SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities

Theofilos Petsios
theofilos@cs.columbia.edu
Columbia University

Jason Zhao
zhao.s.jason@columbia.edu
Columbia University

Angelos D. Keromytis
angelos@cs.columbia.edu
Columbia University

Suman Jana
suman@cs.columbia.edu
Columbia University

Abstract

Algorithmic complexity vulnerabilities occur when the worst-case time/space complexity of an application is significantly higher than the respective average case for particular user-controlled inputs. When such conditions are met, an attacker can launch Denial-of-Service attacks against a vulnerable application by providing inputs that trigger the worst-case behavior. Such attacks have been known to have serious effects on production systems, take down entire websites, or lead to bypasses of Web Application Firewalls.

Unfortunately, existing detection mechanisms for algorithmic complexity vulnerabilities are domain-specific and often require significant manual effort. In this paper, we design, implement, and evaluate SlowFuzz, a domain-independent framework for automatically finding algorithmic complexity vulnerabilities. SlowFuzz automatically finds inputs that trigger worst-case algorithmic behavior in the tested binary. SlowFuzz uses resource-usage-guided evolutionary search techniques to automatically find inputs that maximize computational resource utilization for a given application.

We demonstrate that SlowFuzz successfully generates inputs that match the theoretical worst-case performance for several well-known algorithms. SlowFuzz was also able to generate a large number of inputs that trigger different algorithmic complexity vulnerabilities in real-world applications, including various zip parsers used in antivirus software, regular expression libraries used in Web Application Firewalls, as well as hash table implementations used in Web applications. In particular, SlowFuzz generated inputs that achieve 300-times slowdown in the decompression routine of the `bzip2` utility, discovered regular expressions that exhibit matching times exponential in the input size, and also managed to automatically produce inputs that trigger a high number of collisions in PHP's default hashtable implementation.

1 INTRODUCTION

Algorithmic complexity vulnerabilities result from large differences between the worst-case and average-case time/space complexities of algorithms or data structures used by affected software [31]. An attacker can exploit such vulnerabilities by providing specially crafted inputs that trigger the worst-case behavior in the victim software to launch Denial-of-Service (DoS) attacks. For example, regular expression matching is known to exhibit widely varying levels of time complexity (from linear to exponential) on input string size depending on the type of the regular expression and underlying implementation details. Similarly, the run times of hash table insertion and lookup operations can differ significantly if the hashtable implementation suffers from a large number of hash collisions. Sorting algorithms like quicksort can have an $O(nlogn)$ average-case complexity but an $O(n^2)$ worst-case complexity. Such worst-case behaviors have been known to take down entire websites [22], disable/bypass Web Application Firewalls (WAF) [6], or to keep thousands of CPUs busy by merely performing hash-table insertions [19, 24].

Despite their potential severity, in practice, detecting algorithmic complexity vulnerabilities in a domain-independent way is a hard, multi-faceted problem. It is often infeasible to completely abandon algorithms or data structures with high worst-case complexities without severely restricting the functionality or backwards-compatibility of an application. Manual time complexity analysis of real-world applications is hard to scale. Moreover, asymptotic complexity analysis ignores the constant factors that can significantly affect the application execution time despite not impacting the overall complexity class. All these factors significantly harden the detection of algorithmic complexity vulnerabilities.

Even when real-world applications use well-understood algorithms, time complexity analysis is still non-trivial for the following reasons. First, the time/space complexity analysis changes significantly even with minor implementation variations (for instance, the choice of the pivot in the quicksort algorithm drastically affects its worst-case runtime behavior [30]). Reasoning about the effects of such changes requires significant manual effort. Second, most real-world applications often have multiple inter-connected components that interact in complex ways. This interconnection further complicates the estimation of the overall complexity, even when the time complexity of the individual components is well understood.

Most existing detection mechanisms for algorithmic complexity vulnerabilities use domain- and implementation-specific heuristics or rules, e.g., detect excessive backtracking during regular expression matching [5, 25]. However, such rules tend to be brittle and are hard to scale to a large number of diverse domains, since their

CCS'17, Oct. 30–Nov. 3, 2017, Dallas, TX, USA.

978-1-4503-4946-8/17/10…$15.00
DOI: http://dx.doi.org/10.1145/3133956.3134073

creation and maintenance requires significant manual effort and expertise. Moreover, keeping such rules up-to-date with newer software versions is onerous, as even minor changes to the implementation might require significant changes in the rules.

In this work, we design, implement, and evaluate a novel dynamic domain-independent approach for automatically finding inputs that trigger worst-case algorithmic complexity vulnerabilities in tested applications. In particular, we introduce SlowFuzz, an evolutionary-search-based framework that can automatically find inputs to maximize resource utilization (instruction count, memory usage *etc.*) for a given test binary. SlowFuzz is fully automated and does not require any manual guidance or domain-specific rules. The key idea behind SlowFuzz is that the problem of finding algorithmic complexity vulnerabilities can be posed as an optimization problem whose goal is to find an input that maximizes resource utilization of a target application. We develop an evolutionary search technique specifically designed to find solutions for this optimization problem.

We evaluate SlowFuzz on a variety of real world applications, including the PCRE library for regular expression matching [18], the `bzip2` compression/decompression utility, as well as the hash table implementation of PHP. We demonstrate that SlowFuzz can successfully generate inputs that trigger complexity vulnerabilities in all the above contexts. Particularly, we show that SlowFuzz generates inputs that achieve a 300-times slowdown when decompressed by the `bzip2` utility, can produce regular expressions that exhibit matching times exponential in the input's size, and also manages to automatically generate inputs that trigger a high number of collisions in real-world PHP applications. We also demonstrate that our evolutionary guidance scheme achieves more than 100% improvement over code coverage at steering input generation towards triggering complexity vulnerabilities.

In summary, this work makes the following contributions:

- We present SlowFuzz, the first, to the best of our knowledge, domain-independent dynamic testing tool for automatically finding algorithmic complexity vulnerabilities without any manual guidance.
- We design an evolutionary guidance engine with novel mutation schemes particularly fitted towards generating inputs that trigger worst-case resource usage behaviors in a given application. Our scheme achieves more than 100% improvement over code-coverage-guided input generation at finding such inputs.
- We evaluate SlowFuzz on a variety of complex real-world applications and demonstrate its efficacy at detecting complexity vulnerabilities in diverse domains including large real-world software like the `bzip2` utility and the PCRE regular expression library.

The rest of the paper is organized as follows. We provide a high-level overview of SlowFuzz's inner workings with a motivating example in Section 2. We describe the details of our methodology in Section 3. The implementation of SlowFuzz is described in Section 4 and the evaluation results are presented in Section 5. Section 6 outlines the limitations of our current prototype and discusses possible future extensions. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 OVERVIEW

2.1 Problem Description

In this paper, we detect algorithmic complexity vulnerabilities in a given application by detecting inputs that cause large variations in resource utilization through the number of executed instructions or CPU usage for all inputs of a given size. We assume that our tool has gray-box access to the application binary, i.e., it can instrument the binary in order to harvest different fine-grained resource usage information from multiple runs of the binary, with different inputs. Note that our goal is not to estimate the asymptotic complexities of the underlying algorithms or data structures of the application. Instead, we measure the resource usage variation in some predefined metric like the total edges accessed during a run, and try to maximize that metric. Even though, in most cases, the inputs causing worst-case behaviors under such metrics will be the ones demonstrating the actual worst-case *asymptotic* behaviors, but this may not always be true due to the constant factors ignored in the asymptotic time complexity, the small input sizes, *etc.*

Threat model. Our threat model assumes that an attacker can provide arbitrary specially-crafted inputs to the vulnerable software to trigger worst-case behaviors. This is a very realistic threat-model as most non-trivial real-world software like Web applications and regular expression matchers need to deal with inputs from untrusted sources. For a subset of our experiments involving regular expression matching, we assume that attackers can control regular expressions provided to the matchers. This is a valid assumption for a large set of applications that provide search functionality through custom regular expressions from untrusted users.

2.2 A Motivating Example

In order to understand how our technique works, let us consider quicksort, one of the simplest yet most widely used sorting algorithms. It is well-known [30] that quicksort has an average time complexity of $O(nlogn)$ but a worst-case complexity of $O(n^2)$ where n is the size of the input. However, finding an actual input that demonstrates the worst-case behavior in a particular quicksort implementation depends on low-level details like the pivot selection mechanism. If an adversary knows the actual pivot selection scheme used by the implementation, she can use domain-specific rules to find an input that will trigger the worst-case behavior (e.g., the quadratic time complexity) [40].

However, in our setting, SlowFuzz does not know any domain-specific rules. It also does not understand the semantics of pivot selection or which part of the code implements the pivot selection logic, even though it has access to the quicksort implementation. We would still like SlowFuzz to generate inputs that trigger the corresponding worst-case behavior and identify the algorithmic complexity vulnerability.

