Acquirer: A Hybrid Approach to Detecting Algorithmic Complexity Vulnerabilities

Yinxi Liu
The Chinese University of Hong Kong
Hong Kong SAR, China
yxliu@cse.cuhk.edu.hk

Wei Meng
The Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

ABSTRACT

Algorithmic Complexity (AC) Denial-of-Service attacks have been a threat for over twenty years. Attackers craft particular input vectors to trigger the worst-case logic of some code running on the server side, which leads to high resource consumption and performance degradation. In response, several vulnerability detection tools have been developed to help developers prevent such attacks. Nevertheless, these state-of-the-art tools either focus on a specific type of vulnerability or suffer from state explosion. They are either limited to a small detection scope or unable to run efficiently.

This paper aims to develop a fully automated approach to effectively and efficiently detecting AC vulnerabilities. We present the design and implementation of Acquirer, which detects AC vulnerabilities in Java programs. Acquirer first statically locates potentially vulnerable structures in the target program, then performs efficient selective path exploration to dynamically verify the existence of two different execution paths with a significant computation cost difference. The vulnerable structures it detects can also help the developers fix the corresponding vulnerabilities.

We evaluated Acquirer with two widely used benchmark datasets and compared it with four state-of-the-art tools. In the evaluation, it detected 22 known AC vulnerabilities, which substantially outperformed all the existing tools together. Besides, it discovered 11 previously unknown AC vulnerabilities in popular real-world applications. Our evaluation demonstrates that Acquirer is highly effective and efficient in automatically detecting AC vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Denial-of-Service Attacks; Vulnerability Detection

ACM Reference Format:
Yinxi Liu and Wei Meng. 2022. Acquirer: A Hybrid Approach to Detecting Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3559337

1 INTRODUCTION

Denial-of-Service (DoS) attack has been a major threat to the availability of the vast number of applications running on the Internet. Attackers can craft attack requests to consume the computation resources of the server hosting a victim application. In general, the effect of a DoS attack is restricted by the volume of attack traffic an attacker can initiate, but some exploits can significantly increase the severity of such attacks.

Algorithmic Complexity (AC) vulnerabilities, for example, enable asymmetric DoS attacks. They can cause polynomial or even exponential resource consumption concerning the inputs. Such vulnerabilities amplify the attacks' impact and require much more defense effort. They can typically be launched by a single user, with a relatively small payload, to cause a disproportionately powerful effect. On this account, AC vulnerabilities have become a common attack vector for DoS attacks, which can be exploited to further amplify the power of the traditional distributed DoS attacks.

AC vulnerabilities arise from intended functionalities/algorithms with high worst-case complexity. For example, the Decompression Bombs [1] vulnerability comes from a decompression algorithm; the Billion laughs attack [2] utilizes the worst-case performance of an XML parser; the ReDoS [3] attacks make use of the worst-case super-linear implementation of regex engines; and the Hash-table DoS Attacks [4] triggers collisions in a hash table. Exploiting the intended functionalities makes AC DoS attacks "quieter" than the traditional DoS attacks, because widely used vulnerability indicators like runtime exceptions, unusually high traffic, and excessive logging may not be present. Consequently, system administrators and security experts may fail to notice an ongoing AC DoS attack.

Even though AC DoS has been a well-known issue for years, some high-profile applications still fall victim to it. Since the AC vulnerabilities are not caused by implementation bugs but by algorithmic logic, it is hard for developers to detect and mitigate them. Developers need to carefully design their algorithms or implement input sanitization and hard resource limits to prevent such attacks, which could be a large amount of work with limited effects.

Researchers have proposed many tools for detecting AC vulnerabilities [6–8, 17, 27, 30, 33, 39, 40]. However, developing a completely automated and widely applicable tool for effectively and efficiently detecting AC vulnerabilities remains a challenge. Some works focus on a specific type of AC vulnerabilities with known sources, *e.g.*, regex engines [27], resource-intensive library APIs [17, 39], which have limited application scope. Others [8, 30, 33, 40] do not

[1]Decompression Bombs: https://en.wikipedia.org/wiki/Zip_bomb
[2]Billion Laughs Attack: https://en.wikipedia.org/wiki/Billion_laughs_attack
[3]ReDoS: https://en.wikipedia.org/wiki/ReDoS
[4]https://fahrplan.events.ccc.de/congress/2011/Fahrplan/events/4680.en.html

have specialized AC vulnerability modeling and analysis. Instead, they apply a general vulnerability detection approach, and obtain statistics about the resulting complexity or resource consumption of inputs at run time using either fuzzing or symbolic/concolic execution. These tools suffer from state explosions and take a long time to run. Some researchers [6, 7] use the knowledge of loop analysis to build static analyzers for AC vulnerabilities. However, to the best of our knowledge, such static analyzers have high false-positive rates and require considerable effort from a human analyst for vulnerability validation.

This paper aims to improve the degree of automation and efficiency in detecting general AC vulnerabilities. We develop Acquirer, an automated, efficient hybrid analysis tool for detecting AC vulnerabilities in Java programs. Acquirer incorporates a static analyzer specialized in identifying vulnerable code patterns. To validate the vulnerabilities, it automatically generates the test harnesses for dynamic symbolic execution by constructing the necessary calling context and instrumenting the target program. Using the instrumented program, it then performs selective dynamic symbolic execution guided by branch policies learned from these patterns to efficiently verify the existence of two paths with a significant cost difference. Such a dynamic validator helps Acquirer exclude false positives. Further, the path constraints it reports can help the developers analyze and fix the vulnerabilities.

We extensively evaluated Acquirer with two widely used datasets: the Worst-case Inputs from Symbolic Execution (WISE) benchmark [10] and the challenges in the DARPA Space and Time Analysis for Cybersecurity (STAC) program [5]. We compare Acquirer with four state-of-the-art tools in the evaluation. Acquirer can detect almost all vulnerabilities that can be detected by all the state-of-the-art tools combined in a much more efficient manner. Specifically, Acquirer does not require prior knowledge of the vulnerabilities and human effort for generating test harnesses. This allows it to detect 3.4 times more vulnerabilities in 76.97% less analysis time than a state-of-the-art fuzzer. Besides, Acquirer can detect many vulnerabilities that other tools cannot detect because of its precise static analyzer and its efficient selective dynamic path exploration strategy. We further conducted a large-scale analysis by applying Acquirer to 46 popular open-source Java web applications on GitHub, and detected 11 new vulnerabilities. We disclosed all the newly detected vulnerabilities to the relevant developers. Our evaluation results demonstrate that Acquirer can both *effectively* and *efficiently* detect AC vulnerabilities.

In summary, this paper makes the following contributions:

- We propose a widely applicable code pattern for identifying code blocks potentially vulnerable to AC DoS attacks.
- We develop a hybrid analysis incorporating a static path selection algorithm and a selective symbolic execution to detect AC vulnerabilities.
- We integrate the above analyses with an automated test harness generator into Acquirer, an effective and efficient hybrid analysis tool for automatically detecting AC vulnerabilities in Java programs.
- We demonstrate the capabilities of Acquirer by detecting 11 previously unknown AC vulnerabilities in a benchmark dataset, and 11 AC new vulnerabilities in popular open-source applications.

[5]STAC Dataset: https://github.com/Apogee-Research/STAC

2 BACKGROUND

In this section, we first introduce the necessary background related to Algorithmic Complexity (AC) vulnerabilities in §2.1. Then we present existing detection approaches and their limitations in §2.2.

2.1 AC Vulnerabilities

AC DoS attack was first introduced as a new class of low-bandwidth DoS attacks by Crosby and Wallach in 2003 [14]. These attacks exploit the algorithmic deficiencies in programs. More specifically, for a function in the target program that accepts user input, there exist two inputs of similar sizes that may give rise to very different running time. Many commonly-used textbook algorithms have a worst-case complexity higher than the average-case complexity, like sorting, tree traversal, *etc.* One example shown in Listing 1 implements the insertion sort algorithm to sort a user-provided integer array a of N elements. The average-case complexity of the algorithm is $O(n \log n)$, whereas its worst-case complexity is $O(n^2)$. The worst-case execution of such algorithms can be exploited to launch a DoS attack. For instance, some sorting code can take more than 500 seconds under the input limit of 25K bytes as shown in Table 5, because the comparison time between two sorted objects may be long, or some projects have sub-optimal implementations which perform worse than $O(n^2)$ under some conditions.

```
public static void sort(int[] a) {
    final int N = a.length;
    for (int i = 1; i < N; i++) {
        int j = i - 1;
        int x = a[i];
        while ((j >= 0) && (a[j] > x)) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = x;
    }
}
```

Listing 1: An implementation of the insertion sort algorithm.

Such algorithmic deficiencies widely exist in real-world programs, as it is natural to process data differently depending on their values, which might lead to quite different computation costs. As a result, they have been exploited and have made a huge impact in practice. For instance, in the zip bomb attack [6], the attacker crafted a correctly formatted archive, which took excessive computation resources to unpack. In particular, the data extracted from a crafted archive whose size was only 42KB could be as large as 4.5PB.

2.2 Detection of AC Vulnerabilities

Because of the severity of AC DoS attacks, people have been trying to detect AC vulnerabilities ahead. However, even though some domain-specific AC vulnerabilities have been well studied, detecting general AC vulnerabilities remains difficult. To manifest an AC vulnerability, it is necessary to reason about the running time of a program over different inputs without concrete execution. It has

[6]Zip Bomb: https://www.theregister.com/2001/07/23/dos_risk_from_zip/

been proven that computing discriminants to explain the different execution time of a program is NP-hard [38].

