derived



(a) IVI-derived inputs (b) SRI-derived inputs

Fig. 10. Micro-fuzzing `clojure.asm.ByteVector.putUTF8` with IVI-derived inputs fails to detect a zero-day AC vulnerability in the Clojure runtime, while SRI-derived inputs detect the vulnerability using fewer test cases (see the upper-right-hand corner of Graph (b)). Note that the *y*-axis for Graph (a) is 1 order of magnitude smaller than Graph (b).

from ASM's source code. In addition to testing the PoC emitted by HotFuzz on the version of Clojure given in our evaluation, we also verified the same behavior occurs on the latest version of Clojure available on Maven (1.10.3). Fortunately, a simple mitigation seems possible in this case since the `length` attribute is intended to represent the number of bytes actually stored in the given array. This means any value of `length` that exceeds the actual length of the array demonstrates evidence of tampering, since an array cannot hold more bytes beyond its capacity. Upon detecting the corruption, a method could throw an exception and avoid erroneously allocating memory. This check does not incur significant performance overhead, since Java supports querying the length of arrays in constant time. We reported this issue to the Clojure developers who observed that the compiler never deserializes `ByteVector` objects, and furthermore, they consider the ASM library embedded within Clojure private. Nonetheless, they periodically update the ASM sources embedded in the Clojure source tree to stay up to date with the official ASM library. Upon reviewing our findings, the ASM developers confirmed the behavior as a security issue, and merged in a patch that implements our proposed mitigation [19]. The Clojure developers have confirmed that this mitigation will eventually make its way into a future Clojure release as well.

6 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.

6.1 AC Vulnerability Analysis

Prior work for detecting AC vulnerabilities in Java programs includes static analysis on popular libraries [42, 48, 79], object-graph engineering on Java's serialization facilities [29], and exploiting worst-case runtime of algorithms found in commercial grade networking equipment [28]. On the Android platform, Huang et al. [38] use a combination of static and dynamic analysis to detect AC vulnerabilities within Android's System Server. Further up the application stack, Pellegrino et al. [57] identify common implementation mistakes that make web services vulnerable to DoS attacks. Finally, Holland et al. [37] propose a statically-informed dynamic 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.

6.2 Test Case Generation

Program analysis tools can generate test cases that exercise specific execution paths in a program and demonstrate the presence of bugs. When the program source is at hand, property-based and generative testing tools apply techniques popularized by QuickCheck [26] in order to determine whether a given function deviates from a specification given as a logical formula. When working directly on JAR files, several tools perform symbolic execution within the Java Pathfinder platform [34, 40, 49] in order to increase code coverage in Java test suites. Toffola et al. [75] introduced PerfSyn which uses combinatoric search to construct test programs that trigger performance bottlenecks in Java methods. Symbolic execution has found serious security bugs when applied to whole programs [23, 53] and in under-constrained settings [60] similar to HotFuzz. More recent work in this area utilizes information given in compiler error messages to iteratively refine inputs to polymorphic Rust functions [72].

6.3 Fuzz Testing

State-of-the-art fuzzers [13, 81] 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 [63] and techniques for improving a fuzzer's code coverage [24, 78]. Prior work has seeded fuzzing by replaying sequences of kernel API calls [33], commands from Android apps to smart IoT Devices [25, 64], input provided by human assistants [68], or generating device I/O from hypervisor semantics [21]. Recent techniques for improving code coverage during fuzz testing include introducing selective symbolic execution [71], control- and data-flow analysis on the program under test [62], reducing collisions in code coverage measurements [31], and altering the program under test [58]. Prior work applies existing fuzz testers to discover AC vulnerabilities in whole programs [47, 59], and in Java programs by combining fuzz testing with symbolic execution [54] or seeding black box fuzzing with information taken from program traces [50]. Instead of adapting existing fuzzers to detect AC vulnerabilities, the Singularity tool implements a genetic algorithm over programs in a domain specific language that represent a method's input in order to search for inputs that trigger a Java method's worst-case execution time [76]. In contrast, HotFuzz micro-fuzzes entire libraries without any manual set up and uses a genetic algorithm on individual Java objects of arbitrary type in order to detect AC vulnerabilities in a library's methods. This departs from prior approaches that restrict fuzzing individual methods with either bitstreams, or a simple domain specific language that expresses integer arithmetic and list operations.