This brings us to the following research question: *how can SlowFuzz automatically generate inputs that would trigger worst-case performance in a tested binary in a domain-independent manner?* The search space of all inputs is too large to search exhaustively. Our key intuition in this paper is that evolutionary search techniques can be used to iteratively find inputs that are closer to triggering the worst-case behavior. Adopting an evolutionary testing approach, SlowFuzz begins with a corpus of seed inputs, applies mutations

```
1  function quicksort(array):
2      /* initialize three arrays to hold
3      elements smaller, equal and greater
4      than the pivot */
5      smaller, equal, greater = [], [], []
6      if len(array) <= 1:
7          return
8      pivot = array[0]
9      for x in array:
10         if x > pivot:
11             greater.append(x)
12         else if x == pivot:
13             equal.append(x)
14         else if x < pivot:
15             smaller.append(x)
16     quicksort(greater)
17     quicksort(smaller)
18     array = concat(smaller, equal, greater)
```

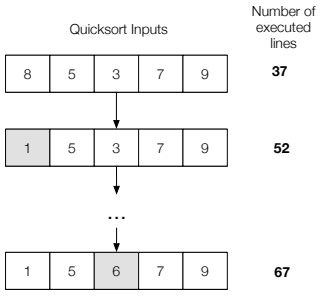

Figure 1: Pseudocode for quicksort with a simple pivot selection mechanism and overview of SlowFuzz's evolutionary search process for finding inputs that demonstrate worst-case quadratic time complexity. The shaded boxes indicate mutated inputs.

to each of the inputs in the corpus, and ranks each of the inputs based on their resource usage patterns. SlowFuzz keeps the highest ranked inputs for further mutations in upcoming generations.

To further illustrate this point, let us consider the pseudocode of Figure 1, depicting a quicksort example with a simple pivot selection scheme—the first element of the array being selected as the pivot. In this case, the worst-case behavior can be elicited by an already sorted array. Let us also assume that SlowFuzz's initial corpus consists of some arrays of numbers and that none of them are completely sorted. Executing this quicksort implementation with the seed arrays will result in a different number of statements/instructions executed based on how close each of these arrays are to being sorted. SlowFuzz will assign a score to each of these inputs based on the number of statements executed by the quicksort implementation for each of the inputs. The inputs resulting in the highest number of executed statements will be selected for further mutation to create the next generation of inputs. Therefore, each upcoming generation will have inputs that are closer to being completely sorted than the inputs of the previous generations.

For example, let us assume the initial corpus for SlowFuzz consists of a single array $\mathcal{I} = [8, 5, 3, 7, 9]$. At each step, SlowFuzz picks at random an input from the corpus, mutates it, and passes the mutated input to the above quicksort implementation while recording the number of executed statements. As shown in Figure 1, the input [8, 5, 3, 7, 9] results in the execution of 37 lines of code (LOC). Let us assume that this input is mutated into [1, 5, 3, 7, 9] that causes the execution of 52 LOC which is higher than the original input and therefore [1, 5, 3, 7, 9] is selected for further mutation. Eventually, SlowFuzz will find a completely sorted array (e.g., [1, 5, 6, 7, 9] as shown in Figure 1) that will demonstrate the worst-case quadratic behavior. We provide a more thorough analysis of SlowFuzz's performance on various sorting implementations in Section 5.2.

3 METHODOLOGY

The key observation for our methodology is that evolutionary search techniques together with dynamic analysis present a promising approach for finding inputs that demonstrate worst-case complexity of a test application in a domain-independent way. However, to enable SlowFuzz to efficiently find such inputs, we need to carefully design effective guidance mechanisms and mutation schemes to drive SlowFuzz's input generation process. We design a new evolutionary algorithm with customized guidance mechanisms and mutation schemes that are tailored for finding inputs causing worst-case behavior.

Algorithm 1 shows the core evolutionary engine of SlowFuzz. Initially, SlowFuzz randomly selects an input to execute from a given seed corpus (line 4), which is mutated (line 5) and passed as input to the test application (line 6). During an execution, profiling info such as the different types of resource usage of the application are recorded (lines 6-8). An input is scored based on its resource usage and is added to the mutation corpus if the input is deemed as a slow unit (lines 9-12).

Algorithm 1 SlowFuzz: Report all slow units for application $\mathcal{A}$ after n generations, starting from a corpus $\mathcal{I}$

```
1:  procedure DiffTest(I, A, n, GlobalState)
2:      units = ∅ ;reported slowunits
3:      while generation ≤ n and I ≠ ∅ do
4:          input = RandomChoice(I)
5:          mut_input = Mutate(input)
6:          app_insn, app_outputs = Run(A, mut_input)
7:          gen_insn ∪ = {app_insn}
8:          gen_usage ∪ = {app_usage}
9:          if SlowUnit(gen_insn, gen_usage,
                        GlobalState) then
10:             I ← I ∪ mut_input
11:             units ∪ = mut_input
12:         end if
13:         generation = generation + 1
14:     end while
15:     return units
16: end procedure
```

In the following Sections, we describe the core components of SlowFuzz's engine, particularly the fitness function used to determine whether an input is a slow unit or not, and the offset and type of mutations performed on each of the individual inputs in the corpus.

3.1 Fitness Functions

As shown in Algorithm 1, SlowFuzz determines, after each execution, whether the executed unit should be considered for further mutations (lines 9-12). SlowFuzz ranks the current inputs based on the scores assigned to them by a fitness function and keeps the fittest ones for further mutation. Popular coverage-based fitness functions which are often used by evolutionary fuzzers to detect crashes, are not well suited for our purpose as they do not consider loop iterations which are crucial for detecting worst-case time complexity.

SlowFuzz's input generation is guided by a fitness function based on resource usage. Such a fitness function is generic and can take into consideration different kinds of resource usage like CPU usage, energy, memory, etc. In order to measure the CPU usage in a fine-grained way, SlowFuzz's fitness function keeps track of the total count of all instructions executed during a run of a test program. The intuition is that the test program becomes slower as the number of executed instructions increases. Therefore, the fitness function selects the inputs that result in the highest number of executed instructions as the slowest units. For efficiency, we monitor execution at the basic-block level instead of instructions while counting the total number of executed instructions for a program. We found that this method is more effective at guiding input generation than directly using the time taken by the test program to run. The runtime of a program shows large variations, depending on the application's concurrency characteristics or other programs that are executing in the same CPU, and therefore is not a reliable indicator for small increases in CPU usage.

3.2 Mutation Strategy

SlowFuzz introduces several new mutation strategies tailored to identify inputs that demonstrate the worst-case complexity of a program. A mutation strategy decides which mutation operations to apply and which byte offsets in an input to modify, to generate a new mutated input (Algorithm 1, line 5).

SlowFuzz supports the following mutation operations: (i) add/remove a new/existing byte from the input; ii) randomly modify a bit/byte in the input; iii) randomly change the order of a subset of the input bytes; iv) randomly change bytes whose values are within the range of ASCII codes for digits (i.e., 0x30-0x39); v) perform a crossover operation in a given buffer mixing different parts of the input; and vi) mutate bytes solely using characters or strings from a user-provided dictionary.

We describe the different mutation strategies supported by SlowFuzz below. Section 5.6 presents a detailed performance comparison of these strategies.

Random Mutations. Random mutations are the simplest mutation strategy supported by SlowFuzz. Under this mutation strategy, one of the aforementioned mutations is selected at random and is applied on an input, as long as it does not violate other constraints for the given testing session, such as exceeding the maximum input length specified by the auditor. This strategy is similar to the ones used by popular evolutionary fuzzers like AFL [58] and libFuzzer [14] for finding crashes or memory safety issues.

Mutation priority. Under this strategy, the mutation operation is selected with ϵ probability based on its success at producing slow units during previous executions. The mutation operation is picked at random with $(1 - \epsilon)$ probability. In contrast, the mutation offset is still selected at random just like the strategy described above.

In particular, during testing, we count all the cases in which a mutation operation resulted in an increase in the observed instruction count and the number of times that operation has been selected. Based on these values, we assign a score to each mutation operation denoting the probability of the mutation to be successful at increasing the instruction count. For example, a score of 0 denotes that the mutation operation has never resulted in an increase in the number of executed instructions, whereas a score of 1 denotes that the mutation always resulted in an increase.

We pick the highest-scoring mutation among all mutation operations with a probability ϵ. The tunable parameter ϵ determines how often a mutation operation will be selected at random versus based on its score. Essentially, different values of ϵ provide different trade-offs between exploration and exploitation. In SlowFuzz, we set the default value of ϵ to 0.5.

Offset priority. This strategy selects the mutation operation to be applied randomly at each step, but the offset to be mutated is selected based on prior history of success at increasing the number of executed instructions. The mutation offset is selected based on the results of previous executions with a probability ϵ and at random with a probability $(1 - \epsilon)$. In the first case, we select the offset that showed the most promise based on previous executions (each offset is given a score ranging from 0 to 1 denoting the percentage of times in which the mutation of that offset led to an increase in the number of instructions).

Hybrid. In this last mode of operation we apply a combination of both mutation and offset priority as described above. For each offset, we maintain an array of probabilities of success for each of the mutation operations that are being performed. Instead of maintaining a coarse-grained success probability for each mutation in the mutation priority strategy, we maintain fine-grained success probabilities for each offset/mutation operation pairs. We compute the score of each offset by computing the average of success probabilities of all mutation operations at that offset. During each mutation, with a probability of ϵ, we pick the offset and operation with the highest scores. The mutation offset and operation are also picked randomly with a probability of $(1 - \epsilon)$.