Therefore, people analyze and compare different execution paths in a program to detect AC vulnerabilities in practice. This task would be difficult for a human analyst, because of the complexity of determining the computation costs and the feasibility of a large number of execution paths. Recently, researchers have focused on developing automated or semi-automated tools to assist the detection of general AC vulnerabilities, but these tools either report many false positives or have low efficiency. Current approaches can be classified as follows.

2.2.1 Code Structure Analysis. Code structure analysis detects AC vulnerabilities by identifying program structures with variable execution time. Loops (and recursions) are code sequences that could be executed repeatedly for an indefinite amount of time. This characteristic makes them the common causes of AC vulnerabilities. Therefore, previous works try to detect AC vulnerabilities using static loop analysis algorithms [6, 44]. However, existing analyses can only provide a sound result for a limited type of loop. They do not work well for analyzing complex loops such as nested loops and loops that contain intricate branches.

To the best of our knowledge, DISCOVER [7] is the state-of-the-art static analysis tool for detecting AC vulnerabilities. It defines and collects a list of loop characteristics for manual inspection by an analyst. It can filter out a list of loops that are unlikely to lead to AC vulnerabilities by statically analyzing their semantics. However, it can analyze only basic loops (*i.e.*, the loop body includes only one basic block), but not complex loops that include other control constructs, *e.g.*, conditionals, jumps, and inner loops, as their control flows are hard to be statically determined.

2.2.2 Computation Cost Analysis. The other approaches analyze the computation cost of program execution paths and report the paths with relatively high costs [8, 29, 30, 40]. Such approaches can cover the cases that code structure analyses cannot model, but also have their limitations.

Fuzz Testing. Fuzzing approaches generate diverse test inputs to exercise different program paths for detecting high-cost paths. Most fuzzers require significant engineering efforts to develop test harnesses. A few works [8, 39] provide automatic test generation. However, they either require an existing test harness defined by humans [39], or generate artificial test harnesses for individual methods that do not consider their calling contexts [8].

Besides, it is challenging for existing fuzzers to discover vulnerabilities residing in very deep loops, as none of them is equipped with AC vulnerability modeling. They generate offspring using general feedback information like code coverage [8], the number of executed instructions [33], or the most visited edges on a program's CFG [26]. Even though the state-of-the-art tool HotFuzz [8] proposes to fuzz over individual methods to improve its efficiency, its path exploration strategy is still inefficient. Thus it spends a lot of time on visiting sub-optimal paths in the huge program space.

Symbolic Execution. Symbolic execution (SE) approaches execute the program with symbolic values that cover multiple possible concrete values sharing the same execution path. Unlike fuzzers, they can reason about paths containing deeply nested loops. However, they also have their limitations. First, SE engines can hardly cover the entire software stack, including the runtime environment and external code, which cannot be easily traced by the executor. Second, it is computationally inefficient, especially compared to concrete execution. The problem becomes aggravated because of path explosion. In the extreme case that a program has indefinite loops, the number of execution paths can become even infinite. To mitigate this problem, some approaches propose to explore only certain interesting paths according to some branch policies [10, 29]. Although such approaches have been demonstrated effective in some simple benchmarks previously, we will show in §7.2 that the state-of-the-art work [29] is still inefficient (especially for complex programs) as it learns a branch policy from exhaustive search. Besides, their implementations do not support unreachable code or complex computation. Such cases cannot be directly handled by SE engines. Researchers have to separately model them with an extra approximation design. Further, those works require significant manual effort to set up a test, *i.e.*, users have to write symbolic instrumentation and test harness for performing symbolic execution.

To address the fundamental limitations of general symbolic execution, dynamic symbolic execution (DSE) has been proposed. DSE mixes concrete and symbolic execution [5, 9, 13, 43, 45]. It is also known as concolic testing [11, 20, 41, 42] or simply concolic execution [10, 34]. To the best of our knowledge, Badger [30] is the state-of-the-art technique that incorporates DSE in finding AC vulnerabilities, but it suffers from path explosion (§7.2). WISE [10] is a particular concolic execution tool that runs symbolic and concolic execution separately. For each target program, it first symbolically finds a high-cost path, then concolically validates the path. It cannot report a sub-optimal path when the validation fails, which makes it likely to report nothing for complex programs.

3 PROBLEM STATEMENT

In this section, we first describe in detail the adversarial capabilities and the type of AC vulnerabilities we focus on (§3.1), then discuss our research goals and the research challenges (§3.2). Finally, we provide the necessary definitions in our work (§3.3).

3.1 Threat Model

In this work, we assume a program containing AC vulnerabilities, which can be exploited with external user inputs for DoS attacks. The vulnerable program may accept user input from the program arguments or the network. An attacker can trigger the AC vulnerabilities and lead to a DoS by manipulating these inputs.

We do not specifically target well-studied types of AC vulnerabilities (*e.g.*, in a regular expression engine or a math library) that existing specialized tools cover [17, 27, 39], as we do not assume the same domain knowledge as they do. However, detecting such vulnerabilities is still in our problem scope when the source code is available.

We focus on AC time vulnerabilities and do not target AC space vulnerabilities, as we assume the attacker launches a DoS attack by consuming CPU resources. Besides, we assume routines regularly performing costly operations cannot be utilized for an attack. Instead, the attackers might exploit routines that consume distinct computation costs under different inputs. We will provide a precise definition of such vulnerabilities in §3.3.

3.2 Research Goals and Challenges

In this work, we aim to develop a fully automated tool to detect AC vulnerabilities. Specifically, we study how to model the AC vulnerabilities through code patterns that can guide us to explore two different-length execution paths selectively. Further, we aim to eliminate the manual efforts needed in the vulnerability detection workflow by developing a test harness generator, which would allow us to detect vulnerabilities in a large number of programs automatically. We do not claim that all the AC vulnerabilities we report can be transformed into working AC exploits. For instance, the worst-case path could be the typical execution path. We do aim to design a tool with a relatively low false-positive rate compared to state-of-the-art techniques. We face the following challenges.

3.2.1 Modeling AC Vulnerabilities. Inspired by DISCOVER [7], we aim to build an effective detection tool by modeling AC vulnerabilities. As we mentioned in §2.2.1, existing static analysis cannot well reason about the possible control flows inside a complex loop. Dynamic approaches would work better for that and can calculate the computation costs. However, it remains difficult to efficiently determine which control flows are vulnerable. Proposing vulnerable code patterns that apply to distinct programs with different computation costs would be quite challenging.

3.2.2 Path Explosion. Symbolic (or concolic) execution can help reason the reachability (or even the exploitability) of a vulnerable code block, but a substantially high number of execution paths potentially need to be explored. In many conditions, exploring all these paths is not feasible. To mitigate this problem, we need to focus on a limited number of paths among a huge number of possible choices. However, determining which paths to explore, *i.e.*, finding the potentially vulnerable paths, is difficult without good modeling of the vulnerable code patterns.

3.2.3 Automated Testing. In order to determine the computation cost differences of the potential vulnerabilities, it is necessary to conduct dynamic analysis. Dynamic analysis requires a test harness to execute the program and generate the test cases. Since different programs could have quite diverse logic and inputs, it is challenging to automatically generate a compatible test harness for each program. For instance, all the state-of-the-art dynamic analysis tools [26, 30, 33, 40] except for HotFuzz [8] require manual test harness configuration. Even if a test harness is available, it would still be difficult to generate test cases for efficiently searching for the high computation-cost paths.

3.3 Definitions

In this section, we first formally define the AC vulnerabilities we study in this work, then present the definitions of conditionals and branch policy generators that we will use in the latter sections.

3.3.1 AC Vulnerability. Following Crosby and Wallach [14], we define an AC witness as *two inputs* of the *same* size for the same program or function that would cause very different computation costs. We do not identify expected deterministic lengthy computations as they cannot be exploited for DoS attacks. However, this definition is not specific enough to distinguish a huge computation cost difference caused by vulnerabilities from a moderate one caused by noise or randomness. We need to apply a threshold for the computation cost difference to detect AC vulnerabilities practically.

To this end, we first propose to use *a threshold for the absolute difference in computation cost* under inputs of a specific size. However, since different programs may perform quite distinct operations consuming various resources, a fixed absolute cost difference threshold cannot be applied to all programs. Therefore, we also use *a threshold for the relative difference in computation cost* under inputs of a certain size. We report an AC vulnerability only when both the absolute and relative computation cost differences exceed the respective threshold. Since the accurate execution time might be challenging to measure (especially in symbolic execution), we estimate the computation cost by measuring the number of executed instructions in JVM. For simplicity, we assume all instructions have (approximately) the same order of computation costs. The real cost would be architecture-specific. The choices of the thresholds are studied in §7.2.

3.3.2 Conditionals and Branch Policy Generators. We define conditionals and branch policy generators to help perform selective path exploration. Conditionals (or conditional statements) are common control constructs for conditionally executing different instruction sequences in a programming language (*e.g.*, the if-else statement). A branch policy instructs an execution engine to execute a particular branch of a conditional.

Each conditional has a condition, a *true* branch and a *false* branch. Conditions are logical phrases (*e.g.*, `i > 0`). The value of a condition (or branch choice) can be either *true* or *false*.