6.4 Resource Analysis

Recent interest in AC and side-channel vulnerabilities have increased the focus on resource analysis research. In this area, Proteus [80] presented by Xie et al. and Awadhutkar et al. [15] study sensitive paths through loops that might represent AC vulnerabilities. Meanwhile, Kothary [44, 66] investigates how human-machine interaction can improve program analysis by better detecting critical paths and side channels. Further work identifies side-channels by solving the trace-set discrimination problem in individual programs [74]. In Comb [36], Holland et al. propose the **Projected Control Graph** (**PCG**) as a technique for computing relevant program behaviors for analysis. Another approach to resource-oriented static analysis formalizes how programs consume resources within an interactive proof assistant. This enables programmers to derive concrete resource bounds over integer programs using **Automatic Amortized Resource Analysis** (**AARA**) [22, 35]. This line of work computes a program's resource usage using a combination of

formal proofs and constraint solving over linear programs. In contrast, HotFuzz provides quantitative measurements of program behavior over concrete inputs in a dynamic, empirical fashion.

7 CONCLUSION

In this work, we present HotFuzz, a fuzzer that detects 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 IVI and SRI techniques to detect AC vulnerabilities in methods under test. We evaluate HotFuzz on the JRE, challenge programs developed in the DARPA STAC program, and the 100 most popular Maven libraries. In conducting this evaluation, we discovered previously unknown AC vulnerabilities in production software, including 52 (26 Time and 26 Space) in the JRE, 165 (132 Time and 33 Space) in 47 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 with respect to both time and space in real-world software.

REFERENCES

[1] 2006. The Java Virtual Machine Specification. Retrieved July 16, 2021 from https://docs.oracle.com/javase/specs/jvms/se8/html/index.html.

[2] 2006. The JVM Tool Interface (JVM TI): How VM Agents Work. Retrieved July 16, 2021 from https://www.oracle.com/technetwork/articles/javase/index-140680.html.

[3] 2015. JSON-Java Project. Retrieved July 16, 2021 from https://stleary.github.io/JSON-java/.

[4] 2018. CVE-2018-1517. Retrieved July 16, 2021 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1517.

[5] 2018. CVE-2018-5390. Retrieved July 16, 2021 from https://nvd.nist.gov/vuln/detail/CVE-2018-5390#vulnCurrentDescriptionTitle.

[6] 2019. Arithmetic in JDK BigDecimal. Retrieved July 16, 2021 from https://hg.openjdk.java.net/jdk8u/jdk8u-dev/jdk/file/d13abc740e42/src/share/classes/java/math/BigDecimal.java#l4464.

[7] 2019. CVE-2019-6486. Retrieved July 16, 2021 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6486.

[8] 2019. Oracle Critical Patch Update Advisory - January 2019. Retrieved July 16, 2021 from https://www.oracle.com/technetwork/security-advisory/cpujan2019-5072801.html.

[9] 2021. CVE-2021-22312. Retrieved July 16, 2021 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-22312.

[10] 2021. CVE-2021-25252. Retrieved July 16, 2021 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-25252.

[11] 2021. CVE-2021-3492. Retrieved July 16, 2021 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3492.

[12] 2021. JSONML Tutorials Point. Retrieved July 16, 2021 from https://www.tutorialspoint.com/org_json/org_json_jsonml.htm.

[13] 2021. libFuzzer – A library for coverage-guided fuzz testing. Retrieved July 16, 2021 from https://llvm.org/docs/LibFuzzer.html.

[14] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with input-to-state correspondence. In *Proceedings of the ISOC Network and Distributed System Security Symposium*.