4 IMPLEMENTATION

The SlowFuzz prototype is built on top of libFuzzer [14], a popular evolutionary fuzzer for finding crash and memory safety bugs. We outline the implementation details of different components of SlowFuzz below. Overall, our modifications to libFuzzer consist of 550 lines of C++ code. We used Clang v4.0 for compiling our modifications along with the rest of libFuzzer code.

Figure 2 shows SlowFuzz's high-level architecture. Similar to the popular evolutionary fuzzers like AFL [58] and libFuzzer [14], SlowFuzz executes in the same address space as the application being tested. We instrument the test application so that SlowFuzz can have access to different resource usage metrics (e.g, number of instructions executed) needed for its analysis. The instrumented test application subsequently is executed under the control of SlowFuzz's analysis engine. SlowFuzz maintains an active corpus of

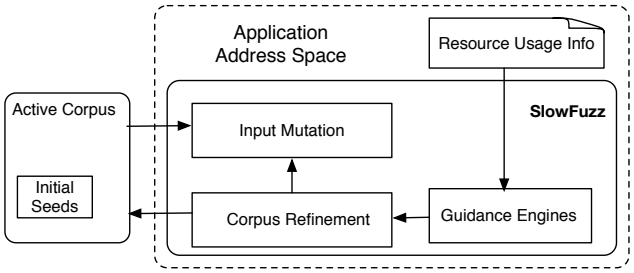

Figure 2: SlowFuzz architecture.

inputs to be passed into the tested applications and refines the corpus during execution based on SlowFuzz's fitness function. For each generation, an input is selected, mutated, then passed into the `main` routine of the application for its execution.

Instrumentation. Similar to libFuzzer, SlowFuzz's instrumentation is based on Clang's SanitizerCoverage [21] passes. Particularly, SanitizerCoverage allows tracking of each executed function, basic block, and edge in the Control Flow Graph (CFG). It also allows us to register callbacks for each of these events. SlowFuzz makes use of SanitizerCoverage's eight bit counter capability that maps each Control Flow Graph (CFG) edge into an eight bit counter representing the number of times that edge was accessed during an execution. We use the counter to keep track of the following ranges: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+. This provides a balance between accuracy of the counts and the overhead incurred for maintaining them. This information is then passed into SlowFuzz's fitness function, which determines whether an input is slow enough to keep for the next generation of mutations.

Mutations. LibFuzzer provides API support for custom input mutations. However, in order to implement the mutation strategies proposed in Section 3.2, we had to modify libFuzzer internals. Particularly, we augment the functions used in libFuzzer's `Mutator` class to return information on the mutation operation, offset, and the range of affected bytes for each new input generated by LibFuzzer. This information is used to compute the scores necessary for supporting mutation piority, offset priority, and hybrid modes as described in Section 3.2 without any additional runtime overhead.

5 EVALUATION

In this Section, we evaluate SlowFuzz on the following objectives: a) Is SlowFuzz capable of generating inputs that match the theoretical worst-case complexity for a given algorithm's implementation? b) Is SlowFuzz capable of efficiently finding inputs that cause performance slowdowns in real-world applications? c) How do the different mutation and guidance engines of SlowFuzz affect its performance? d) How does SlowFuzz compare with code-coverage-guided search at finding inputs demonstrating worst-case application behavior?

We describe the detailed results of our evaluation in the following Sections. All our experiments were performed on a machine with 23GB of RAM, equipped with an Intel(R) Xeon(R) CPU X5550 @ 2.67GHz and running 64-bit Debian 8 (jessie), compiled with GCC version 4.9.2, with a kernel version 4.5.0. All binaries were compiled using the Clang-4.0 compiler toolchain. All instruction counts and execution times are measured using the Linux `perf` profiler v3.16, averaging over 10 repetitions for each `perf` execution.

5.1 Overview

In order to adequately address the questions outlined in the previous Section, we execute SlowFuzz on applications of different algorithmic profiles and evaluate its ability of generating inputs that demonstrate worst case behavior.

First, we examine if SlowFuzz generates inputs that demonstrate the theoretical worst-case behavior of well-known algorithms. We apply SlowFuzz on sorting algorithms with well-known complexities. The results are presented in Section 5.2. Subsequently, we apply SlowFuzz on different applications and algorithms that have been known to be vulnerable to complexity attacks: the PCRE regular expression library, the default hash table implementation of PHP, and the `bzip2` binary. In all cases, we demonstrate that SlowFuzz is able to trigger complexity vulnerabilities. Table 1 shows a summary of our findings.

Tested Application	Fuzzing Outcome
Insertion sort [30]	41.59x slowdown
Quicksort (Fig 1)	5.12x slowdown
Apple quicksort	3.34x slowdown
OpenBSD quicksort	3.30x slowdown
NetBSD quicksort	8.7% slowdown
GNU quicksort	26.36% slowdown
PCRE (fixed input)	78 exponential & 765 superlinear regexes
PCRE (fixed regex)	8% - 25% slowdown
PHP hashtable	20 collisions in 64 keys
`bzip2` decompression	~300x slowdown

Table 1: Result Summary

As shown in Table 1, SlowFuzz is successful at inducing significant slowdown on all tested applications. Moreover, when applied to the PCRE library, it managed to generate regular expressions that exhibit exponential and super-linear (worse than quadratic) matching automatically, without any knowledge of the structure of a regular expression. Likewise, it successfully generated inputs that induce a high number of collisions when inserted into a PHP hash table, without any notion of hash functions. In the following Sections, we provide details on each of the above test settings.

5.2 Sorting

Simple quicksort and insertion sort. Our first evaluation of SlowFuzz's consistency with theoretical results is performed on common sorting algorithms with well-known worst-performing inputs. To this end, we initially apply SlowFuzz on an implementation of the insertion sort algorithm [30], as well as on an implementation of quicksort [30] in which the first sub-array element is always selected as the pivot. Both of the above implementations demonstrate quadratic complexity when the input passed to them is sorted. We run SlowFuzz for 1 million generations on the above

implementations, sorting a file with a size of 64 bytes, and examine the slowdown SlowFuzz introduced over the fastest unit seen during testing. To do so, we count the total instructions executed by each program for each of the inputs, subtracting all instructions not relevant to the quicksort functionality (e.g., loader code). Our results are presented in Figure 3.

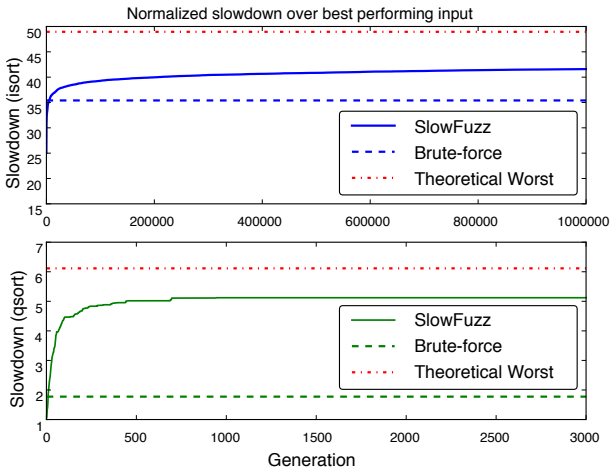

Figure 3: Best slowdown achieved by SlowFuzz at each generation (normalized over the slowdown of the best-performing input) versus best random testing outcome, on our insertion sort and quicksort drivers, for an input size of 64 bytes (average of 100 runs). The SlowFuzz achieves slowdowns of 84.97% and 83.74% compared to the theoretical worst cases for insertion sort and quicksort respectively.

Figure 3 represents an average of 100 runs. In each run, SlowFuzz started execution with a single random 64 byte seed, and executed for 1 million generations. We notice that SlowFuzz achieves 41.59x and 5.12x slowdowns for insertion sort and quicksort respectively. In order to examine how this behavior compares to random testing, we randomly generated 1 million inputs of 64 bytes each and measured the instructions required for insertion sort and quicksort, respectively. Figure 3 depicts the *maximum* slowdown achieved through random testing *across all* runs. We notice that in both cases SlowFuzz outperforms the brute-force worst-input estimation. Finally, we observe that the gap between brute-force search and SlowFuzz is much higher for quicksort than insertion, which is consistent with the fact that average case complexity of insertion sort is $O(n^2)$, compared to quicksort's $O(nlogn)$. Therefore, a random input is more likely to demonstrate worst-case behavior for insertion sort but not for quicksort.

Real-world quicksort implementations. We also examined how SlowFuzz performs when applied to real-world quicksort implementations. Particularly, we applied it to the Apple [12], GNU [9], NetBSD [15], and OpenBSD [13] quicksort implementations. We notice that SlowFuzz's performance on real world implementations is consistent with the quicksort performance that we observed in the experiments described above. In particular, the slowdowns generated by SlowFuzz were (in increasing order) 8.7%, for theNetBSD implementation, 26.36% for the GNU quicksort implementation, 3.30x for the OpenBSD implementation and 3.34x for the Apple implementation. We notice that, despite the fact these implementations use efficient pivot selection strategies, SlowFuzz still manages to trigger significant slowdowns. On the contrary, repeating the same experiment using naive coverage-based fuzzing yields slowdowns that never surpass 5% for any of the libraries. This is an expected result, as coverage-based fuzzers are geared towards maximizing coverage, and thus do not favor inputs exercising the same edges repeatedly over inputs that discover new edges.

Finally, we note that, similar to the experiment of Figure 3, the slowdowns for Figure 4 are also measured in terms of executed instructions, normalized over the instructions of the best performing input seen during testing.