When symbolic execution reaches a conditional, it will perform differently according to the value of its condition. A branch policy generator produces a branch policy for a conditional. The policy specifies the condition's value, hence the branch to be taken.

4 CODE PATTERN FOR COST DIFFERENCE

In this section, we present the rationale of our AC vulnerabilities modeling through conditional patterns (§4.1) and the types of conditionals we consider (§4.2).

4.1 Preliminaries

As we mentioned in §3.3, this paper defines AC vulnerability using the difference in computation cost between two execution paths. In general, a longer path is likely to have a higher computation cost than a shorter one. We consider using different branch choices to represent distinct paths, as a path can be uniquely identified by the branch choices it takes for the corresponding conditionals.

Existing work has shown that for many algorithms, the worst-case path can be captured by constantly taking particular branch choices for the conditionals [10, 29]. For example, in Listing 1, the worst-case path constantly takes the *true* branch of the while loop. However, their approach is inefficient, as they learn branch choices through an exhaustive search (§2.2.2). To address such an inefficiency issue, we investigate whether the constant branch choices can be learned without an exhaustive search, from which we can then explore slow/fast paths accordingly. By analyzing the STAC dataset, we validate that the worst-case path typically contains certain conditionals that take determined choices across different iterations— 31 out of 36 AC vulnerabilities in the STAC dataset contain at least one such special conditional. Therefore, for these conditionals, we can take a pre-determined strategy to make the particular choices for finding a slow path, and similarly find a fast path by making the opposite choices.

We define next code patterns for such special conditionals to help identify the potential AC vulnerabilities without prior knowledge.

4.2 Vulnerable Conditional Patterns

We consider a conditional vulnerable if one of its branches is likely to require a considerably higher computation cost (*i.e.*, number of instructions) than the other(s). We focus on conditional patterns related to loops and recursions, because these two common structures are the leading causes of high algorithmic complexity (and therefore AC vulnerabilities). For simplicity, in the following sections, we discuss only the loop-based patterns. The recursion-based patterns are almost identical as both loops and recursions are circles in the global CFG we build (see §5 for details).

To detect AC time vulnerabilities, we first identify vulnerable conditionals exhibiting huge computation cost difference. The cost difference generally comes from different numbers of loop iterations or different numbers of executed instructions in each iteration. Consider the insertion sort example in Listing 1, It consists of two loops, *i.e.*, the outer *for* loop and the inner *while* loop. Intuitively, both loops should be taken as many rounds as possible to allow more instruction execution. Therefore, in each round, their condition values (the values of `i < N` and `(j >= 0) && (a[j] > x)`) should be *true*. On the contrary, if the loop conditionals take the *false* branch, the loop execution will be terminated immediately, leading to lower execution costs. From another perspective, for each iteration of the outer *for* loop, the number of executed instructions is larger when the inner *while* loop is *true* rather than *false*.

We attempt to consider as many cost difference situations as possible and summarize them as the following vulnerable patterns.

4.2.1 Non-Determined Loops.

A non-determined loop might run in an unknown amount of time. Its execution (or termination) is usually determined only at runtime by dynamically evaluating some condition expressions. In particular, a loop has loop termination conditionals whose branch choices determine if the execution of the loop shall be terminated. For example, the `while` statement of a *while* loop is a loop termination conditional; the conditional that has a *break* construct in a branch is also a loop termination conditional. The computation cost can increase substantially by taking the non-termination branches compared to terminating the loop execution early, as additional instructions could get executed.

4.2.2 Single-Branch Conditionals in Loops.

Some conditionals in the loop have only one branch, *e.g.*, an *if* conditional without the *else* branch. Therefore, the computation cost difference, which is the number of instructions in that only branch, could be quite large if this branch is taken. Such conditionals are labeled vulnerable, as taking the only branch is favorable in finding a slow path.

4.2.3 Termination-Branch Conditionals in Loops.

Some control constructs, *i.e.*, `return` and `throw` statements, would terminate the function execution (hence the loops containing them) immediately. A branch with such constructs could have a lower cost than the others since the remaining instructions in the loop and function will not be executed. We call such a branch a termination branch. Conditionals with a termination branch are considered vulnerable. A slow path would avoid these branches. Such termination-branch conditionals, *i.e.*, the exceptions, are also avoided in searching for the fast paths to prevent early abnormal return or termination.

Conditionals with a throw statement that would get caught inside the same loop body are not considered vulnerable because the loop iteration might continue after handling the exception. When an exception is thrown and caught (*e.g.*, a try-catch block), the exception handling code may have a higher cost compared to the normal execution (in the try block). However, it is outside the scope of this paper because the computation costs of the exceptions are affected by the program's execution environment.

5 ACQUIRER

We develop Acquirer to address the limitations of the prior works we have introduced in §2. Acquirer employs a hybrid analysis to effectively detect AC vulnerabilities, following the workflow in Figure 1. It statically identifies the vulnerable code according to the pattern defined in §4 with an inter-procedural analysis, and generates branch policies for the vulnerable conditionals accordingly (§5.1). It then performs DSE following the branch policies to search for a slow path and a fast path and validates their computation cost difference (§5.2). Acquirer also enables full automation by generating a test harness for each potential vulnerability (§5.3). Besides, to speed up the analysis of real-world applications that contain a large number of loops, it can optionally filter some non-vulnerable loops for generating fewer test harnesses (§5.4).

5.1 Branch Policy Generation

We develop a static analysis to generate branch policies for discovering the fast and slow paths. A policy can favor either the branches leading to long (slow) execution paths or those leading to short (fast) execution paths. In the following, we generate *slow policies* in favor of the slow paths. Similarly, the *fast policies* for fast paths can be generated by negating the branch choice at each conditional.

The inter-procedural analysis is conducted over a global CFG we construct from the compiled bytecode of the target program. We enhance the global CFG with a call graph, where the call target at a call site includes all the possible callees for polymorphic calls. Therefore, both loops and recursions can be identified as circles on the enhanced global CFG graph. To enable a precise cost estimation, we expand the source code of a called function on the CFG as an inline function. Acquirer additionally supports the analysis of high-cost external methods with user annotation (§5.4).

We use the same insertion sort example in Listing 1 with its CFG in Figure 2 to illustrate how Acquirer generates slow branch policies. Specifically, it first determines a value for each condition of the vulnerable code patterns (§5.1.1), then generates a branch policy for a single loop or recursion structure (§5.1.3), and finally generates a policy for a function (§5.1.4).

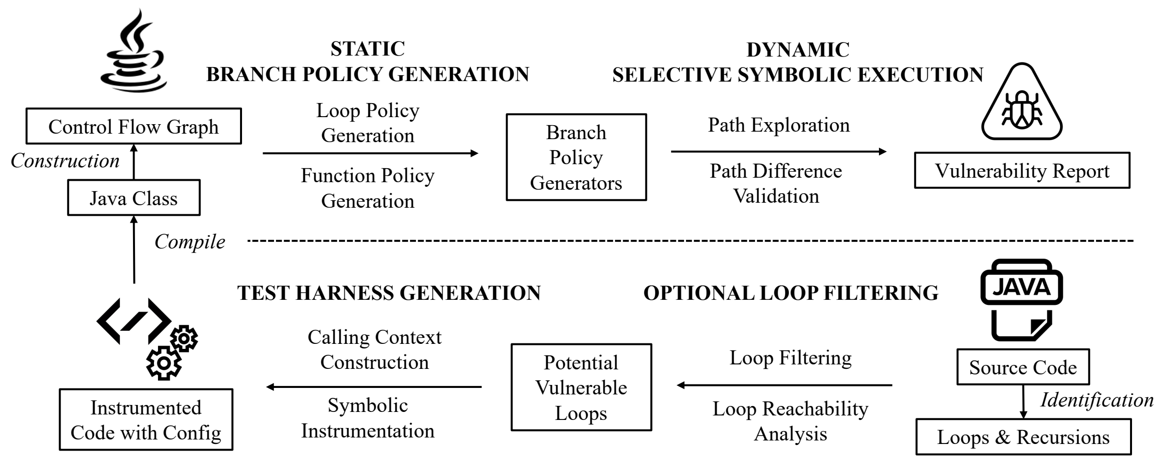

Figure 1: An architecture overview of Acquirer.

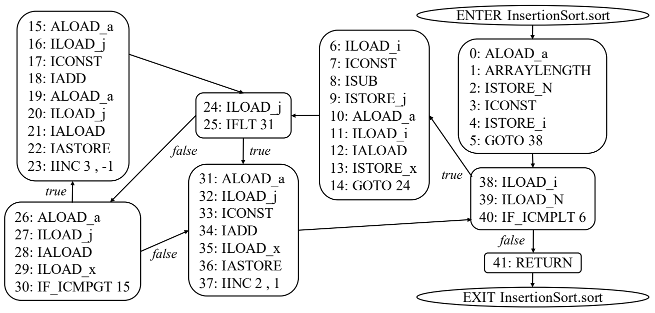

Figure 2: CFG of the Insertion Sort algorithm.