[15] Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, and Suresh Kothari. 2017. Intelligence amplifying loop characterizations for detecting algorithmic complexity vulnerabilities. In *Proceedings of the Asia-Pacific Software Engineering Conference*.

[16] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: Fuzz driver generation at scale. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[17] Scott Behrens and Bryan Payne. 2016. Starting the Avalanche: Application DDoS In Microservice Architectures. Retrieved July 16, 2021 from https://medium.com/netflix-techblog/starting-the-avalanche-640e69b14a06.

[18] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2020. HotFuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In *Proceedings of the ISOC Network and Distributed System Security Symposium*.

[19] Eric Bruneton. 2021. Add an assertion in ByteVector.enlarge(). Retrieved August 28, 2021 from https://gitlab.ow2.org/asm/asm/-/commit/cf20d046cab42e80866d8557e1b4ba4d47186300.

[20] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *Proceedings of the Adaptable and Extensible Component Systems*.
[21] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. 2022. Morphuzz: Bending input space to fuzz virtual devices. In *Proceedings of the USENIX Security Symposium*.
[22] Quentin Carbonneaux, Jan Hoffmann, Thomas W. Reps, and Zhong Shao. 2017. Automated resource analysis with coq proof objects. In *Proceedings of the International Conference on Computer-Aided Verification*.
[23] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[24] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[25] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. In *Proceedings of the ISOC Network and Distributed System Security Symposium*.
[26] Koen Claessen and John Hughes. 2011. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM SIGPLAN Notices*.
[27] Scott A. Crosby and Dan S. Wallach. 2003. Denial of service via algorithmic complexity attacks. In *Proceedings of the USENIX Security Symposium*.
[28] Adam Czubak and Marcin Szymanek. 2016. Algorithmic complexity vulnerability analysis of a stateful firewall. In *Proceedings of the International Conference on Information Systems Architecture and Technology*.
[29] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil pickles: DoS attacks based on object-graph engineering. In *Proceedings of the European Conference on Object-Oriented Programming*.
[30] A. E. Eiben and James E. Smith. 2015. *Introduction to Evolutionary Computing*. Springer.
[31] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path sensitive fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[32] Patrice Godefroid. 2014. Micro execution. In *Proceedings of the International Conference on Software Engineering*.
[33] HyungSeok Han and Sang Kil Cha. 2017. IMF: Inferred model-based fuzzer. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[34] Klaus Havelund. 1999. Java pathfinder, a translator from java to promela. In *Proceedings of the Theoretical and Practical Aspects of SPIN Model Checking*.
[35] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource aware ML. In *Proceedings of the International Conference on Computer-Aided Verification*.
[36] Benjamin Holland, Payas Awadhutkar, Suresh Kothari, Ahmed Tamrawi, and Jon Mathews. 2018. COMB: Computing relevant program behaviors. In *Proceedings of the International Conference on Software Engineering*.
[37] Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. 2016. Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*.
[38] Heqing Huang, Sencun Zhu, Kai Chen, and Peng Liu. 2015. From system services freezing to system server shutdown in android: All you need is a loop in an app. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[39] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. Fuzzgen: Automatic fuzzer generation. In *Proceedings of the USENIX Security Symposium*.
[40] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. 2009. jFuzz: A concolic whitebox fuzzer for java. In *Proceedings of the NASA Formal Methods Symposium*.
[41] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. 2017. POSTER: AFL-based fuzzing for java with kelinci. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[42] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static analysis for regular expression denial-of-service attacks. In *Proceedings of the International Conference on Network and System Security*.
[43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[44] Suresh Kothari, Ahmed Tamrawi, and Jon Mathews. 2016. Human-machine resolution of invisible control flow. In *Proceedings of the IEEE International Conference on Program Comprehension*.
[45] Eugene Kuleshov. 2007. Using the ASM framework to implement common Java bytecode transformation patterns. *Aspect-Oriented Software Development*.
[46] laf intel. 2016. Circumventing Fuzzing Roadblocks with Compiler Transformations. Retrieved July 16, 2021 from https://lafintel.wordpress.com/.
[47] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: Automatically generating pathological inputs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[48] V. Benjamin Livshits and Monica S. Lam. 2005. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the USENIX Security Symposium*.