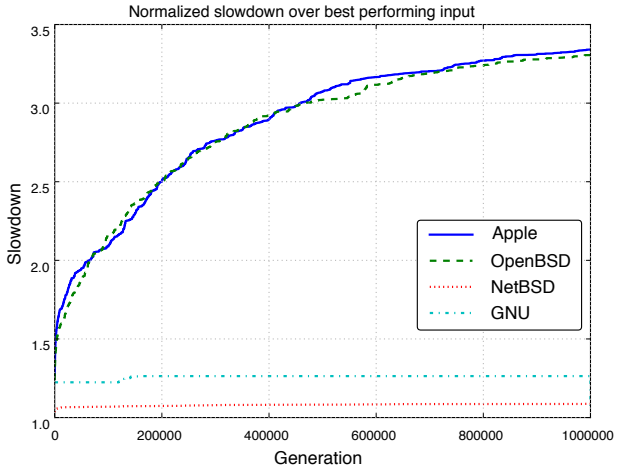

Figure 4: Best slowdown (with respect to the best-performing input) achieved by SlowFuzz at each generation normalized over the best random testing outcome, on real-world quicksort implementations, for an input size of 64 bytes (average of 100 runs).

> **Result 1**: SlowFuzz was able to generate inputs for quicksort and insertion sort that achieve 83.74% and 84.97% of the theoretical worst-case, respectively without any information on the algorithm internals.

5.3 Regular Expressions

Regular expression implementations are known to be susceptible to complexity attacks [17, 20, 24]. In particular, there are over 150 Regular expression Denial of Service (ReDoS) vulnerabilities registered in the National Vulnerability Database (NVD), which are the result of exponential (e.g., [8]) or super-linear (worse than quadratic) e.g., [7] complexity of regular expression matching by several existing matchers [57].

Even performing domain-specific analyses of whether an application is susceptible to ReDoS attacks is non-trivial. Several works are solely dedicated to the detection of exploitation of such vulnerabilities. Recently, Rexploiter [57] presented algorithms to detect

whether a given regular expression may result in non-deterministic finite automata (NFA) that require super-linear or exponential matching times for specially crafted inputs. They have also presented domain-specific algorithms to generate inputs capable of triggering such worst-case performance. The above denote the hardness of SlowFuzz's task, namely finding regular expressions that may result in super-linear or exponential matching times without any domain knowledge.

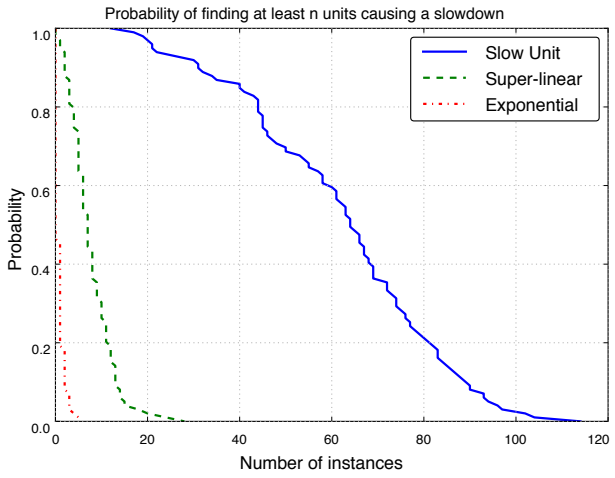

Figure 5: Probability of SlowFuzz finding at least n unique instances of regexes that cause a slowdown, or exhibit super-linear and exponential matching times, after 1 million generations (inverse CDF over 100 runs).

For the regular expression setting we perform two separate experiments to check whether SlowFuzz can produce i) regular expressions which exhibit super-linear and exponential matching times, ii) inputs that cause slowdown during matching, given a fixed regular expression. To this end, we apply SlowFuzz on the PCRE regular expression library [18] and provide it with a character set of the symbols used in PCRE-compliant regular expressions (in the form of a dictionary). Notice that we do not further guide SlowFuzz with respect to what mutations should be done and SlowFuzz's engine is completely agnostic of the structure of a valid regular expression. In all cases, we start testing from an empty corpus without providing any seeds of regular expressions to SlowFuzz.

Fixed string and mutated regular expressions. For the first part of our evaluation, we apply SlowFuzz on a binary that utilizes the PCRE library to perform regular expression matching and we let SlowFuzz mutate the regular expression part of the `pcre2_match` call used for the matching, using a dictionary of regular expression characters. The input to be matched against the regular expression is selected from a random pool of characters and SlowFuzz executes for a total of 1 million generations, or until a time-out is hit. The regular expressions generated by SlowFuzz are kept limited to 10 characters or less. Once a SlowFuzz session ends, we evaluate the time complexity of the generated regular expressions utilizing `Rexploiter` [57], which detects if the regular expression is super-linear, exponential, or none of the two. We repeat the above process for a total of 100 fuzzing sessions.

Overall, SlowFuzz generates a total of 33343 regular expressions during the above 100 sessions, out of which 27142 are rejected as invalid whereas 6201 are valid regular expressions that caused a slowdown. Out of the valid regular expressions, 765 are superlinear and 78 are exponential. This experiment demonstrates that despite being agnostic of the semantics of regex matching, SlowFuzz successfully generates regexes requiring super-linear and exponential matching times. Six such examples are presented in Table 2.

Super-linear (greater than quadratic)	Exponential
c*ca*b*a*b	(b+)+c
a+b+b+b+a+	c*(b+b)+c
c*c+ccbc+	a(a|a*)+a

Table 2: Sample regexes generated by SlowFuzz resulting in super-linear (greater than quadratic) and exponential matching complexity.

A detailed case study. The regexes presented in Table 2 are typical examples of regular expressions that require non-linear matching running times. This happens due to the existence of different paths in the respective NFAs, which reach the same state through an identical sequence of labels. Such paths have a devastating effect during backtracking [57]. To further elaborate on this property, let us consider the NFA depicted in Figure 6, which corresponds to the regular expression (b+)+c of Table 2.

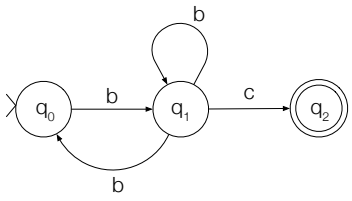

Figure 6: NFA for the regular expression (b+)+c suffering from exponential matching complexity as found by SlowFuzz. q_0 is the entry state, q_2 the accept state, and q_1 the pivot state for the exponential backtracking.

We notice that, for the NFA shown in Figure 6, starting from state q_1, it is possible to reach q_1 again, through two different paths, namely the paths ($q_1 \xrightarrow{b} q_0, q_0 \xrightarrow{b} q_1$) and ($q_1 \xrightarrow{b} q_1, q_1 \xrightarrow{b} q_1$). Moreover, we notice that the labels in the transitions for both of the above paths are the same: 'bb' is consumed in both cases. Thus, as it is possible to reach q_2 from q_1 (via label c) as well as reach q_1 from the initial state q_0, there will be an exponentially large number of paths to consider in the case of backtracking. Similar issues arise with loops appearing in NFAs with super-linear matching [57].

As mentioned above, on average, among the valid regular expressions generated by SlowFuzz, approximately 12.33% of the regexes have super-linear matching complexity, whereas 2.29% on average have exponential matching complexity. The aforementioned results are aggregates across all the 100 executions of the experiment. In order to estimate the probability of SlowFuzz to generate a regex

that exhibits a slowdown [1], or super-linear and exponential matching times in a *single* session, we calculate the respective inverse CDF which is shown in Figure 5. We notice that, for all the regular expressions observed, SlowFuzz successfully generates inputs that incur a slowdown during matching. In particular, with 90% probability, SlowFuzz generates at least 2 regular expressions requiring super-linear matching time and at least 31 regular expressions that cause a slowdown. SlowFuzz generates at least one regex requiring exponential matching time with a probability of 45.45% .

Fixed regular expression and mutated string. In the second part of our evaluation of SlowFuzz on regular expressions, we seek to examine if, for a given *fixed* regular expression, SlowFuzz is able to generate inputs that can introduce a slowdown during matching. We collect PCRE-compliant regular expressions from popular Web Application Firewalls (WAF) [2], and utilized the PCRE library to match input strings generated by SlowFuzz against each regular expression. For this experiment, we apply SlowFuzz on a total of 25 regular expressions, and we record the total instructions executed by the PCRE library when matching the regular expression against SlowFuzz's generated units, at each generation. For our set of regular expressions, SlowFuzz achieved monotonically increasing slowdowns, ranging from 8% to 25%. Figure 7 presents how the slowdown varies as fuzzing progresses, for three representative regex samples with different slowdown patterns.