5.1.1 Determining Branch Choices. Discovering a slow path typically involves making branch choices for multiple conditionals. Acquirer first identifies the vulnerable conditional patterns in §4 from the global CFG by visiting the directed circles and their linked conditionals. A directed circle is a loop (or recursion). The termination conditionals of a loop are the ones that have edge(s) connecting to code blocks outside the loop. In Figure 2, block `24-25` is a termination conditional of the `while` loop. Acquirer can also identify the termination-branch conditionals by searching for `return` and `throw` blocks (*e.g.*, block `41`) in the loop circle. Similarly, Acquirer identifies the single-branch conditionals by finding the conditionals with two distinct transition paths to the same destination block, including a transition path consisting of a single edge. Block `24-25` is also a single-branch conditional as it can transit to block `31-37` through either the *true* branch or a path through block `26-30`.

For each identified vulnerable conditional pattern, Acquirer then takes the higher-cost branch. It sets the loop termination conditionals to continue the loop iteration, and takes the only branch of the single-branch conditionals to execute additional statements. If one conditional is involved in multiple patterns, it will merge the branch choices from different patterns into a determined one for maximizing the overall cost. For example, there are three loop termination conditionals (`38-40`, `24-25` and `26-30`) and two single-branch conditionals (`24-25` and `26-30`) in Figure 2. Their corresponding condition values should be: condition `40` - *true*; condition `25` - *false*; condition `30` - *true*; condition `25` - *false*; condition `30` - *true*.

5.1.2 Resolving Branch Choice Conflicts. In the above example, the branch choices of the two conditional patterns do not conflict with each other. However, sometimes, multiple vulnerable conditional patterns may conflict, such that selecting a slow branch of one conditional may lead to a fast branch of another conditional. We illustrate with the following examples and discuss how Acquirer handles such cases.

```
while (condition_a) {
    if (condition_b) {
        ...
        break;
    }
    ...
}
```

Listing 2: A conflict example between vulnerable patterns.

Conditional in Loop. In Listing 2, the if conditional in the while loop is both a single-branch conditional and a loop termination conditional. On the one hand, to execute the only branch of it, `condition_b` shall be assigned *true*. On the other hand, to not terminate this loop, `condition_b` shall be assigned *false*. To resolve such conflicts, Acquirer prioritizes termination-branch conditionals over non-determined loops, then over single-branch conditionals to not terminate the function execution and the loop iteration. Accordingly, to find a longer slow path, `condition_b` is assigned *false*.

Besides, a path may contain multiple loops, which conflict with each other. We discuss the following two types of such cases.

Alternative Loops. Alternative loops are loops in different branches of the same conditional. They cannot be executed on the same path. In Listing 3, the first loop gets executed when `condition_a` is *true*; the second loop gets executed, otherwise. Therefore, Acquirer considers each of the alternative loops separately.

```
if (condition_a) {
    ... // Execute Loop 1
} else {
    ... // Execute Loop 2
}
```

Listing 3: A conflict example in alternative loops.

Nested Loops. When one loop is nested within another, the condition to reach the inner one may also lead to the termination of the outer one. In Listing 4, the inner loop is reachable when `condition_b` is *true*, but the outer loop would be terminated. Acquirer gives the inner loops a lower priority than the outer ones.

```
while (condition_a) { // Loop 1
    if (condition_b) {
        while (condition_c) {...} // Loop 2
        break;
    }
}
```

Listing 4: A conflict example in nested loops.

There may exist other conflicting cases that we do not discuss, *e.g.*, when the logical value of one condition is determined to be opposite to another one. However, we believe that the above strategies cover the most common scenarios.

5.1.3 Loop-level Policy. In the previous step, each condition in the vulnerable conditional patterns is assigned a concrete boolean value to specify the choice of the longer/slower branch. Acquirer then constructs a set of condition-to-value maps for each loop in favor of the longer/slower paths accordingly. Specifically, we visit each loop L in the global CFG and collect all the concrete logical values assigned to its containing conditions C. We build a map $M_L : C \rightarrow \{true, false\}$ accordingly. In our previous example, there are two loops. The `while` loop contains two conditions `25` and `30`, which are assigned *false* and *true*, respectively. Its final condition-to-value map is $\{$`25` $\rightarrow false$, `30` $\rightarrow true\}$. Similarily, the map of the `for` loop is $\{$`25` $\rightarrow false$, `30` $\rightarrow true$, `40` $\rightarrow true\}$.

5.1.4 Function-level Policy. We learn a (slow) branch policy for a function from the (slow) policies for the loops it contains. Firstly, we update loop policies by considering the branches that need to be taken to reach the loops, and then we generate function policies using the loop policies.

Reaching Vulnerable Loops. We update a loop policy by considering outer conditionals that are related to its reachability. Starting from the loop head block, Acquirer transits backward towards the function entry block. Along the transition, it updates the condition-to-value map from the edges with a logical value.

In the example, the `while` loop is nested in the `for` loop, Acquirer transits from block `24-25` to block `ENTER`, and would go through the *true* branch of the conditional `38-40`. Therefore, the condition-to-value map of the `while` loop should be updated by adding $\{$`40` $\rightarrow true\}$. The `for` loop is not nested in other conditionals, its condition-to-value map would not be updated.

Generating Function Policy. After updating the loop policies, we generate a function policy from the loop policies. When a function has only one loop, the function policy is the loop policy. When it has multiple loops, we merge all the non-conflicting loop policies to maximize the potential cost of the entire function. Otherwise, we separately produce multiple policies from the loops, which would be consumed independently in different test harnesses. In particular, the branch choice of each condition in the function is determined jointly by the two maps in the loop policies. In the example, two updated loop policies are the same (not conflicting), therefore, we can produce a single policy for the `sort` function, *i.e.*, $\{$`25` $\rightarrow false$, `30` $\rightarrow true$, `40` $\rightarrow true\}$.

5.2 Selective Dynamic Symbolic Execution

Acquirer selectively explores a slow path and a fast path in a program guided by the corresponding branch policies in its dynamic symbolic execution. It then measures and compares the absolute and relative computation cost differences of the two paths to report a vulnerability.

5.2.1 Selective Path Exploration. Acquirer progressively explores a slow path given a slow branch policy. The slow policy instructs the dynamic symbolic execution to take branches that lead to higher computation costs. Acquirer builds and maintains a path constraint when it takes the next branch at each conditional (including a loop) following the branch policy. It then queries a constraint solver to produce a solution for the path constraint. If a solution can be found, it continues to the next conditional on the path until no satisfying solution can be generated by the solver (within a time budget). The final solution represents the concrete values (or even exploits) to take this slow path. Similarly, Acquirer can explore a fast path following a fast branch policy, which takes the other branches with lower computation costs.

Some conditional in the branch policies may not be assigned a choice. Acquirer first takes a random branch and backtracks to another branch if more vulnerable conditionals can be executed. It does take the same random branches when exploring the slow and fast paths to ensure the computation cost difference is caused by only the vulnerable conditionals.

5.2.2 Vulnerability Validation. Acquirer dynamically measures the computation cost difference between a slow path and a fast path for finally reporting an AC vulnerability. Given the corresponding path constraint solutions, Acquirer conducts concolic execution using the concrete solution values and records the execution time and the number of executed instructions along with the execution. It then computes the absolute and relative cost differences and reports a vulnerability according to §3.3.1.

To have a noticeable execution time difference in the production environment, very large inputs (*e.g.*, dozens of KBs) are usually required. Due to the inherent limitation of symbolic execution, it is very expensive and impractical to explore the paths using such inputs. Nevertheless, we find in §7.2 that the number of executed instructions is an effective metric to detect AC vulnerabilities.

5.3 Test Harness Generation

To ensure a fully automated analysis, we also develop a test harness generation algorithm allowing Acquirer to analyze a target program automatically. A test harness consists of a test execution engine and the corresponding test scripts [7]. The test script should specify the function/method to invoke, and provide the necessary calling context (*e.g.*, objects, parameters, *etc.*).

Even though one can simply start the program execution from the program's entry point, vulnerable program states are hard to reach from these entry points when the codebase gets large. Therefore, exiting works [8, 29, 30, 40] generate test harness under an

[7]Test harness: https://en.wikipedia.org/wiki/Test_harness

artificial calling context. Instead of using the minimal calling context like HotFuzz [8], or manually constructing a calling context like the other works [29, 30, 40], we seek to automatically generate a test program that provides the same calling context as the original program but includes only the necessary functions/statements. This is because the minimal calling context is less precise. A vulnerable function might not be exploitable as the attack inputs are sanitized in all contexts; whereas in an artificial test harness it might be exploitable as arbitrary arguments could be provided, causing a false positive.

We build our test program following this strategy: Starting from the main/entry function, all functions that can invoke the target function and its data-dependent functions are preserved. The irrelevant statements in those functions are removed.

A test script for running DSE needs to specify what variables to be replaced by symbols in the DSE engine. We will discuss in §6.2 how we implement that through source code instrumentation.

5.4 Non-Vulnerable Code Block Filtering

We filter loops that are not likely to be vulnerable to AC DoS attacks. This allows ACQUIRER to only explore a limited number of loops, significantly reducing the analysis time.

ACQUIRER filters the following two types of loops conservatively.

5.4.1 *Unreachable Loops.*

To exploit a vulnerable loop for launching AC DoS attacks, it must be reachable in the program, *i.e.*, there shall exist a feasible execution path from the program entry point(s) to the loop. We aim to exclude unreachable loops from the heavy test harness generation and instrumentation, but it is very difficult to statically determine the reachability of arbitrary code blocks in a program. Therefore, ACQUIRER filters only the ones that definitely cannot be executed— their containing function cannot be invoked from any program entry point(s). We do not leverage symbolic execution for its expensive computation cost. Besides, this simple rule can already help ACQUIRER exclude the majority of unreachable loops from dynamic analysis.