[49] Kasper Søe Luckow, Marko Dimjasevic, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamaric, and Vishwanath Raman. 2016. JDart: A dynamic symbolic analysis framework. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*.
[50] Qi Luo, Aswathy Nair, Mark Grechanik, and Denys Poshyvanyk. 2017. FOREPOST: Finding performance problems automatically with feedback-directed learning software testing. *Empirical Software Engineering* 22, 1 (2017), 6–56.
[51] Microsoft. 2015. Microsoft Security Risk Detection. Retrieved July 16, 2021 from https://www.microsoft.com/en-us/security-risk-detection/.
[52] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the ACM* 33, 12 (1990), 32–44.
[53] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the USENIX Security Symposium*.
[54] Yannic Noller, Rody Kersten, and Corina S. Pasareanu. 2018. Badger: Complexity analysis with fuzzing and symbolic execution. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*.
[55] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. 2018. Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the ACM Symposium on Applied Computing*.
[56] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*.
[57] Giancarlo Pellegrino, Davide Balzarotti, Stefan Winter, and Neeraj Suri. 2015. In the compression hornet's nest: A security study of data compression in network services. In *Proceedings of the USENIX Security Symposium*.
[58] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by program transformation. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[59] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[60] David A. Ramos and Dawson R. Engler. 2015. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the USENIX Security Symposium*.
[61] Guido Ranken. 2018. libFuzzer Java. Retrieved July 16, 2021 from https://github.com/guidovranken/libfuzzer-java.
[62] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the ISOC Network and Distributed System Security Symposium*.
[63] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *Proceedings of the USENIX Security Symposium*.
[64] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2021. DIANE: Identifying fuzzing triggers in apps to generate under-constrained inputs for IoT devices. In *Proceedings of the IEEE Symposium on Security and Privacy*.
[65] Kenneth B. Russell and Lars Bak. 2001. The hotspot serviceability agent: An out-of-process high-level debugger for a java virtual machine. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*.
[66] Ganesh Ram Santhanam, Benjamin Holland, Suresh Kothari, and Nikhil Ranade. 2017. Human-on-the-loop automation for detecting software side-channel vulnerabilities. In *Proceedings of the Information Systems Security*.
[67] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*.
[68] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the ACM Conference on Computer and Communications Security*.
[69] Google Open Source. 2021. OSS-Fuzz. Retrieved July 16, 2021 from https://github.com/google/oss-fuzz.
[70] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the web: A study of ReDoS vulnerabilities in javascript-based web servers. In *Proceedings of the USENIX Security Symposium*.
[71] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the ISOC Network and Distributed System Security Symposium*.
[72] Yoshiki Takashima, Ruben Martins, Limin Jia, and Corina S Păsăreanu. 2021. SyRust: Automatic testing of rust libraries with semantic-aware program synthesis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[73] EJ Technologies. 2015. Java Profiler. Retrieved July 16, 2021 from https://www.ej-technologies.com/products/jprofiler/overview.html.

[74] Saeid Tizpaz-Niari, Pavol Černý, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Ashutosh Trivedi. 2017. Discriminating traces with time. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*.

[75] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2018. Synthesizing programs that expose performance bottlenecks. In *Proceedings of the International Symposium on Code Generation and Optimization*.

[76] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 2018. Singularity: Pattern fuzzing for worst case complexity. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[77] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the International Conference on Software Engineering*.

[78] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[79] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. 2017. Static detection of DoS vulnerabilities in programs that use regular expressions. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*.

[80] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: Computing disjunctive loop summary via path dependency analysis. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[81] Michał Zalewski. 2014. American Fuzzy Lop Technical Details. Retrieved May 14, 2022 from https://lcamtuf.coredump.cx/afl/technical_details.txt.

Received August 2021; revised February 2022; accepted April 2022