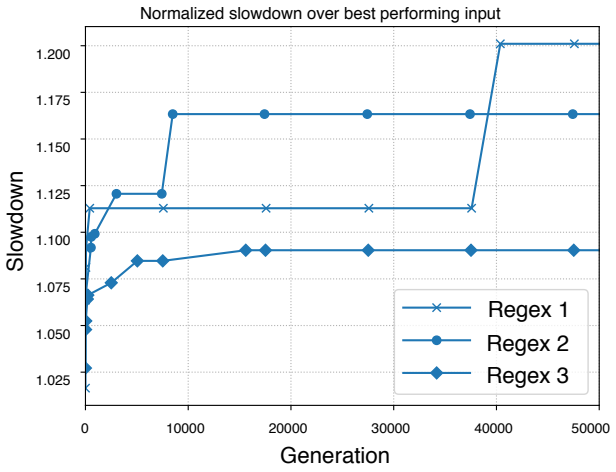

Figure 7: Best slowdown achieved by SlowFuzz-generated input strings (normalized over the slowdown of the best-performing input), when matching against fixed regular expressions used in WAFs (normalized against best performing input over an average of 100 runs). The corresponding regexes are listed in Appendix A.

5.4 Hash Tables

Hash tables are a core data structure in a wide variety of software. The performance of hash table lookup and insertion operations significantly affects the overall application performance. Complexity attacks against hash table implementations may induce unwanted effects ranging from performance slowdowns to full-fledged DoS [8, 17, 19, 20, 24]. In order to evaluate if SlowFuzz can generate inputs that trigger collisions without knowing any details about the underlying hash functions, we apply it on the hash table implementation of PHP (v5.6.26), which is known to be vulnerable to collision attacks.

PHP Hashtables. Hashtables are prevalent in PHP and they also serve as the backbone for PHP's array interface. PHP v5.x utilizes the DJBX33A hash function for hashing using string keys, which can bee seen in Listing 1.

We notice that for two strings of the form 'ab' and 'cd' to collide, the following property must hold [10]:

$$c = a + n \wedge d = b - 33 * n, n \in \mathbb{Z}$$

It is also easy to show that if two equal-length strings A and B collide, then strings xAy, xBy where x and y are any prefix and suffix respectively, also collide. Using the above property, one can construct a worst-case performing sequence of inputs [3], forcing a worst-case insertion time of $O(n^2)$.

```
/*
 * @arKey is the array key to be hashed
 * @nKeyLenth is the length of arKey
 */
static inline ulong
zend_inline_hash_func(const char *arKey, uint
     nKeyLength)
{
    register ulong hash = 5381;

    for (uint i = 0; i < nKeyLength; ++i) {
        hash = ((hash << 5) + hash) + arKey[i];
    }

    return hash;
}
```

Listing 1: DJBX33A hash without loop unrolling.

Abusing the complexity characteristics of the BJBX33A hash, attackers performed DoS attacks against PHP, Python and Ruby applications in 2011. As a response, PHP added an option in its `ini` configuration to set a limit on the number of collisions that are allowed to happen. However, in 2015, similar DoS attacks [1] were reported, abusing PHP's JSON parsing into hash tables. In this experiment we examine how SlowFuzz performs when applied to this particular hash function implementation.

Our experimental setup is as follows: we ported the PHP hash table implementation so that the latter can be used in any C/C++ implementation, removing all the interpreter-specific variables and macros, however leaving all the non interpreter-related components intact. Subsequently, we created a hash table with a size of 64 entries, and utilized SlowFuzz to perform *a maximum* of 64 insertions to the hash table, using strings as keys, starting from a corpus consisting of a single input that causes 8 collisions. In particular, the keys for the hash table insertions were provided by SlowFuzz at each generation and SlowFuzz evolved its corpus of strings using a hybrid mutation strategy. Given a hash table of 64 entries and 64 insertions to the hash table, the maximum number of collisions that can be performed is also 64. In order to

[1]Notice that due to SlowFuzz's guidance engine, any regex produced must exhibit increased instruction count as compared to *all* previous regexes.

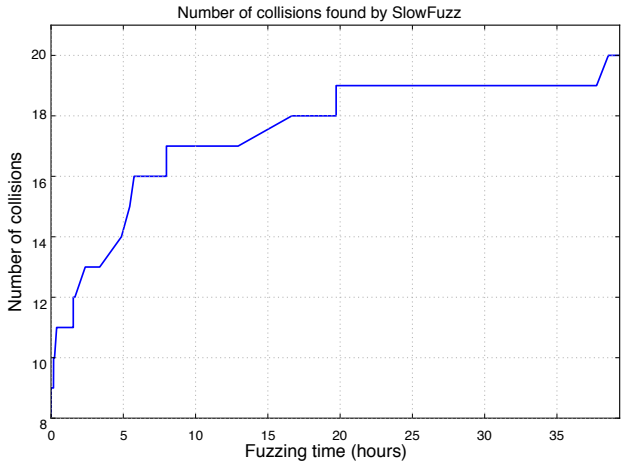

Figure 8: Number of collisions found by SlowFuzz per generation, when applying it on the PHP 5.6 hashtable impelemntation, for at most of 64 insertions with string keys.

measure the number of collisions occurring in the hashtable at each generation, we created a PHP module (running in the context of PHP), and measured the number of collisions induced by each input that SlowFuzz generates. We perform our measurements after the respective elements are inserted into a *real* PHP array. Our results are presented in Figure 8.

We notice that despite the complex conditions required to trigger a hash collision and without knowing any details about the hash function, SlowFuzz's evolutionary engine reaches 31.25% of the theoretical worst-case after approximately 40 hours of fuzzing, using a single CPU. SlowFuzz's stateful, evolutionary guidance achieves monotonically increasing slowdowns, despite the complex constraints imposed by the hash function. On the contrary, repeating the same experiment using coverage-based fuzzing, yielded non-monotonically increasing collisions, and at no point an input was generated causing more than 8 collisions. In particular, fuzzing using coverage generated 58 inputs with a median of 5 collisions.

5.5 ZIP Utilities

Zip utilities that support various compression/decompression schemes are another instance of applications that have been shown to suffer from Denial of Service attacks. For instance, an algorithmic complexity vulnerability used in the sorting algorithms in the `bzip2` application [2] allowed remote attackers to cause DoS via increased CPU consumption, when they provided a file with many repeating inputs [16].

In order to evaluate how SlowFuzz performs when applied to the compression/decompression libraries, we apply it on `bzip2` v1.0.6. In particular, we utilize SlowFuzz to create compressed files of a maximum of 250 bytes, and we subsequently use the libbzip2 library to decompress them. Based on the slowdowns observed during decompression, SlowFuzz evolves its input corpus, mutating each input using its hybrid mode of operation. Our experimental results are presented in Figure 9.

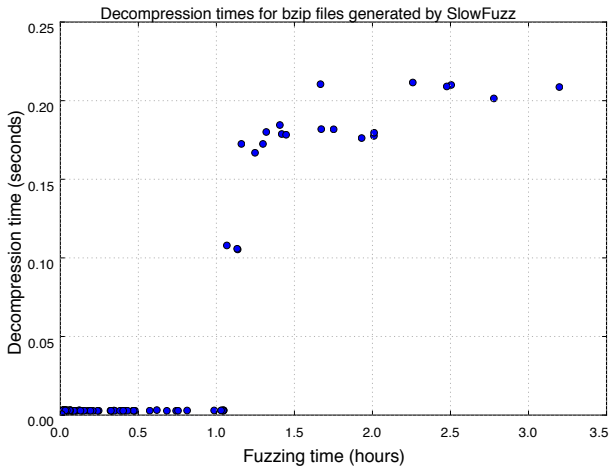

Figure 9: Slowdowns observed while decompressing inputs generated by SlowFuzz using the `bzip2` binary. The maximum file size is set to 250 bytes.

A detailed case study. Figure 9 depicts the time required by the `bzip2` binary to decompress each of the inputs generated by SlowFuzz. We notice that for the first hour of fuzzing, the inputs generated by SlowFuzz do not exhibit significant slowdown during their decompression by `bzip2`. In particular, each of the 250-byte inputs of SlowFuzz's corpus for the first hour of fuzzing is decompressed in approximately 0.0006 seconds. However, in upcoming generations, we observe that SlowFuzz successfully achieves decompression times reaching 0.18s to 0.21s and an overall slowdown in the range of 300x. Particularly, in the first 6 minutes after the first hour, SlowFuzz achieves a decompression time of 0.10 sec. This first peak in the decompression time is achieved because of SlowFuzz triggering the randomization mechanism of `bzip2`, by setting the respective header byte to a non-zero value. This mechanism, although deprecated, was put in place to protect against repetitive blocks, and is still supported for legacy reasons. However, even greater slowdowns are achieved when SlowFuzz mutates two bytes used in `bzip2`'s Move to Front Transform (MTF) [4] and particularly in the run length encoding of the MTF result. Specifically, the mutation of these bytes affects the total number of invocations of the `BZ2_bzDecompress` routine, which results in a total slowdown of 38.31x in decompression time.

The respective code snippet in which the affected bytes are read is shown in Listing 2: the `GET_MTF_VAL` macro reads the modified bytes in memory [3]. These bytes subsequently cause the routine `BZ2_bzDecompress` to be called 4845 times, contrary to a single call before the mutation. We should note at this point, that the total size of the input before and after the mutation remained unchanged.