5.4.2 *(Almost)-Constant Cost Loops.*

Some loops have almost constant computation costs when inputs are of the same size, so they are unlikely to lead to significant computation cost differences. All possible paths in such loops require almost the same computation time. We consider the following types of such loops. Note that we filter only loops for which we can statically determine their conditional values.

Determined Loops. A determined loop runs a fixed (small) number of iterations under all user inputs. For example, a simple loop `for (i=0; i<6; i++)` is a determined loop if the iterator `i` is not changed inside the body. ACQUIRER filters loops whose iterator/counter changes monotonically in each iteration and is bounded by a constant value. In some special cases, the iterator/counter is bounded by a variable. As long as the variable is also bounded by a constant, the number of loop iterations is still determined.

Single Block Loops. A single block loop has only one basic block in its loop body. As a result, the same number of instructions is executed in each iteration. Even if the number of iterations might be affected by user inputs, the computation cost of the loop under inputs of the same size should be constant. However, the basic block might contain external method calls with non-trivially high costs. To analyze these special cases, users can mark high-cost method calls by passing a simple flag `-c {file}#{line}:...`. ACQUIRER will then not filter the marked loops. This mechanism is implemented but not adopted in the evaluation, as we do not assume such knowledge is always available and for a fair comparison with other tools.

6 IMPLEMENTATION

We implement a prototype of ACQUIRER for efficiently and accurately detecting AC vulnerabilities in Java programs. We choose Java as it has been a staple programming language for a long time. We implement our selective DSE based on the symbolic execution tool Symbolic PathFinder (SPF) [8], which is a DSE prototype that requires source-code level instrumentation. We present the essential technical details in the following.

6.1 Calling Context Construction

As the first step of test harness generation, for a target function ACQUIRER generates a test program that provides the same calling context in the original program. To preserve all control- and data-dependent functions of a target vulnerable function (§5.3), we build a partial system dependence graph (SDG) to analyze the control- and data-dependency. We implemented a complete SDG at our early research stage, and later found a simpler one is sufficient for generating the test harness targeting a specific function. We retain all the flow reasoning mechanisms (including object points-to analysis, type inference, and caller-callee matching) and incrementally build the partial SDG. Our on-demand analysis takes only a few seconds to analyze the most complex application in the STAC dataset, whereas building a complete SDG requires more than ten minutes. These flow reasoning mechanisms are implemented by searching for mapped objects from the Java compilation units. To support method calls through Java reflections and dynamic '`invoke`', we analyze the program's XML bean configuration to identify the objects (and methods) that are loaded dynamically.

The test harness of a Java program includes the code to instantiate the objects and invoke the corresponding functions and methods. In the case that ACQUIRER has to generate the object instantiation code, *e.g.*, to instantiate the receiver object of the implicit handler function, it leverages the existing object instantiation code if available, or calls the default constructor of its class. We conduct type inference for the objects in the data flow to learn the subclass they refer to, then invoke the target methods through caller-callee matching. Some arguments in the object instantiation code also need to be constructed similarly.

In some cases, an explicit calling context of a function cannot be directly found in the source code. For instance, some functions are registered as handlers and callbacks, and are implicitly dispatched when certain events happen. To test such functions, ACQUIRER identifies the dispatchers and generates the necessary code for invocation. For external code, including external libraries and other environments, ACQUIRER monitors the referred classes by constructing a new class for replacement. The new class functions as a binding layer. It implements all the methods invoked in the source code and

[8]Symbolic PathFinder: https://github.com/SymbolicPathFinder/jpf-symbc

```
1 public static void main(String[] args) {
2     final int N = 128; int[] a = new int[N];
3     for (int i = 0; i < N; i++)
4         a[i] = Debug.makeSymbolicInteger("args[i]" + i);
5         // before: a[i] = args[i];
6     sort(a);
7 }
```

Listing 5: An entrance example after symbolic instrumentation.

performs the same behavior by calling the external code. We extend the methods further in the code instrumentation step to support symbolic execution.

6.2 Code Instrumentation

Acquirer needs to further instrument the test harness to symbolize some variables (*e.g.*, program and function arguments) for symbolic execution. SPF has a `Debug` class to generate symbolic variables in primitive types, *e.g.*, Integer. We extend the primitive types of SPF to make it support some other commonly used types, *e.g.*, `java.math.BigInteger`. A primitive type variable can be directly replaced with a symbolic variable of the same type. We explain such instrumentation with Listing 5, which is the instrumented entry of the test harness for the sort function in Listing 1. Line 4 was `a[i] = args[i];`. Acquirer replaces `args[i]` with a symbolic integer created by `Debug.makeSymbolicInteger("args[i]" + i);`. A class object is instantiated from a constructor. The primitive-type arguments of the constructor can be similarly made symbolic. The type of some variables might be unknown. Acquirer can only initialize them as null. The instrumentation of instantiation of a type T can be summarized as follows.

$$I(T) = \begin{cases} \text{symbolic variable} & \text{, if } T \text{ is primitive} \\ I(I(T_0), I(T_1), \ldots) & \text{, if } T \text{ is a known class} \\ \textit{null} & \text{, otherwise.} \end{cases}$$

To approximate complex operations such as string operations, Acquirer replaces them with symbolic variables of the same type. For instance, consider a new example `String s = complex_comp(sym);` where a function performs operations on the symbolic variable `sym`, the right assignment operand is replaced with a new symbolic string.

Acquirer similarly approximates the external code. In the calling context construction, each external class is replaced with a new local class, whose methods invoke the external code. SPF cannot handle the symbolic variables in such methods as the detailed computation in the external code is unknown. Therefore, when these methods accept a symbolic variable as a parameter, we directly return a new symbolic variable of the return type instead of invoking the external code.

6.3 Dynamic Symbolic Execution

Acquirer drives SPF to run mixed symbolic and concolic execution toward a specified target following the corresponding branch policies. Along with the symbolic execution, it collects constraints from not only the conditions (and their corresponding values) it meets, but also all the direct or indirect data and control dependencies of the variables in these conditions. Using these constraints, Acquirer can assign a concrete value for each symbolic variable. It then validates the slow and fast execution paths with these concrete values in concolic execution.

7 EVALUATION

We extensively evaluate the effectiveness of Acquirer. We first demonstrate its effectiveness in verifying known AC vulnerabilities under several standard benchmarks (§7.2) in comparison with existing tools. We further show that it can automatically detect (unknown) vulnerabilities from a widely-used large dataset containing 45 applications (§7.3) and characterize the vulnerabilities (§7.4). Finally, we apply it to detect unknown AC vulnerabilities in popular real-world applications (§7.6). We describe the experiment setup next.

7.1 Datasets and Setup

We compare Acquirer on vulnerability detection effectiveness and efficiency with four state-of-the-art AC vulnerability detection tools: 1) Singularity [40], a pattern fuzzing tool for detecting the worst-case complexity of a program; 2) SPF-WCA [29], a worst-case path learner based on symbolic execution; 3) Badger [30], a complexity analysis tool based on fuzzing and concolic execution; and 4) HotFuzz [8], a method-level genetic fuzzing tool for AC vulnerability detection.

All the tools had been evaluated on the DARPA Space and Time Analysis for Cybersecurity (STAC) dataset, which collects programs vulnerable to AC attacks and side-channel attacks. However, only HotFuzz was able to automatically analyze the entire dataset, while the others selected several benchmarks extracted from the dataset in their evaluation. Nevertheless, we are unable to apply the three tools to the entire dataset. First, they cannot run automatically. They require significant manual efforts on instrumentation, writing test harnesses, and verifying the result for each case. These test harnesses are not included in their released code [1–3]. Second, SPF-WCA and Badger use manually-written custom test harnesses and drivers to analyze a *known* vulnerable function. It would be very difficult to use them to detect *unknown* vulnerabilities. Besides, the design of the test drivers affects their performance significantly. As a result, we are unable to conduct a fair comparison with them on the full STAC dataset.

Therefore, we include another dataset—the Worst-case Inputs from Symbolic Execution (WISE) benchmark dataset [10], which we obtain from [1]. It contains the implementation of nine algorithms of different complexities. Singularity and SPF-WCA analyzed the entire WISE dataset, and Badger verified two benchmarks from it.

As HotFuzz is not publicly available, we compare it with Acquirer using their results on the STAC dataset in the paper [8]. We also discuss the vulnerabilities that other tools verified on the STAC dataset. The STAC dataset we use contains two sets of engagement challenges including 45 programs: `Engagement_2` and `Engagement_4`. We use the WISE dataset for comparison with the other three tools.

All the experiments were performed on an 8-core Intel Xeon W-2123 desktop with 16 GB RAM running Debian 10.

7.2 Benchmark Analysis

We use the benchmarks in the WISE dataset to learn the thresholds used by Acquirer for detecting AC vulnerabilities. We also include the commonly tested `Insertion Sort` algorithm in the evaluation. We compare with Singularity, SPF-WCA and Badger on the efficiency of analyzing the worst-case performance of an algorithm.

Table 1: Detailed evaluation result of the benchmarks.