Finally, in order to compare with a non complexity-targeting strategy, we repeated the previous experiment using traditional coverage-based fuzzing. The fuzzer, when guided only based on

[2] The vulnerability is found in `BZip2CompressorOutputStream` for Apache Commons Compress before 1.4.1

[3] Via the macros `GET_BITS(BZ_X_MTF_3, zvec, zn)` and `GET_BIT(BZ_X_MTF_4, zj)`

coverage, did not generate any input causing executions larger than 0.0008 seconds, with the maximum slowdown achieved being 23.7%.

```
do {
  /* Check that N doesn't get too big, so that
   es doesn't go negative.  The maximum value
   that can be RUNA/RUNB encoded is equal
   to the block size (post the initial RLE),
   viz, 900k, so bounding N at 2 million
   should guard against overflow without
   rejecting any legitimate inputs. */
  if (N >= 2*1024*1024) RETURN(BZ_DATA_ERROR);
  if (nextSym == BZ_RUNA) es = es + (0+1) * N; else
  if (nextSym == BZ_RUNB) es = es + (1+1) * N;
  N = N * 2;
  GET_MTF_VAL(BZ_X_MTF_3, BZ_X_MTF_4, nextSym);
}
  while (nextSym == BZ_RUNA || nextSym == BZ_RUNB);
```

Listing 2: Excerpt from bzip2's BZ2_decompress routine (decompress.c). A two byte modification by SlowFuzz results in a 38.31x slowdown compared to the previous input.

From the above experiment we observe that SlowFuzz's guidance and mutations engines are successful in pinpointing locations that trigger large slowdowns even in very complex applications such as a state-of-the-art compression utility like `bzip2`.

Result 2: SlowFuzz is capable of exposing complexity vulnerabilities (e.g., 300x slowdown in `bzip2`, PCRE-compliant regular expressions with exponential matching time, and PHP hash table collisions) in real-world, non-trivial applications without knowing any domain-specific details.

5.6 Engine Evaluation

Effect of SlowFuzz's fitness function. In this section, we examine the effect of using code-coverage-guided search versus SlowFuzz's resource usage based fitness function, particularly in the context of scanning an application for complexity vulnerabilities. To do so, we repeat one of the experiments of Section 5.2, applying SlowFuzz on the OpenBSD quicksort implementation with an input size of 64 bytes, for a total of 1 million generations, using hybrid mutations. Our results are presented in Figure 10. We observe that SlowFuzz's guidance mechanism yields significant improvement over code-coverage-guided search. In particular, SlowFuzz achieves a 3.3x slowdown for OpenBSD, whereas the respective slowdown achieved using only coverage-guided search is 23.41%. This is an expected result, since, as mentioned in previous Sections, code coverage cannot encapsulate behaviors resulting in multiple invocations of the same line of code (e.g., an infinite loop). Moreover, we notice that the total instructions of each unit that is created by SlowFuzz at different generations is not monotonically increasing. This is an artifact of our implementation, using SanitizerCoverage's 8-bit counters, which provide a coarse-grained, imprecise tracking of the real number of times each edge was invoked (Section 4). Thus, although a unit might result in execution of fewer instructions, it will only be observed by SlowFuzz's guidance engine if the respective number of total CFG edges falls into a separate bucket (8 possible ranges representing the total number of CFG edge accesses). Future work can consider applying more precise instruction tracking (e.g., using hardware counters or utilities similar to `perf`) with static analyses passes, to achieve more effective guidance.

Finally, when choosing the SlowFuzz fitness function, we also considered the option of utilizing time-based tracking instead of performance counters. However, performing time-based measurements in real-world systems is not trivial, especially at instruction-level granularity and when multiple samples are required in order to minimize measurement errors. In the context of fuzzing, multiple runs of the same input will slow the fuzzer down significantly. To demonstrate this point, in Figure 10, we also include an experiment in which the execution time of an input is used to guide input generation. In particular, we utilized CPU clock time to measure the execution time of a unit and discarded the unit if it was not slower than all previously seen units. We notice that the corpus degrades due to system noise and does not achieve any slowdown larger than 23%. [4]

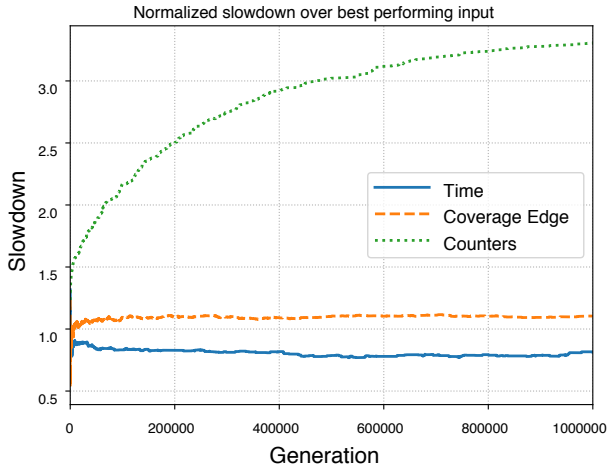

Figure 10: Comparison of the slowdown achieved by SlowFuzz under different guidance mechanisms, when applied on the OpenBSD quicksort implementation of Section 5.2, for an input size of 64 bytes, after 1 million generations (average of 100 runs).

Result 3: SlowFuzz's fitness function and mutation schemes outperform code-coverage-guided evolutionary search by more than 100%.

Effect of Mutation Schemes. To highlight the different characteristics of each of SlowFuzz's mutation schemes described in Section 3, we repeat one of the experiments of Section 5.2, applying SlowFuzz on the OpenBSD quicksort, each time using a different mutation strategy. Our experimental setup is identical with that of Section 5.2: we sort inputs with a size of 64 bytes and fuzz for a total

[4] Contrary to the slowdowns measured during fuzzing using a single run, the slowdowns presented in Figure 10 are generated using the `perf` utility running ten iterations per input. Non-monotonic increases denote corpus degradation due to bad input selection.

of 1 million generations. For each mode of operation, we average on a total of 100 SlowFuzz sessions. Our results are presented in Figure 11.

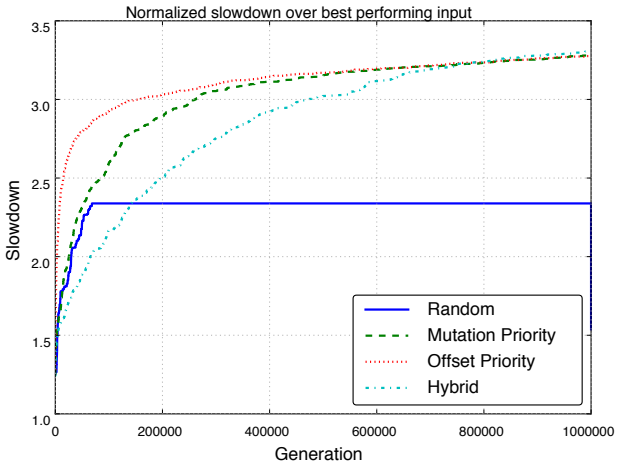

Figure 11: Comparison of the best slowdown achieved by SlowFuzz's different mutation schemes, at each generation, when applied on the OpenBSD quicksort implementation of Section 5.2, for an input size of 64 bytes, after 1 million generations (average of 100 runs).

We notice that, for the above experiment, choosing a mutation at random, is the worst performing option among all mutation options supported by SlowFuzz (Section 3.2), however still achieving a slowdown of 2.33x over the best performing input. Indeed, all of SlowFuzz's scoring-based mutation engines (offset-priority, mutation-priority and hybrid), are expected to perform at least as good as selecting mutations at random, given enough generations, as they avoid getting stuck with unproductive mutations. We also observe that offset priority is the fastest mode to converge out of the other mutation schemes for this particular experiment, and results in an overall slowdown of 3.27x.

For sorting, offsets that correspond to areas of the array that should *not* be mutated, are quickly penalized under the offset priority scheme, thus mutations are mainly performed on the non-sorted portions of the array. Additionally, we observe that mutation priority also outperforms the random scheme due to the fact that certain mutations (e.g., crossover operations) may have devastating effects on the sorting of the array. The mutation priority scheme picks up such patterns and avoids such mutations. By contrast, these mutations continue to be used under the random scheme. Finally, we observe that the hybrid mode eventually outperforms all other strategies, achieving a 3.30x slowdown, however is the last mutation mode to start reaching a plateau. We suspect that this results from the fact the hybrid mode does not quickly penalize particular inputs or mutations as it needs more samples of each mutation operation and offset pair before avoiding any particular offset or mutation operation.

Instrumentation overhead. SlowFuzz's runtime overhead, measured in executions per second, matches the overhead of native libFuzzer. The executions per second achieved on different payloads are mostly dominated by the runtimes of the native binary, as well as the respective I/O operations. Despite our choice to prototype SlowFuzz using libFuzzer, the design and methodology presented in Section 3 can be applied to any evolutionary fuzzer and can also be implemented using Dynamic Binary Instrumentation frameworks, such as Intel's PIN [39], to allow for more detailed runtime tracking of the application state. However, such frameworks are known to incur slowdowns of more than 200%, even with minimal instrumentation [43]. For instance, for our PHP hashtable experiments described in Section 5.4, an insertion of 16 strings, resulting in 8 collisions, takes 0.02 seconds. Running the same insertion under a PIN tool that only counts instructions, requires a total of ~2 seconds. By contrast, hashtable fuzzing with SlowFuzz achieves up to 4000 execs/sec, unless a significant slowdown is incurred due to a particular input. [5]

6 DISCUSSION

In this paper, we demonstrated that evolutionary search techniques commonly used in fuzzing to find memory safety bugs can be adapted to find algorithmic complexity vulnerabilities. Similar strategies should be applicable for finding other types of DOS attacks like battery draining, filling up memory or hard disk, etc. Designing the fitness functions and mutation schemes for detecting such bugs will be an interesting future research problem. Besides evolutionary techniques, using other mechanisms like reinforcement learning or Monte Carlo search techniques can also be adapted for finding inputs with worst-case resource usage.