Benchmark	Input Size (N)	Absolute Diff.	Relative Diff.	Complexity Range
`Sorted Linked-List`	50	13,975	3.46	$O(n) - O(n^2)$
`Heap Insert`	1000	372,377	4.41	$O(n) - O(n \log n)$
`Red-Black Tree`	1000	16,463	1.02	$O(n \log n)$
`Quicksort`	60	14,157	2.34	$O(n \log n) - O(n^2)$
`Binary Search Tree`	40	13,533	3.87	$O(n \log n) - O(n^2)$
`Merge Sort`	1000	53,160	1.62	$O(n \log n)$
`Bellman-Ford`	9	29,410	6.09	$O(n) - O(n^3)$
`Dijkstra's`	30	52,583	5.75	$O(n \log n) - O(n^2)$
`Traveling Salesman`	7	197,544	36.7	$O(n) - O(n!)$
`Insertion Sort`	50	20,531	4.65	$O(n) - O(n^2)$

Table 2: Average Analysis time on the WISE dataset.

	Acquirer	SPF-WCA	Badger	Singularity
Avg. Analysis Time	92.6 secs	16.5 mins	87.1 mins	3 hrs

7.2.1 Results. The overall results are shown in Table 1. For each benchmark, Acquirer can find the worst-case complexity (slow) path and the best-case complexity (fast) path.

Suppose T_A and T_R are the absolute and relative computation cost difference thresholds, respectively. We empirically determine their values according to the results of algorithms with variable best-case and worst-case complexities. These thresholds are then used to evaluate Acquirer on the STAC dataset and real-world apps. In particular, the `Red-Black Tree` and `Merge Sort` algorithms have the same best-case and worst-case complexity. We filter these two cases and preserve the others by setting $T_A = 1e5$ and $T_R = 2$.

For the following experiments, Acquirer iteratively tries input sizes ranging from 10 to 1000 (*i.e.*, [10, 20, 50, 100, 200, 500, 1000]), and stops when it meets a vulnerability. We set a maximum two-minute validation time for each input, regardless of the input sizes.

7.2.2 Comparison. We present the average analysis time of each tool in Table 2. Singularity, SPF-WCA and Badger can find the worst-case inputs/paths of these algorithms because they can fall back to an exhaustive path search strategy. However, their performance was rather slow. For example, Singularity spent three hours on each benchmark. Even though it could find the worst-case input with $N = 1000$ for all cases, it was less efficient for vulnerability detection, which we further compare in §7.3. Badger runs more efficiently than a traditional fuzzer by integrating a concolic execution. For the case `Insertion Sort`, it generated an input to cause a slowdown close to the worst-case after 61 minutes. However, the concolic execution did not find the worst-case input, because it could explore only a few inputs within a fixed time budget.

In comparison, Acquirer can complete the analysis more efficiently. It took Acquirer only 97.012 seconds to find the worst-case path of `Red-Black Tree` with $N = 1000$, and less than a minute for all other benchmarks. In practice, Acquirer can detect and validate a vulnerability in several minutes. We will further discuss its performance advantage in §7.3.4.

Comparing to Badger, Acquirer could find the worst-case input ($N = 64$, the size chosen in Badger) of `Insertion Sort` in only 126.74 seconds. For `Quicksort`, Badger took between 20 and 150 minutes to get its best slowdown. Other symbolic execution tools like SPF-WCA (and WISE) also did not perform well in this case. These tools would conduct an exhaustive search on a small input size to help the worst-case search in larger input sizes. However, the `Merge Sort` behaves differently for $N \leq 7$ and $N > 7$, forcing them to conduct an inefficient exhaustive search on $N = 8$.

Table 3: Vulnerability detection on the STAC dataset.

Tool	# of TP	# of Report	Precision	Analysis Time
HotFuzz	5	57	0.078	16.5 hrs
Acquirer	22	41	0.537	3.8 hrs
	+11		+0.268	(with validation)

7.3 Automatic Vulnerability Detection

We evaluate the effectiveness of Acquirer in detecting AC vulnerabilities using the STAC dataset. We first introduce additional dataset information, then conduct comparison with HotFuzz and the other tools, and finally evaluate the detection efficiency of Acquirer.

7.3.1 The STAC Dataset. The DARPA STAC dataset contains several challenge problems for each program and provides a document describing the vulnerabilities included. These challenge problems ask whether a certain vulnerability exists in the program. The dataset provides ground truth answers for these problems, but no details about the specific vulnerable functions. Therefore, for a challenge problem, we manually collect a set of corresponding vulnerable functions using two approaches: 1) if the exploit is provided in the dataset, we apply the exploit and find the functions in the call stack that match the description; 2) otherwise, we manually locate vulnerable functions according to the description. We exclude non-AC time vulnerabilities from the STAC dataset as all the other tools do not handle non-AC time vulnerabilities, either. We conclude that a vulnerability is a *true positive* if the reported function is among the vulnerable functions we collect from the dataset.

Note that the public STAC dataset we use contains only 45 engagement challenges, which are fewer than the 80 (including many non-public ones) in the dataset HotFuzz used. In particular, the challenge `inandout_2`, which HotFuzz reported as a true positive, is not in our dataset. Even using a smaller dataset, Acquirer detected more vulnerabilities with better precision.

We noticed many duplicate vulnerabilities in the STAC dataset. These duplicates do not refer to different program functionalities. Therefore, we deduplicate the vulnerabilities and report only the unique ones. Specifically, many challenges with minor differences are from the same program, *e.g.*, the program `textcrunchr` contributes to seven challenges. The same vulnerable package could also exist in different programs. `snapbuddy`, `textcrunchr` and `gabfeed` all include package `com.cyberpointllc.stac.hashmap`, which implements an unbalanced tree-based hash table. Besides, an exploit may invoke multiple vulnerable functions. For example, similar worst-case conflicts exist in the hash insert and search algorithms.

7.3.2 Comparison with HotFuzz. The overall results are shown in Table 3. Specifically, Acquirer detected 22 *unique* known vulnerabilities, and another 11 new ones that also have a noticeable execution time difference. We manually verified all the vulnerabilities Acquirer detected. We consider these previously unknown ones as true positives because they have a more significant impact than some known ones.

One new vulnerability lies in the `EnigmaMachine` class of the `textcrunchr_1` challenge. The `encodeLine` function visits every character in the input `s` and selectively performs a heavy encoding operation depending on the character value. We generated an end-to-end exploit showing that the execution time difference would be 5.6 seconds under a 130K-long string. It is more severe

Table 4: Benchmark verification on the STAC dataset.

	SPF-WCA	Badger	Singularity	In Total
# of Verification	4	2	4	7

than a known vulnerability, which is a sorting algorithm with a worst-case complexity $O(n^2)$. The sorting algorithm required an over 200K-long string to reach the same 5.6-second execution time difference.

Acquirer has higher precision than HotFuzz, because of the following reasons. First, Acquirer verifies a vulnerability by comparing two paths instead of evaluating only the slow path performance. Second, it avoids false positives caused by unreachable ones. Third, it preserves as much context as possible in constructing test harnesses to provide a realistic execution environment.

However, Acquirer still reported eight false positives (FP). Four are related to the difficulty in manipulating the values in a set. Specifically, in two cases, the worst case can occur only when a user can manipulate the element order of a list. But this list is converted from a set, so it would be hard to control the order. In the other two FP cases, the worst cases exist when a list contains duplicate elements, which are infeasible because they are converted from sets. Another two FP cases contain complex operations that were over-approximated in the code instrumentation. For the final two cases, the worst-case paths are typical execution paths, while the faster ones handle exceptions. We further discuss that Acquirer is more efficient than HotFuzz in §7.3.4.

7.3.3 Comparison with Other Tools. SPF-WCA, Badger and Singularity verified 4, 2 and 4 cases, respectively, and 7 cases in total from the STAC dataset. Acquirer performs better than all these tools combined. It did not detect only one case related to an internal math function, which does not contain any vulnerable code structures we define. However, fuzzers (Badger and Singularity) could detect it because they do not assume specific code patterns.

Singularity failed on the cases `textcrunchr` and `airplan_3`. For `textcrunchr`, it can only learn sub-optimal patterns even using a 1KB input. For `airplan_3`, the program itself took a long time, exceeding the time budget. Acquirer succeeded in both because its static analysis efficiently identified vulnerable code blocks.

Due to the limitation we will discuss in §8, Acquirer cannot fully automatically generate test harnesses for some cases, leading to 8 false negatives. However, with slight manual efforts like SPF-WCA, Acquirer could still detect them in a semi-automated way.

7.3.4 Performance Analysis. We measured the performance of Acquirer on analyzing the STAC dataset. It spent a total 3.8 hours analyzing the entire dataset and validating reported vulnerabilities. It is more efficient than HotFuzz, which took 16.5 hours on only fuzzing before validation. Specifically, Acquirer's static loop filtering and test harness generation took 4.56 minutes (2%); the branch policy generation took 27.46 minutes (12%); the selective symbolic execution took 3.268 hours (86%). The most expensive step is the selective dynamic symbolic execution, as it would wait until both the absolute and relative cost differences exceed the thresholds. In practice, the absolute cost difference threshold could be quickly passed for many cases. Acquirer spent most of the time exploring paths to pass the relative cost difference threshold.

Table 5: Characterization of vulnerabilities in the STAC dataset.

Group	Vulnerability Type	Count	Size / Time
Out of Scope	Math Computation	4	70KB / 1Ks
	Regular Expression	1	5KB / 300s
	Branch Only	1	800KB / 1.2Ks
True Positive	Graph Search/Computation	5	infinite loop
	Sorting Algorithm	4	25KB / 500s
	(De)-Compression	4	400KB / 300s
	Unbalanced Tree	3	400KB / 150s
	URL/XML parser	2	25KB / 500s
	Score Computation	2	400KB / 300s
	Logging	1	3KB / 30s
	Spell Checking	1	2KB / 2Ks
Fail to Handle	Inherited Handler	5	400KB / 300s
	Hash Collision	1	400KB / 300s
	Others	2	5KB / 30s

7.4 Characterization of Vulnerabilities

We characterize the vulnerabilities in the STAC dataset by analyzing all the 36 unique known bugs in the STAC documentation, and present the results in Table 5. The last column shows the concrete execution time under inputs of a certain size. For vulnerabilities of the same type, we only show the severest result. In general, all the vulnerable programs run for at least 30 seconds, showing that the vulnerabilities could be exploited for DoS attacks.

The first group contains vulnerabilities that are out of our detection scope, *i.e.*, caused not by complex loops. Most of them are about math computation. The complexity difference is in the math library instead of the program source code in these cases. Another case is about regular expression. The vulnerability lies in the external regex library. Existing works have studied this type of AC vulnerability well [16, 27, 37]. The final case contains only branches. The program performs some high-cost operations only when specific conditions are met. These operations do not have algorithmic complexity differences.

All the other 30 cases involve vulnerable loop structures, showing that loop-based AC vulnerabilities are common in practice. The "True Positive" group includes the ones that Acquirer successfully detected. We classified them according to their functionalities. We highlight three types of functionalities that are commonly vulnerable to AC DoS attacks: graph search/computation, sorting algorithm, compression/decompression algorithm. These algorithms are vulnerable for two reasons: 1) graph computation algorithms typically have different average-case and worst-case performances; 2) visiting a node multiple times may potentially lead to an infinite loop. Similarly, sorting algorithms have different average-case and worst-case performances. Developers need to handle the worst-case inputs to carefully mitigate these vulnerabilities. Compressors, parsers and spell-checking would conduct different operations on different inputs (of the same size). They can be even more vulnerable when nested traversing (in compressors) and backtracking (in parsers) are allowed. Existing works have discussed these vulnerabilities in detail [18, 35]. The last group contains the cases our current implementation of Acquirer fails to support.

7.5 Component-wise Analysis

We show earlier that Acquirer outperforms the state-of-the-art techniques in efficiency and effectiveness. In this section, we provide a component-wise analysis to show how our design choices contribute to the good performance of Acquirer. In general, the

Table 6: Loop filtering on the STAC dataset.

Avg. # of Filtered Loops	135
Avg. # of Filtered Functions	129
Avg. # of Remaining Functions	28
Avg. # of Filtered Code Blocks	3,527
Avg. Saved Analysis Time	12.8 mins

Table 7: Additional Strategies on the STAC dataset.

Total	Conflict Resolution	Calling Context Construction	Code Summary
16	2	11	5

optional loop filtering and the selective dynamic symbolic execution ensure the high efficiency of Acquirer. The effective branch policies, together with reasonable approximation in test harness generation and constraint solving, enable Acquirer to effectively detect AC vulnerabilities that prior works cannot find.

Optional Loop filtering. The majority of the non-vulnerable loops (functions) could be optionally filtered by Acquirer while introducing no false negative on the STAC dataset. We present the number of filtered non-vulnerable code blocks and the saved time on average in Table 6. Acquirer on average filtered the majority of functions—129 functions, leaving only 28 functions to analyze on average. Loop filtering prevents these 129 functions from being the target of the costly concolic execution and saves us a large amount of analysis time (69.7% or 12.8 minutes). At the same time, the filtering does not introduce new false negatives on the STAC dataset. We observe that most filtered loops are pretty simple (the loop body contains less than three statements without any conditional). Our filtering strategy can distinguish them well from vulnerable patterns and save a lot of analysis time.

Selective Dynamic Symbolic Execution. Selective dynamic symbolic execution is the foundation of our efficient and effective detection. Without the proper guidance of the branch policies, the execution time becomes significantly (8 times) higher, and many vulnerabilities cannot be detected in a large amount of time. We set a one-hour timeout for running without the guidance of branch policy. The majority of previously detected cases fail under the time limit when we disable branch policies. Acquirer can only report 9 out of 41 cases compared to the analysis guided by branch policies. The analysis time also rises 9.1 times (from 4.9 minutes to 44.7 minutes). This is because we would meet path explosion and become unable to detect most cases within the time limit.

Additional Strategies. Our conflict resolution strategies, calling context construction, and external code summaries also contribute to the effectiveness of Acquirer. We list in Table 7 the number of vulnerabilities that cannot be detected without these additional strategies. The statistics show that the calling context construction is the most critical component, without which 11 vulnerabilities cannot be detected. This is because the vulnerable code blocks are far from the program entry points. The SE engine would waste most of its execution time exploring a path to the vulnerable function instead of searching for fast/slow paths. Besides, our external code summary contributes to the detection of five vulnerabilities. Without a proper code summary, we cannot run the SE engine to analyze them as some objects cannot be resolved. Another two vulnerabilities require conflict resolution. We cannot produce a valid branch policy for them without our conflict-resolving strategy.

7.6 Detecting Real-World Vulnerabilities

In this section, we show Acquirer can detect unknown vulnerabilities in real-world Java programs. We focus on Java web applications since they naturally accept user inputs and are potentially exploitable. We searched on GitHub for popular (with more than 600 stars) Java projects using the keywords: "web app", "web framework", "platform", and "database". We filtered out the irrelevant projects manually and analyzed the remaining 46 applications.

Even though the real-world applications are much more complex than those in the STAC dataset (the codebases are 7.4 times larger on average), Acquirer can still effectively analyze them under 18.2 minutes on average (only 3.2 times longer). This is because Acquirer models the complex external code (environment) in the constructed calling context and focuses on only a limited number of non-determined loops. Acquirer reported 24 unique vulnerabilities in the 46 applications. We manually verified the vulnerabilities and found that 11 vulnerabilities could be transformed into AC witnesses. We report a vulnerability when the worst-case performance is evitable, *i.e.*, we can provide an implementation that eliminates the worst case and ensures the necessary functionalities. Five developers have already fixed the corresponding bugs according to our patch suggestion. Two developers thanked our report but thought it was not necessary for fixing such a potential performance issue. The remaining four developers have not responded to our report. We discuss several representative cases next.

Compressor. Hackpad (3.4K stars) is a web-based real-time wiki, based on a collaborative document editor. One function `extractDataUrls` in the CssCompressor class contains an AC vulnerability. An attacker could use a specific input to trigger DoS on the server. Other users of this web application would therefore get affected.

The code snippet tries to extract URLs from a given string by finding terminator characters iteratively. However, it does not properly handle the case when the input string does not contain any terminator. When a simple string without a terminator is provided, *e.g.*, `"url(data::"`, the function would run forever.

Sorting Algorithm. Alink (2.8K stars) is a machine learning platform developed in Java. The `sortImpl` function in its SparseVector class contains an AC vulnerability. The function implements a Quicksort algorithm with a center-most pivot. The average complexity of this algorithm is $O(n \log n)$, but a well-designed malicious input, *e.g.*, the array [2, 4, 6, 8, 10, 11, 3, 7, 1, 9, 5] can exploit the worst-case complexity of this algorithm, which is quadratic. A well-designed input would take 11.94 more seconds to sort compared to a random one when both inputs contain 300K elements.

Graph Computation. BTrace (4.7K stars) is a safe and dynamic tracing tool for the Java platform. The `findCycles` function contains a performance issue. For the worst case, the function would run for 15.37 seconds with a 50K nodes graph input, which takes 15.32 seconds more than an average case.

The function implements a cycle detection algorithm that removes nodes without outgoing edges in each iteration. The function traverses all the nodes in the graph again whenever any change is made (marked by the `changesMade` flag), which causes the time consumption to be related to the input graph structure. For example, it takes linear time on processing an acyclic graph input, but quadratic time for a chain graph of the same size. With massive inputs, the time consumption difference can be considerable.

8 LIMITATIONS

Design Limitations. Our vulnerable code patterns do not cover differential program behaviors caused not by conditionals, *e.g.*, math computation. Our summaries of external code and calling context construction inevitably involve some approximation and may import some false positives. Such an approximation does introduce a few false positives in our test set (§7.3.2). Our conflict resolution strategies are not comprehensive. They maximize the cost of only some conditional patterns while neglecting others. This may bring potential false negatives, as it prevents us from finding the slow paths when the neglected patterns are more costly. However, we do not observe such cases in our experiments. Further, the slow path we report is not necessarily the worst-case path by definition, as finding the worst-case path among all possible paths is NP-hard. Nevertheless, we are confident that these slow paths are computationally much more expensive than many other paths. In practice, this is enough for vulnerability detection.

Implementation Limitations. The dynamic symbolic execution tool we employ requires source-code level instrumentation, so we cannot support the analysis of compiled libraries. In our static analysis, we assume that the cost of a function from an external library is constant. We cannot fully automate the test harness generation for some classes, including nested classes, abstract classes, or classes that directly extend classes in external libraries. Besides, our test harness generation uses the `symbolsolver` in the JavaParser. Due to its incomplete implementation, the return type of some expressions cannot be determined.