Our current prototype of SlowFuzz is completely dynamic. However, integrating static analysis techniques into SlowFuzz can further improve its performance. Using static analysis to find potentially promising offsets in an input for mutation will further reduce the search space and therefore will make the search process more efficient. For example, using taint tracking and loop analysis together with runtime flow profiles can identify potentially promising code locations that can cause significant slowdowns [41, 52].

The current prototype implementation of SlowFuzz uses the SanitizerCoverage passes to keep track of the number of times a CFG edge is accessed. Such tracking is limited by the total number of buckets allowed by SanitizerCoverage. This reduces the accuracy of resource usage information as tracked by SlowFuzz. This results from the fact that any edge that is accessed more than 128 times is assigned to the same bucket regardless of the actual number of accesses. Although, under its current implementation, the actual edge count information is imprecise, this is not a fundamental design limitation of SlowFuzz but an artifact of our prototype implementation. Alternative implementations can offer more precise tracking can via custom callbacks for SanitizerCoverage, by adopting hardware counters or by utilizing per-unit `perf` tracking. On the other hand, the benefit of the current implementation is that it can be incorporated into libFuzzer's main engine orthogonally, without requiring major changes to libFuzzer's dependencies.

[5] Execution under SlowFuzz does not require repeated loading of the required libraries, but is only dominated by the function being tested, which is only a fraction of the total execution of the native binary (thus smaller than 0.02 seconds).

7 RELATED WORK

Complexity attacks. Detecting and mitigating algorithmic complexity attacks is an active field of research. Crosby et al. [31] were the first to present complexity attacks abusing collisions in hash table implementations. Contrary to SlowFuzz's approach, however, their attack required expert knowledge. Since then, several lines of work have explored attacks and defenses targeting different types of complexity attacks: Cai et al. [28] leverage complexity vulnerabilities in the Linux kernel name lookup hash tables to exploit race conditions in the kernel `access(2)/open(2)` system calls, whereas Sun et al. [54] explore complexity vulnerabilities in the name lookup algorithm of the Linux kernel to achieve an exploitable covert timing channel. Smith et al. [51] exploit the syntax of the Snort IDS to perform a complexity attack resulting in slowdowns during packet inspection. Shenoy et al. [49, 50] present an algorithmic complexity attack against the popular Aho-Corasick string searching algorithm and propose hardware and software-based defenses to mitigate the worst-case performance of their attacks. Moreover, several lines of work focus particularly on statically detecting complexity vulnerabilities related to regular expression matching, especially focusing on backtracking during the matching process [25, 38, 42, 57]. Contrary to SlowFuzz, all the above lines of work require deep domain-dependent knowledge and do not expand to different categories of complexity vulnerabilities.

Finally, recent work by Holland et al. [34] combines static and dynamic analysis to perform analyst-driven exploration of Java programs to detect complexity vulnerabilities. However, contrary to SlowFuzz, this work requires a human analyst to closely guide the exploration process, specifying which portions of the binary should be analyzed statically and which dynamically as well as defining the inputs to the binary.

Performance bugs. Several prior works target generic performance bugs not necessarily related to complexity vulnerabilities. For instance, Lu et al. study a large set of real-world performance bugs to construct a set of rules that they use to discover new performance bugs via hand-built checkers integrated in the LLVM compiler infrastructure [36]. Along the same lines, LDoctor [52] detects loop inefficiencies by implementing a hybrid static-dynamic program analysis that leverages different loop-specific rules. Both the above lines of work, contrary to SlowFuzz, require expert-level knowledge for creating the detection rules, and are orthogonal to the current work. Another line of work focuses on application profiling to detect performance bottlenecks. For example, Ramanathanet al. utilize flow profiling for the efficient detection of memory-related performance bugs in Java programs [41]. Grechanik et al. utilize a genetic-algorithm-driven profiler for detecting performance bottlenecks [48] in Web applications, and cluster execution traces to explore different combinations of the input parameter values. However, contrary to SlowFuzz, their goal is to explore a large space of input combinations in the context of automatic application profiling and not to detect complexity vulnerabilities.

WCET. Another related line of work addresses accurate Worst-Case Execution Time (WCET) estimation for a given application. Apart from static analysis and evolutionary testing approaches [26], traditionally WCET estimation has been achieved using search based methods measuring end-to-end execution times [55]. Moreover, Hybrid Measurement-Based Analyses (HMBA) have been used to measure the execution times of program segments via instrumentation points [27, 45, 46] and execution profiles [26]. Wegener et al. [56] utilize evolutionary techniques for testing timing constraints in real-time systems, however contrary to SlowFuzz, apply processor-level timing measurements for their fitness function and only perform random mutations. Finally, recent techniques combine hardware effects and loop bounds with genetic algorithms [37]. However, all of the above methods attempt to detect worst-case execution times for simple and mostly straight-line program segments often used in real-time systems. By contrast, SlowFuzz detects algorithmic complexity attacks in large complex programs deployed in general purpose hardware.

Evolutionary Fuzzing. Several lines of work deploy evolutionary mutation-based fuzzing to target crash-inducing bugs. Notable examples are the AFL [58], libFuzzer [14], hongfuzz [11], and syzkaller [23] fuzzers, as well as the CERT Basic Fuzzing Framework (BFF) [35], which utilize coverage as their main guidance mechanism. Moreover, several frameworks combine coverage-based evolutionary fuzzing with symbolic execution [29, 32, 33, 53], or with static analysis and dynamic tainting [47] to achieve higher code coverage and increase their effectiveness in detecting bugs. Finally, NEZHA [44] utilizes evolutionary-based, mutation-assisted testing to target semantic bugs. Although many of the aforementioned lines of research share many common building blocks with SlowFuzz, they do not target complexity vulnerabilities and mainly utilize random mutations contrary to SlowFuzz's targeted mutation strategies.

8 CONCLUSION

In this work we designed SlowFuzz, the first, to the best of our knowledge, evolutionary-search-based framework targeting algorithmic complexity vulnerabilities. We evaluated SlowFuzz on a variety of real-world applications including zip utilities, regular expression libraries and hash table implementations. We demonstrated that SlowFuzz can successfully generate inputs that match the theoretical worst-case complexity in known algorithms. We also showed that SlowFuzz was successful in triggering complexity vulnerabilities in all the applications we examined. SlowFuzz's evolutionary engine and mutation strategies generated inputs causing more than 300-times slowdown in the `bzip2` decompression routine, produced inputs triggering high numbers of collisions in production-level hash table implementations, and also generated regular expressions with exponential matching complexities without any knowledge about the semantics of regular expressions. We believe our results demonstrate that customized evolutionary search techniques present a promising direction for automated detection of not only algorithmic complexity vulnerabilities, but also of other types of resource exhaustion vulnerabilities, and hope to aspire tighter integration of existing techniques and static analyses with modern mutation-based evolutionary testing.

9 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This work is sponsored in part by the Office of Naval

Research (ONR) grant N00014-17-1-2010, the National Science Foundation (NSF) grants CNS-13-18415 and CNS-16-17670, and a Google Faculty Fellowship. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, ONR, NSF, or Google.

REFERENCES

[1] #800564 - PHP5: trivial hash complexity DoS attack - Debian Bug report logs. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=800564.

[2] attackercan/regexp-security-cheatsheet. https://github.com/attackercan/regexp-security-cheatsheet/tree/master/RegexpSecurityParser/WAF-regexps.

[3] bk2204/php-hash-dos: A PoC hash complexity DoS against PHP. https://github.com/bk2204/php-hash-dos.

[4] bzip2. http://www.bzip.org/1.0.3/html/index.html.

[5] Controlling backtracking. https://msdn.microsoft.com/en-us/library/dsy130b4(v=vs.110).aspx#controlling_backtracking.

[6] CVE-2011-5021. http://cve.mitre.org/cgi-bin/cvename.cgi?name={CVE}-2011-5021.

[7] CVE-2013-2099. http://cve.mitre.org/cgi-bin/cvename.cgi?name={CVE}-2013-2099.

[8] CVE-2015-2526. http://cve.mitre.org/cgi-bin/cvename.cgi?name={CVE}-2015-2526.

[9] gnulib/qsort.c at master coreutils/gnulib. https://github.com/coreutils/gnulib/blob/master/lib/qsort.c.

[10] Hash algorithm and collisions - PHP Internals Book. http://www.phpinternalsbook.com/hashtables/hash_algorithm.html.

[11] honggfuzz. https://github.com/google/honggfuzz.

[12] https://opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/kern/qsort.c. https://opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/kern/qsort.c.

[13] libc/stdlib/qsort.c. https://sourceforge.net/u/lluct/me722-cm/ci/f3ae3e66860629a7ebe223fdda3fdc8ffbdd9c6d/tree/bionic/libc/stdlib/qsort.c.

[14] libFuzzer - a library for coverage-guided fuzz testing - LLVM 3.9 documentation. http://llvm.org/docs/LibFuzzer.html.

[15] NetBSD: qsort.c,v 1.13 2003/08/07. http://cvsweb.netbsd.org/bsdweb.cgi/src/lib/libc/stdlib/qsort.c.

[16] NVD - CVE-2012-2098. https://nvd.nist.gov/vuln/detail/{CVE}-2012-2098.

[17] NVD - CVE-2013-4287. https://nvd.nist.gov/vuln/detail/{CVE}-2013-4287.

[18] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.

[19] PHP Vulnerability May Halt Millions of Servers - PHP Classes. https://www.phpclasses.org/blog/post/171-PHP-Vulnerability-May-Halt-Millions-of-Servers.html.

[20] Regular expression denial of service - redos - owasp. https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

[21] SantizerCoverage - Clang 4.0 documentation. http://clang.llvm.org/docs/SanitizerCoverage.html.

[22] Stack exchange network status - outage postmortem -Âăjuly 20, 2016. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[23] syzkaller. https://github.com/google/syzkaller.

[24] Why does stack overflow use a backtracking regex implementation? - meta stack overflow. https://meta.stackoverflow.com/questions/328376/why-does-stack-overflow-use-a-backtracking-regex-implementation.

[25] Berglund, M., Drewes, F., and van der Merwe, B. Analyzing catastrophic backtracking behavior in practical regular expression matching. *arXiv preprint arXiv:1405.5599* (2014).

[26] Bernat, G., Colin, A., and Petters, S. M. WCET analysis of probabilistic hard real-time systems. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE* (2002), IEEE, pp. 279–288.

[27] Betts, A., Merriam, N., and Bernat, G. Hybrid measurement-based WCET analysis at the source level using object-level traces. In *OASIcs-OpenAccess Series in Informatics* (2010), vol. 15, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[28] Cai, X., Gui, Y., and Johnson, R. Exploiting Unix file-system races via algorithmic complexity attacks. In *Security and Privacy, 2009 30th IEEE Symposium on* (2009), IEEE, pp. 27–41.

[29] Cha, S. K., Woo, M., and Brumley, D. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy (S&P)* (May 2015), pp. 725–741.

[30] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*, vol. 6. MIT press Cambridge, 2001.

[31] Crosby, S. A., and Wallach, D. S. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003), SSYM'03, USENIX Association, pp. 3–3.

[32] Godefroid, P., Levin, M. Y., Molnar, D. A., et al. Automated Whitebox Fuzz Testing. In *Proceedings of the 2008 Network and Distributed Systems Symposium (NDSS)* (2008), vol. 8, pp. 151–166.

[33] Haller, I., Slowinska, A., Neugschwandtner, M., and Bos, H. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *22nd USENIX Security Symposium (USENIX Security '13)* (Washington, D.C., 2013), USENIX, pp. 49–64.

[34] Holland, B., Santhanam, G. R., Awadhutkar, P., and Kothari, S. Statically-Informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on* (2016), IEEE, pp. 79–84.

[35] Householder, A. D., and Foote, J. M. Probability-based parameter selection for black-box fuzz testing. In *CMU/SEI Technical Report - CMU/SEI-2012-TN-019* (2012).

[36] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices 47*, 6 (2012), 77–88.

[37] Khan, U., and Bate, I. WCET Analysis of Modern Processors Using Multi-Criteria Optimisation. In *2009 1st International Symposium on Search Based Software Engineering* (2009).

[38] Kirrage, J., Rathnayake, A., and Thielecke, H. Static analysis for regular expression denial-of-service attacks. In *International Conference on Network and System Security* (2013), Springer, pp. 135–148.

[39] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices* (2005), vol. 40, ACM, pp. 190–200.

[40] McIlroy, M. D. A killer adversary for quicksort. *Softw., Pract. Exper. 29*, 4 (1999), 341–344.

[41] Mudduluru, R., and Ramanathan, M. K. Efficient flow profiling for detecting performance bugs. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (2016), ACM, pp. 413–424.

[42] Namjoshi, K., and Narlikar, G. Robust and fast pattern matching for intrusion detection. In *INFOCOM, 2010 Proceedings IEEE* (2010), IEEE, pp. 1–9.

[43] Petsios, T., Kemerlis, V. P., Polychronakis, M., and Keromytis, A. D. Dynaguard: Armoring canary-based protections against brute-force attacks. In *Proceedings of the 31st Annual Computer Security Applications Conference* (2015), ACM, pp. 351–360.

[44] Petsios, T., Tang, A., Stolfo, S., Keromytis, A. D., and Jana, S. NEZHA: Efficient Domain-Independent Differential Testing. In *Proceedings of the 38th IEEE Symposium on Security & Privacy,(San Jose, CA)* (2017).

[45] Petters, S. M. Bounding the execution time of real-time tasks on modern processors. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on* (2000), IEEE, pp. 498–502.

[46] Petters, S. M., and Farber, G. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on* (1999), IEEE, pp. 442–449.

[47] Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017).

[48] Shen, D., Luo, Q., Poshyvanyk, D., and Grechanik, M. Automating Performance Bottleneck Detection Using Search-based Application Profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (2015), ISSTA 2015, ACM, pp. 270–281.

[49] Shenoy, G. S., Tubella, J., and González, A. Improving the resilience of an IDS against performance throttling attacks. In *International Conference on Security and Privacy in Communication Systems* (2012), Springer, pp. 167–184.

[50] Shenoy, G. S., Tubella, J., and Gonz'lez, A. Hardware/Software Mechanisms for Protecting an IDS Against Algorithmic Complexity Attacks. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International* (2012), IEEE, pp. 1190–1196.

[51] Smith, R., Estan, C., and Jha, S. Backtracking algorithmic complexity attacks against a NIDS. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual* (2006), IEEE, pp. 89–98.

[52] Song, L., and Lu, S. Performance Diagnosis for Inefficient Loops. *Under Submission.*

[53] Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2016).

[54] Sun, X., Cheng, L., and Zhang, Y. A Covert Timing Channel via Algorithmic Complexity Attacks: Design and Analysis. In *Communications (ICC), 2011 IEEE International Conference on* (2011), IEEE, pp. 1–5.

[55] Tracey, N., Clark, J., McDermid, J., and Mander, K. A search-based automated test-data generation framework for safety-critical systems. In *Systems engineering for business process change: new directions*. Springer, 2002, pp. 174–213.

[56] Wegener, J., and Grochtmann, M. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems 15*, 3 (1998), 275–298.

[57] Wüstholz, V., Olivo, O., Heule, M. J., and Dillig, I. Static detection of dos vulnerabilities in programs that use regular expressions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2017), Springer, pp. 3–20.
[58] Zalewski, M. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

A WAF REGEXES

The slowdowns presented in Figure 7 correspond to inputs matched against the following regular expressions:

Regex 1:

```
(?i:(j|(&#x?0*((74)|(4A)|(106)|(6A));?))
([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|
(tab;)|(newline;))))*(a|(&#x?0*((65)|
(41)|(97)|(61));?))([\t]|(&((#x?0*(9|
(13)|(10)|A|D);?)|(tab;)|(newline;))
))*(v|(&#x?0*((86)|(56)|(118)|(76));?)
)([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|
(tab;)|(newline;))))*(a|(&#x?0*((65)|
(41)|(97)|(61));?))([\t]|(&((#x?0*(9|
(13)|(10)|A|D);?)|(tab;)|(newline;))))*
(s|(&#x?0*((83)|(53)|(115)|(73));?))(
[\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|
(tab;)|(newline;))))*(c|(&#x?0*((67)|
(43)|(99)|(63));?))([\t]|(&((#x?0*(9|
(13)|(10)|A|D);?)|(tab;)|(newline;))))*
(r|(&#x?0*((82)|(52)|(114)|(72));?))
([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|
(tab;)|(newline;))))*(i|(&#x?0*((73)|
(49)|(105)|(69));?))([\t]|(&((#x?0*(9|
(13)|(10)|A|D);?)|(tab;)|(newline;))))*
(p|(&#x?0*((80)|(50)|(112)|(70));?))
([\t]|(&((#x?0*(9|(13)|(10)|A|D);?)|
(tab;)|(newline;))))*(t|(&#x?0*((84)|
(54)|(116)|(74));?))([\t]|(&((#x?0*(9|
(13)|(10)|A|D);?)|(tab;)|(newline;))))
*(:|(&((#x?0*((58)|(3A));?)|(colon;)
))).)
```

Regex 2:

```
<(a|abbr|acronym|address|applet|area|
audioscope|b|base|basefront|bdo|
bgsound|big|blackface|blink|
blockquote|body|bq|br|button|caption|
center|cite|code|col|colgroup|
comment|dd|del|dfn|dir|div|dl|
dt|em|embed|fieldset|fn|font|
form|frame|frameset|h1|head|hr|
html|i|iframe|ilayer|img|input|ins|
isindex|kdb|keygen|label|layer|
legend|li|limittext|link|listing|
map|marquee|menu|meta|multicol|
nobr|noembed|noframes|noscript|
nosmartquotes|object|ol|optgroup|
option|p|param|plaintext|pre|q|
rt|ruby|s|samp|script|select|
server|shadow|sidebar|small|
spacer|span|strike|strong|style|
sub|sup|table|tbody|td|textarea|
tfoot|th|thead|title|tr|tt|u|ul|
var|wbr|xml|xmp)\\W
```

Regex 3:

```
(?i:<.*[:]vmlframe.*?[ /+\t]*?src[ /+\t]*=)
```