9 RELATED WORKS

Algorithmic Complexity Denial-of-Service. In 2003, Crosby *et al.* found a new type of Denial-of-Service attack that exploits the algorithmic complexity of programs [14]. Later, AC DoS is found harmful in many application scenarios. Gal *et al.* noted that mobile code systems could be vulnerable to AC DoS attacks [19]. Chang *et al.* proved the existence of AC vulnerabilities in real-world networked applications [12]. Czubak and Szymanek exploited the AC vulnerabilities to attack the firewall [15]. David proposed a better zip bomb attack [18]. Pellegrino *et al.* measured the DoS attacks on network services [32]. Altmeier *et al.* measured how DoS attacks affect web services [4].

Algorithmic Complexity Measurement. The severity of program performance has brought many works that measure the worst-case performance of a program. Goldsmith *et al.* proposed a method for measuring the empirical computational complexity [21]. Holland *et al.* improved analysis capabilities by computing relevant program behaviors [23]. Toffola *et al.* investigated how to expose performance bottlenecks [39]. Specifically, many researchers focus on analyzing the complexity of loops. Xie *et al.* classified several types of loop structure to understand their complexity [44]. Song and Lu proposed a static-dynamic hybrid analysis tool that provides accurate performance diagnosis for loops [36]. Padhye and Sen proposed a dynamic analysis technique for detecting data-structure traversals [31]. Han *et al.* used the information extracted from performance bug reports to generate test frames for guiding actual performance test case generation [22].

Detecting AC Vulnerabilities in Java. Some previous works have studied the detection of AC vulnerabilities in Java. Burnim *et al.* proposed WISE to find the worst-case complexity of Java programs [10]. Holland *et al.* proposed a pragmatic engineering approach using statically-informed dynamic (SID) analysis and two tools to provide critical capabilities for detecting AC vulnerabilities [24]. Luckow *et al.* implemented a symbolic analysis framework [28]; they then proposed a search policy for analyzing the worst-case path of Java programs [29]. Wei *et al.* transferred the worst-case asymptotic complexity problem into optimal program synthesis by looking for input pattern instead of concrete input [40]. Noller proposed a hybrid approach that uses fuzzing and symbolic execution in tandem to discover vulnerabilities [30]. Awadhutkar *et al.* presented a suite of tools that facilitates human-on-the-loop detection of Algorithms Complexity vulnerabilities, which supports analysis of Java source code and Java byte code [7]. Blair *et al.* used micro-fuzzing to automatically discover AC vulnerabilities in Java libraries [8].

General AC Vulnerabilities Detection. Khan and Traore proposed a model based on a regression analysis that can prevent algorithmic complexity attacks [25]. Chang *et al.* presented a static analysis tool called SAFER for identifying DoS vulnerabilities and the root causes of resource exhaustion attacks [12]. Petsios *et al.* implemented a domain-independent framework for automatically finding algorithmic complexity vulnerabilities using resource-usage-guided evolutionary search techniques [33]. Tizpaz *et al.* produced decision-tree discriminants that are useful for detecting timing vulnerabilities [38]. Awadhutkar *et al.* investigated techniques to amplify intelligence so that the analyst can gain a deeper knowledge of complex loops that is necessary to discover AC vulnerabilities [6]. Lemieux *et al.* proposed a tool that can be used to generate inputs that demonstrate algorithmic complexity vulnerabilities [26].

10 CONCLUSION

Algorithmic Complexity (AC) Denial-of-Service attacks can severely degrade the availability of applications and their hosting servers. In this paper, we present Acquirer, a hybrid analysis tool that automatically detects AC vulnerabilities in Java programs. It statically locates potentially vulnerable structures in the target program, then dynamically examines whether there exist two different execution paths with a large cost difference. To improve the analysis efficiency, it leverages a selective path exploration strategy guided by branch policies. We thoroughly evaluated the effectiveness and efficiency of Acquirer on two widely-used datasets and on real-world popular applications. We demonstrated that Acquirer significantly outperformed the state-of-the-art detection tools, and can detect unknown AC vulnerabilities effectively and efficiently.

ACKNOWLEDGMENT

The work described in this paper was partly supported by a grant from CUHK (Project No.: 4055175) and a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14210219).

REFERENCES

[1] 2017. Tool for algorithmic complexity analysis based on symbolic execution. https://github.com/isstac/spf-wca.

[2] 2018. Badger: complexity analysis with fuzzing and symbolic execution. https://github.com/isstac/badger.

[3] 2018. Pattern Fuzzing for Worst-Case Algorithmic Complexity. https://github.com/MrVPlusOne/Singularity.

[4] Christian Altmeier, Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. 2015. Adidos–adaptive and intelligent fully-automatic detection of denial-of-service weaknesses in web services. In *Data Privacy Management, and Security Assurance*. Springer, 65–80.

[5] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. Hyderabad, India.

[6] Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, and Suresh Kothari. 2017. Intelligence amplifying loop characterizations for detecting algorithmic complexity vulnerabilities. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 249–258.

[7] Payas Awadhutkar, Ganesh Ram Santhanam, Benjamin Holland, and Suresh Kothari. 2019. DISCOVER: Detecting Algorithmic Complexity Vulnerabilities. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Tallinn, Estonia.

[8] 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 2020 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[9] Suhabe Bugrara and Dawson Engler. 2013. Redundant state detection for dynamic symbolic execution. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*. San Jose, CA.

[10] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. Vancouver, Canada, 463–473.

[11] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. 2008. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)* 12, 2 (2008), 1–38.

[12] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. 2009. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *22nd IEEE Computer Security Foundations Symposium*.

[13] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding dynamic symbolic execution toward unverified program executions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. Austin, TX.

[14] Scott A Crosby and Dan S Wallach. 2003. Denial of Service via Algorithmic Complexity Attacks.. In *Proceedings of the 12th USENIX Security Symposium (Security)*. Washington, DC.

[15] Adam Czubak and Marcin Szymanek. 2017. Algorithmic complexity vulnerability analysis of a stateful firewall. In *Information Systems Architecture and Technology: Proceedings of 37th International Conference on Information Systems Architecture and Technology–ISAT 2016–Part II*. Springer, 77–97.

[16] James C Davis, Christy A Coghlan, Francisco Servant, and Dongyoon Lee. 2018. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Lake Buena Vista, FL.

[17] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil pickles: DoS attacks based on object-graph engineering. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[18] David Fifield. 2019. A better zip bomb. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*.

[19] Andreas Gal, Christian W Probst, and Michael Franz. 2005. Complexity-based denial-of-service attacks on mobile code systems. *INSTITUT FUR INFORMATIK UND PRAKTISCHE MATHEMATIK* (2005), 1–10.

[20] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Chicago, IL.

[21] Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. 2007. Measuring empirical computational complexity. In *Proceedings of the 12th European Software Engineering Conference (ESEC) / 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Dubrovnik, Croatia.

[22] Xue Han, Tingting Yu, and David Lo. 2018. PerfLearner: learning from bug reports to understand and generate performance test frames. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Montpellier, France, 17–28.

[23] Benjamin Holland, Payas Awadhutkar, Suresh Kothari, Ahmed Tamrawi, and Jon Mathews. 2018. COMB: Computing relevant program behaviors.. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. Gothenburg, Sweden.

[24] Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. 2016. Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 79–84.

[25] Suraiya Khan and Issa Traore. 2005. A prevention model for algorithmic complexity attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 160–173.

[26] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*. Amsterdam, Netherlands.

[27] Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[28] Kasper Luckow, Marko Dimjašević, Dimitra Giannakopoulou, Falk Howar, Malte Isberner, Temesghen Kahsai, Zvonimir Rakamarić, and Vishwanath Raman. 2016. JDart: A Dynamic Symbolic Analysis Framework. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Eindhoven, The Netherlands.

[29] Kasper Luckow, Rody Kersten, and Corina Păsăreanu. 2017. Symbolic complexity analysis using context-preserving histories. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 58–68.

[30] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. 2018. Badger: complexity analysis with fuzzing and symbolic execution. In *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA)*. Amsterdam, Netherlands.

[31] Rohan Padhye and Koushik Sen. 2017. Travioli: A dynamic analysis for detecting data-structure traversals. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. Buenos Aires, Argentina, 473–483.

[32] 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 24th USENIX Security Symposium (Security)*. Washington, DC.

[33] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.

[34] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[35] Randy Smith, Cristian Estan, and Somesh Jha. 2006. Backtracking algorithmic complexity attacks against a NIDS. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 89–98.

[36] Linhai Song and Shan Lu. 2017. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. Buenos Aires, Argentina, 370–380.

[37] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[38] Saeid Tizpaz-Niari, Pavol Cerný, Bor-Yuh Evan Chang, Sriram Sankaranarayanan, and Ashutosh Trivedi. 2017. Discriminating Traces with Time. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Uppsala, Sweden.

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

[40] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 2018. Singularity: pattern fuzzing for worst case complexity. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Lake Buena Vista, FL.

[41] Nicky Williams, Bruno Marre, and Patricia Mouy. 2004. On-the-fly generation of k-path tests for C functions. In *Proceedings of the 19th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Linz, Austria.

[42] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. 2005. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *European Dependable Computing Conference*. Springer, 281–292.

[43] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. 2009. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. 359–368.

[44] 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 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Seattle, WA.

[45] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular property guided dynamic symbolic execution. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy.