Evocatio: Conjuring Bug Capabilities from a Single PoC

Zhiyuan Jiang
NUDT
China

Shuitao Gan*
SKL-MEAC, Tsinghua University
China (*Corresponding author)

Adrian Herrera
Australian National University
Australia

Flavio Toffalini
EPFL
Switzerland

Lucio Romerio
EPFL
Switzerland

Chaojing Tang
NUDT
China

Manuel Egele
Boston University
USA

Chao Zhang
Tsinghua University, BNRist
China

Mathias Payer
EPFL
Switzerland

ABSTRACT

The popularity of coverage-guided greybox fuzzers has led to a tsunami of security-critical bugs that developers must prioritize and fix. Knowing the *capabilities* a bug exposes (e.g., type of vulnerability, number of bytes read/written) enables prioritization of bug fixes. Unfortunately, understanding a bug's capabilities is a time-consuming process, requiring (a) an understanding of the bug's *root cause*, (b) an understanding how an attacker may exploit the bug, and (c) the development of a patch mitigating these threats. This is a mostly-manual process that is qualitative and arbitrary, potentially leading to a misunderstanding of the bug's capabilities.

Evocatio automatically discovers a bug's capabilities. Evocatio analyzes a crashing test case (i.e., an input exposing a bug) to understand the full extent of how an attacker can exploit a bug. Evocatio leverages a *capability-guided fuzzer* to efficiently uncover new bug capabilities (rather than only generating a single crashing test case for a given bug, as a traditional greybox fuzzer does).

We evaluate Evocatio on 38 bugs (34 CVEs and four bug reports) across eight open-source applications. From these bugs, Evocatio: (i) discovered 10× more capabilities (that is, the number of unique capabilities induced by a set of crashes was 10× higher) than AFL++'s crash exploration mode; (ii) converted 19 of the 38 bugs to new bug types (demonstrating the limitations of manual qualitative analysis); and (iii) generated new proof-of-concept (PoC) test cases violating patches for 7 out of 16 tested CVEs, one of which still triggers in the latest version of the software.

CCS CONCEPTS

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

KEYWORDS

Bug Capability, Bug Triaging, Fuzzing

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560575

ACM Reference Format:
Zhiyuan Jiang, Shuitao Gan, Adrian Herrera, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. 2022. Evocatio: Conjuring Bug Capabilities from a Single PoC. 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, 15 pages. https://doi.org/10.1145/3548606.3560575

1 INTRODUCTION

Dynamic software testing techniques generate proof-of-concept (PoC) test cases that trigger bugs in a target program. Developers analyze these PoCs to locate the bug's *root cause* and then develop a patch to fix it. Modern bug-finding tools (notably, fuzzers) automatically (and quickly) discover large numbers of PoCs, greatly improving software security. However, as the number of discovered PoCs increases, developers face a challenge: *given finite developer resources, how can PoCs be analyzed quickly and efficiently to prioritize bug fixes based on bug severity?*

Clustering techniques for grouping PoCs based on their root cause are widespread [7, 35]. PoC clustering helps developers estimate the number of bugs in a set of PoCs. Afterwards, developers can focus their efforts on bugs (with PoCs corresponding to the same bug clustered together). However, simply clustering PoCs does not reveal the full set of *capabilities* an attacker can leverage when exploiting the given bug; after clustering, an analyst must invest resources to understand a bug's capabilities from its PoC (or cluster of PoCs).

Intuitively, capabilities allow an attacker to "program the weird machine" [17] that emerges when exploiting a bug. This may include reading (arbitrary) memory locations or altering the runtime state of the program. In the context of memory safety bugs, a capability is defined as a unique tuple of: bug type (e.g., out-of-bounds read/write, use-after-free, out-of-memory); access type (read or write); the number of bytes accessed; the name of victim object (e.g., `buffer`); the offset within the victim object; and the location (e.g., stack, heap, global). An example capability is `(OOB, read, 5, buffer, 10, stack)` which describes a 5 bytes out-of-bounds read access that starts at the 10^{th} byte to `buffer` on the stack. Any input that crashes with an unobserved "capability-tuple" is considered to expose a new capability (thereby increasing the number of discovered capabilities).

Understanding these capabilities is crucial for prioritizing bug fixes: an incomplete understanding may lead to unexploitable bugs receiving too much attention, or, *vice versa*, exploitable bugs being disregarded. *While all bugs should be fixed, security-critical vulnerabilities must be prioritized.*

Ideally, bug fixes are prioritized by the level of risk the bug poses under some threat model (e.g., when considering code execution, arbitrary memory writes are considered more severe than illegal reads at fixed, unmapped addresses). However, prioritization is hampered by the overwhelming number of bug reports developers face [16, 61]. At the time of writing, Google's automated fuzzing platform, `syzbot`, has filed 971 unfixed open bug reports on the Linux kernel [26]. Moreover, a single PoC (or cluster of PoCs) may not completely demonstrate the complete set of capabilities. For example, if a PoC crashes with a 1-byte out-of-bounds (OOB) read, is this the full extent of the bug? If an attacker were to exploit this bug, what capabilities would they have?

Interestingly, coverage-guided greybox fuzzers (the de facto standard for automatic bug finding) may obscure the answers to these questions: the goal of a fuzzer is to find bugs, not to investigate their capabilities. Fuzzers consider PoCs triggering the same bug through the same execution path as duplicates: they keep the first one encountered and discard all others, irrespective of their full set of capabilities. Consequently, the full attack power of a bug may remain hidden, placing a greater burden on the developer to judge a bug's capabilities. Automated techniques for exploring and assessing bugs' capabilities are required to reduce this burden.

In theory, Automatic Exploit Generation (AEG) [3, 13, 30, 72–74] can determine a bug's capabilities (in the context of a given threat model). However, most AEG engines rely on symbolic execution to generate an exploit [3]. In practice, symbolic execution engines are inherently incomplete (e.g., due to path divergence [4]) and suffer from state explosion. For example, while SyzScope [78] uncovers the security impact of fuzzer-exposed kernel bugs, its application is limited by state explosion [1, 66].

Observing the lack of reliable tools to automatically asses a bug's capabilities, we propose Evocatio[1]: a tool for *conjuring* (i.e., discovering) a bug's capabilities and assisting human analysts in severity estimation. Evocatio reduces the amount of time spent on bug analysis while improving the developer's evaluation of a bug.

Evocatio leverages fuzzing and sanitization to uncover new capabilities for memory corruption bugs, ensuring these capabilities are captured as separate PoCs. From this, we propose a framework to derive a *quantitative* score of a bug's severity according to a given threat model.

In summary, we make the following contributions:

- Evocatio, a system for assisting developers prioritize and develop critical bug fixes. Evocatio consists of a custom fuzzer and sanitizer that uncover new attack capabilities beyond those exercised by a single PoC (as discovered by a traditional greybox fuzzer).
- A demonstration of Evocatio's utility when applied to memory corruption bugs. We show how Evocatio can automatically (a) derive a quantitative severity score (analogous to existing vulnerability scoring systems), and (b) verify the efficacy of software patches.
- An evaluation of Evocatio using 38 bugs across eight open-source applications. From these bugs, Evocatio: (i) discovered 10× more capabilities than AFL++'s crash exploration mode; (ii) converted 19 of the 38 CVEs to new bug types; and (iii) generated new PoCs that violated the patches for 7 out of 16 tested CVEs, one of which triggers in the latest version.

Our results highlight the need for automated bug assessment and show that Evocatio is well-placed to achieve this goal.

2 BACKGROUND

The following sections formalize capabilities (Section 2.1), discuss existing approaches for vulnerability scoring (Section 2.2), and provide a motivating example for our work (Section 2.3).

2.1 Using Capabilities to Program the Weird Machine

A vulnerable, buggy program exposes a *weird machine* that can be "programmed" by an attacker via an exploit [5, 17]. The ability to program this weird machine depends on the *capabilities* a particular bug exposes. Put another way, when modeling a weird machine as a collection of "weird states" in an *intended finite state machine* [17], different capabilities allow an attacker to transition to different weird states. Ultimately, a capability defines what a bug "can do" (when exploited).

For example, the ability to write a sequence of bytes to a function pointer (e.g., in a vtable data structure) in the heap may allow an attacker to crash or redirect execution in the target program (depending on the values written). Similarly, the ability to read several bytes of stack memory from which sensitive data can be gleaned (such as in CVE-2020-11104 [70]) is another capability.

Depending on the defender's threat model, some capabilities may be more severe than others. For example, if system up time is critical, then a `null` pointer dereference is a severe bug (that may crash the system). However, if arbitrary code execution is the primary concern, then the same bug may be relatively benign on its own. Regardless, it is difficult to assess the severity of a bug without a complete understanding of the capabilities the bug possesses; intuitively, the more capabilities a bug has, the more severe the bug is. This lack of understanding may lead to the misclassification of a bug's severity and/or release of an incomplete patch.

2.2 Current Approaches for Measuring Bug Severity

The Common Vulnerability Scoring System (CVSS) [51] is the de facto measure of a bug's severity. Several indicators (e.g., attack vector, complexity, user interaction) are manually aggregated and summarized by a value between zero and ten; the higher the value, the higher the severity. Some of these indicators are qualitative and cannot be generated programmatically (e.g., attack complexity). Analysts must carefully inspect the bug and determine a value for these qualitative metrics. This is a time-consuming, error-prone, and subjective process; potentially leading different analysts to assign different scores to the same bug [9, 43]. Moreover, analysts

[1] *Evocatio* is Latin for "calling forth"—or conjuring—a deity.

Listing 1: A motivating example program.

```
1  void foo(char *src, char *des){
2    int len_one = src[4]; // Single capability byte
3    // Multiple capability bytes
4    int len_two = src[20] + src[22] * src[33];
5
6    memcpy(des, src, len_one);
7    memcpy(des + len_one, src, len_two);
8  }
9
10 int main(){
11   // Difficult for symbolic execution
12   char *src = inflate(input_file);
13
14   // Control flow bytes
15   if(src[0] == 'a' && src[1] == 'b'){
16     char *data = parseData(src);
17     if (strlen(data) != 120)
18       return 1;
19
20     // Capability bytes with constraints
21     int buf_size = src[40]+src[41]+src[42]+src[43];
22     if (src[40] + src[41] > 50 || src[42] < 100)
23       return 1;
24
25     char *dest = malloc(buf_size);
26     foo(src, dest); // Buggy function
27   }
28   return 0;
29 }
```

are typically provided with only a single PoC [65] from which they must manually determine the full extent of the bug.

For these reasons, we argue that a more complete understanding of a bug's capabilities empowers analysts to accurately estimate a vulnerability score that truly reflects the full extent of the bug.

2.3 Motivating Example

The program[2] in Listing 1 motivates the need for Evocatio. This program contains a heap buffer overflow in `foo`, which may crash at Lines 6 and 7 depending on the values of `buf_size`, `len_one`, and `len_two`. Importantly, the abilities granted to an attacker (when exploiting this overflow) vary depending on these values.

The program first calls `zlib`'s `inflate` (Line 12) to read the contents of the input file. Most symbolic execution engines fail at this point, because they are often unable to accurately model external libraries (such as `zlib`) and succumb to state explosion during the decompression process. Following decompression, a validity check is performed on the first two bytes of the input data (Line 15). We label these two bytes "*control-flow bytes*" because they impact the program's control flow. Reaching the bug at Lines 6 and 7 requires satisfying these control-flow constraints.

Once these control-flow constraints are satisfied, the input bytes at offsets 40–43 are read (Line 21). The data contained at these offsets effects the size of the buffer used in `foo`, ultimately determining the size of the overflow. We label these four bytes "*capability bytes*" (astute readers will recognize these capability bytes correspond to changes in data flow). Additional checks are performed on bytes 40–42 (Line 22), ensuring these bytes fall within a certain range. These bytes are thus both capability *and* control-flow bytes.

The `foo` function (Lines 11 to 22) demonstrates different types of capability bytes: single capability bytes (Line 2), where a single byte affects the capability and sequences of capability bytes (Line 4, here the three-byte sequence at offsets $\{20, 22, 33\}$ affects the capability.

Listing 1 contains ten critical bytes: two control-flow bytes, five capability bytes, and three bytes that are both control-flow and capability bytes. Identifying these bytes is important for efficiently discovering new capabilities, ensuring time is not wasted exploring erroneous states in the state space (i.e., states unrelated to the given bug). Satisfying control-flow bytes ensures a new PoC maintains the same control-flow that triggered the original bug, while exploring different capability byte values leads to new PoCs that manifest the same bug in different ways. We use fuzzing to automatically explore this state space.

3 EVOCATIO

Fuzzers discover bugs in target programs by exercising a large number of randomly-generated inputs. Crash-inducing inputs (exercising one or more bugs) grant an attacker control over the resulting weird machine. Importantly, these inputs also grant an attacker a set of capabilities that allow them to program this weird machine.

Given a crash-inducing input (generated by a fuzzer), *what level of control is granted to an attacker when they trigger the underlying bug?* Evocatio answers this question by systematically exploring the range of capabilities a bug possesses from a given PoC. Evocatio consists of the following components (Fig. 1):

Capability detection (CapSan). New capabilities are detected via CapSan. CapSan extends AddressSanitizer's (ASan) visibility into triggered memory safety bugs (Section 3.1).

Capability discovery (CapFuzz). New capabilities are discovered via CapFuzz. CapFuzz perturbs an input PoC to uncover new capabilities by rescoping traditional coverage-guided fuzzers. CapFuzz uses CapSan to hone in on new capabilities that grant an attacker more power and control (Section 3.2).

The newly-discovered capabilities can be used by analysts and developers alike. We demonstrate two applications—estimating bug severity and patch testing—in Section 4.

3.1 Capability Detection

Evocatio requires a mechanism to efficiently extract capabilities from a given PoC. In the spirit of fuzzing, the most natural way to achieve this is through a sanitizer. Indeed, sanitizers and fuzzers are often paired together, increasing the fuzzer's sensibility towards specific behaviors. Given our focus on memory errors, we adopt AddressSanitizer (ASan) [63] as our capability detector and collector.[3] ASan instruments the target program so that it crashes on memory safety violations. Importantly, ASan generates a detailed report whenever an invalid memory access is detected. While fuzzers typically ignore this report (as they only care if a crash occurs), it contains rich bug information required to determine when a new capability is discovered. Capability detection is based on the five properties listed in Section 2.1. We modify ASan—dubbing our tool CapSan—to expose a machine-readable version of this crash report.

[2] Modeled after CVE-2016-9532. We have simplified the logic and variable names.

[3] The design can be extended to cover other types of sanitizers, provided capabilities can be extracted from the sanitizer.

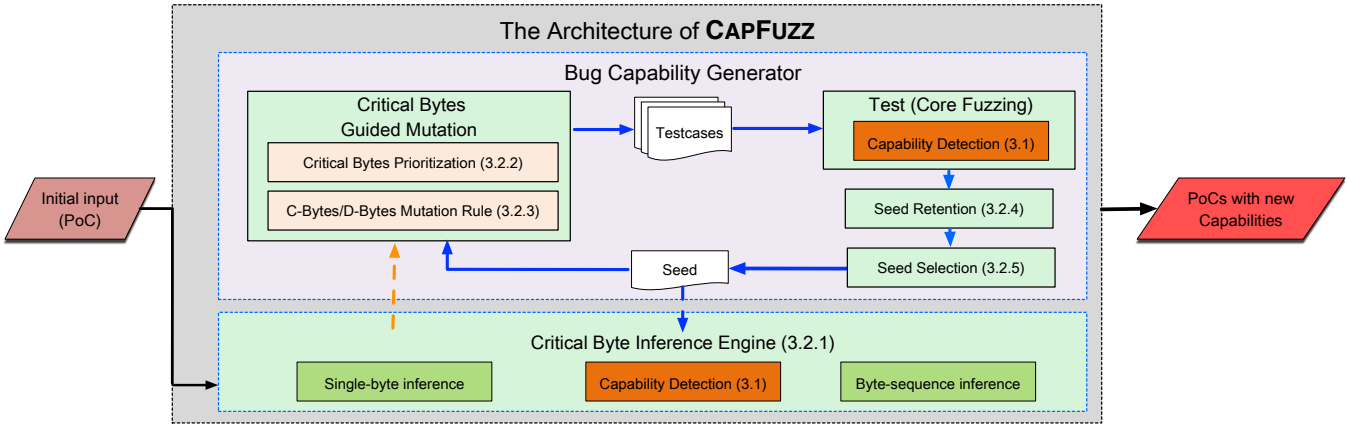


Figure 1: Evocatio workflow.

3.1.1 Forced Execution to Detect Out-of-Bounds Accesses. By default, ASan aborts program execution at the first error encountered [25]. For example, if a bug overwrites multiple OOB bytes, ASan aborts execution at the first OOB write. This design works in most scenarios; i.e., when analysts needs to locate the cause of a crash. And while this limits false positives, it also loses the opportunity for deep exploration of bug capabilities; ASan cannot discover bugs hiding behind other bugs (e.g., several OOB writes).

For example, CVE-2021-3156 (Listing 2) performs multiple writes when copying bytes from the `from` buffer to the `to` buffer. However, because the code copies a single byte at a time (Line 5), ASan terminates the program on the first OOB write, reporting an OOB length of *one*. Instead, the bug allows the attacker to overflow multiple bytes (provided the conditional on Line 3 remains satisfiable). Here, ASan's early exit hides the complete set of bug capabilities.

CapSan mitigates this limitation by continuing execution after an error has been detected to fully explore bug capabilities (by disabling ASan's `halt_on_error` option). To correctly detect the OOB capability for each buffer (and prevent false positives), CapSan records the overflow length and identifies the data structure being overflowed. This allows CapSan to distinguish different overflow lengths across different buffers. We stop exploration whenever a new data structure is reached (intuitively, this means we are no longer triggering the same bug).

Listing 2: Code snippet from CVE-2021-3156.

```
1  if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
2    while (*from) {
3      if (from[0] == '\\' && !isspace(from[1]))
4        from++;
5      *to++ = *from++;
6    }
7  }
```

3.2 Capability Discovery

Coverage-guided greybox fuzzers explore a target program's state space by continuously generating new inputs via random mutation. Fuzzers do not suffer from the scalability limitations inherent in approaches like symbolic execution, making them ideal for discovering new capabilities in real-world programs. Unfortunately, existing fuzzers are designed to explore the target's code, thus moving exploration away from recently-discovered bugs. In doing so, fuzzers may ignore crashes exposing other capabilities (associated with the same bug). This implies analyzing a single fuzzer-produced PoC is insufficient for fully capturing a bug's security impact. Conversely, a complete security assessment must—at least—consider all unique crashes for the same bug. We propose CapFuzz to meet these requirements. CapFuzz is a capability-driven fuzzer for discovering new crashes (and thus, new capabilities) from an initial PoC. CapFuzz is comprised of two stages: a module for labeling and prioritizing critical bytes leading to new capabilities (Sections 3.2.1 to 3.2.2), and a mutation engine for exploring the capability space (Sections 3.2.3 to 3.2.5).

3.2.1 Critical Bytes Inference. As shown in Section 2.3 and by prior work [21, 46, 76], only a subset of the input bytes affect program behavior, and hence a bug's capabilities. Thus, efficient capability exploration requires understanding which bytes impact a bug's capabilities. Per Section 2.3, we classify critical bytes into two categories: those affecting control flow (C_{byte}), and those affecting data flow (D_{byte}). Algorithm 1 outlines our approach for identifying and categorizing critical bytes.

Single-byte inference. Prior work [21, 76] has successfully leveraged single-byte inference techniques to improve a fuzzer's ability to expand code coverage. In contrast, Evocatio uses single-byte inference to determine which bytes contribute to a new capability. It does this by performing an exhaustive search to label a given input byte a C_{byte} or D_{byte} (or both). Lines 2 to 10 in Algorithm 1 describes this process. For each input byte i, every possible value v is tested. Byte i is labeled a C_{byte} if (a) a new v induces a change in $\mathcal{P}$'s control flow and (b) it does not introduce a new capability. Conversely, i is labeled a D_{byte} if a change in capabilities is detected. We infer changes in control flow through variations in the coverage map (recorded in p), while data flow changes are inferred through differences in capabilities (recorded in $\mathbb{C}$).

Byte-sequence inference. The previously-described exhaustive search is fast and efficient at performing single byte inference, but is oblivious to multi-byte relations (e.g., due to a specific grammar).

Algorithm 1: Critical Bytes Inference.

Input: Instrumented program $\mathcal{P}$, initial PoC S, exploration times t
Output: C_{byte}, D_{byte}

```
/* Get bug capability ℂ and coverage map p, O is the
capability of uncrashed seed */
1  {ℂ, p} ← Execute(P, S)
2  foreach i ∈ {0, 1, ... |S|} do /* Single byte inference */
3      S' ← S
4      foreach v ∈ {0, 1, ... 255} do /* Exhaustive search */
5          S'[i] ← v
6          {ℂ', p'} ← Execute(P, S')
7          if p' ≠ p ∧ (ℂ' = O ∨ ℂ' = ℂ) then
8              C_byte[S] ← C_byte[S] ∪ {i}
9          else if ℂ' ≠ ℂ ∧ ℂ' ≠ O then
10             D_byte[S] ← D_byte[S] ∪ {{i}}
11 repeat/* Byte-sequence inference */
12     z ← RndByteSequenceSelect(S)
13     S' ← RndMutate(S, z)/* Apply a random mutation */
14     {ℂ', p'} ← Execute(P, S')
15     if ℂ' ≠ ℂ ∧ ℂ' ≠ O then
16         z' ← ByteSequenceReduction(z)
17         D_byte[S] ← D_byte[S] ∪ {z'}
18     t ← t − 1
19 until t = 0
```

These relations require related bytes to be mutated together. An example of a multi-byte relation is shown at Lines 4 and 22 in Listing 1. These sequences can quickly grow in size, making an exhaustive search impossible; fortunately, fuzzing has been shown to efficiently handle such large search spaces. Lines 11 to 19 (Algorithm 1) describes our fuzzing-based approach for multi-byte sequence inference.

Our multi-byte inference first selects a random subset of input bytes and mutates them to create S' (Lines 12 to 13). The program $\mathcal{P}$ is then executed with S'. The sequence is added to D_{byte} if new capabilities are discovered. Unfortunately, the discovered sequences may be irreducible (i.e., redundant bytes cannot be removed) and noisy (due to the random selection of input bytes). This noise exists in bytes not contributing to changes in $\mathcal{P}$'s behavior. Incorrectly identifying these bytes as critical significantly reduces performance (because time is wasted mutating these bytes). Noisy bytes must be filtered out as early as possible.

CapFuzz handles noisy bytes with a `ByteSequenceReduction` operation (Line 16). When a sequence is found, `ByteSequenceReduction` restores as many bytes as possible to their original value, while simultaneously ensuring new program behaviors remain. Because byte sequences may be large, we adopt a "divide-and-conquer" approach (inspired by delta debugging [44, 47]) where we restore blocks of bytes, rather than single bytes. Here, all bytes in a block are restored to their original value: if the new behavior persists, the reduction was successful; otherwise, a smaller block size is selected and the process repeats until we reach a single byte.

3.2.2 Critical Bytes Prioritization. Critical bytes inference may result in a large number of sequences in D_{byte}. This is particularly pronounced when the target program has a complex input syntax. Although mutating any of these sequences will introduce new capabilities, careful prioritization of sequence mutation is crucial to improve CapFuzz's efficiency. Sequences contributing most to

Algorithm 2: Mutation Rule on Critical Bytes.

Input: Instrumented program $\mathcal{P}$, initial PoC s, C_{byte}, D_{byte}
Output: Seed after mutation s'

```
/* seqs is a set of sequence, seq is one sequence */
1  seqs ← PrioritizedSeqSelect(D_byte)
2  s' ← s
3  foreach seq ∈ seqs do
4      if seq ∈ D_byte then
5          bytes ← RndPositionsSelect(seq)
6          foreach byte ∈ bytes do
7              if byte ∈ C_byte then
8                  continue
9              s'' ← RndMutate(s', byte)
10             s' ← s''
```

the discovery of new capabilities should be prioritized over other sequences. At each cycle, for each sequence $s \in S$, we use the following predicate for determining if s induces a new capability c_i:

$$P(c_i, s) = \begin{cases} 1 & c_i \text{ is from } s \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

This predicate is applied over the last $N = 20$ seeds[4] to determine their contribution $W(s)$:

$$W(s) = \sum_{i=0}^{N} P(c_i, s), \tag{2}$$

Here, $W(s)$ represents the *most energetic sequence* that changes dynamically over time. By prioritizing sequences with higher values of $W(s)$, CapFuzz focuses on sequences that (a) find more capabilities, and (b) is faster at finding them. Sequences are prioritized using the `PrioritizedSeqSelect` function (Line 1 in Algorithm 2).

3.2.3 Mutation. CapFuzz mutates critical bytes to explore the surrounding state space and discover new capabilities. CapFuzz uses the information collected during the critical byte inference phase (Section 3.2.1) to discover new crashes sharing the same control flow with the original PoC. We recall the inference phase assigns a C_{byte} or D_{byte} label to each input byte (or sequence of) affecting the control flow and/or data flow. To concentrate exploration towards data flow—thus finding new crashes that follow the same execution path as the original PoC—CapFuzz only considers D_{byte}s and sequences during the mutation phase (Algorithm 2).

Critical bytes are tightly related to the input length; changing the input length invalidates the previously-collected information. CapFuzz therefore keeps the input length constant and mutations changing the input length are disabled (e.g., inserting, copying, and splicing). Similarly, the deterministic mutation phase is skipped because this process previously occurred during the inference phase.

3.2.4 Seed Retention. CapFuzz is designed to discover new capabilities. Thus, it only retains crashing test cases that introduce new capabilities. These test cases are also stored in the fuzzing queue, ensuring they are also mutated (helping guide the fuzzer towards uncovering new capabilities).

A check is performed on each newly-generated test case to verify whether it introduces new capabilities. These checks are frequent

[4]We empirically select N and leave parameter optimization to future work.

and could become a performance bottleneck for long fuzzing campaigns (due to an increased number of capabilities required to be checked). We use a hash table to make capability queries efficient and prevent the number of capabilities from impacting performance.

3.2.5 Seed Selection. CapFuzz follows the strategy of traditional coverage-guided fuzzers and stores "interesting" test cases (i.e., those reaching new code) in the fuzzer queue for further mutation. At any point during the fuzzing campaign there may be many test cases in the queue. Moreover, the probability a given test case helps find new capabilities dynamically evolves over time. For these reasons, a mechanism for selecting the most promising seeds for mutation is required. We introduce the following two complementary rules for seed selection, where a *new sequence* is defined as one for which no test cases have been executed yet: (1) prefer the seed generated by a new sequence, and (2) prefer the seed generated by the most energetic sequence. CapFuzz applies these rules sequentially. First, it verifies whether a new sequence exists. If true, its first test case is set to the highest priority and the sequence is labeled as tested (i.e., it is no longer a new sequence, and further test cases of this sequence will not be set to the highest priority). If no new sequence is found, CapFuzz identifies the most energetic sequence using Eq. (2) and prioritizes that one instead.

4 APPLICATIONS

We demonstrate two scenarios where the discovery of bug capabilities is useful to analysts and developers: bug severity assessment (Section 4.1) and validating patch efficacy (Section 4.2).

4.1 Severity Assessment

Defenders must prioritize which bug to fix based on the bug's severity. However, scoring bug severity is a subjective process, and is highly dependent on the defender's threat model. Automating this scoring process requires developing context-dependent rules for determining a bug's severity across multiple dimensions. While this is impossible in the general case, we demonstrate how bug capabilities help this process.

We design a scoring system that uses the bug capabilities discovered by Evocatio to automatically derive a severity score. Our system is designed for a threat model in which an attacker desires to achieve remote code execution over a target program. However, the model can be configured to adhere to more specific users' scenarios. This model is presented in detail in Section 4.1.1, and experimental results are presented in Section 6.3. Our results show combining capabilities (discovered by Evocatio) with an automatic scoring system improves the efficiency of severity assessment.

4.1.1 Scoring Bug Severity. We merge the individual capability reports produced by CapFuzz into a *bug capability report* composed of the following six metrics (based on those introduced in Section 2.1):

Bug type: Different bug types—e.g., stack-buffer-overflow (SOF), heap-buffer-overflow (HOF), and UAF—are discoverable by CapFuzz. A larger variety of bug types leads to a higher probability of satisfying an exploit condition (i.e., the minimum combination of capabilities required to exploit a vulnerability).
Max. length of OOB reads/writes: The larger the length of OOB reads/writes, the greater the area to exploit.
Readable/writable address ranges The larger the range of addresses that can be read from/written to, the higher the probability an arbitrary read/write succeeds.
Num. of OOB objects: The larger this number, the more opportunity an attacker has for different attacks (e.g., SOF, HOF).
Max. OOB size objects: CapFuzz records the different sizes of the overflown buffer, reporting information about the originating memory object (ASan reports the closest object). The ability to overflow objects with different sizes helps generate an exploit (e.g., the attacker can target different locations).
Num. of different read/write offsets: The ability to read/write at different offsets within the original object gives an attacker more flexibility (as more/different memory regions are potentially readable/writable).

Evocatio aggregates these metrics across multiple PoCs to compute a bug severity score (between zero and ten) for each primitive type (i.e., read and write). Providing two scores—rather than a single score—allows a developer to more-accurately assess the impact of a bug. Again, we stress that a scoring system requires a specific threat model. Here we provide (a) an example threat model, and (b) a configurable scoring system instance to show how capabilities can be used to measure bug severity (based on human knowledge). Further details are provided in our implementation.

4.2 Patch Testing

A developer requires a deep understanding of a given bug to develop a complete patch. Ideally, this patch prevents transitions into all weird states relating to the bug. Evocatio is well-placed to help developers implement complete patches, because it provides them with an understanding of what weird states are reachable (through a bug's capabilities). Moreover, the PoCs generated by Evocatio can be used to test proposed patches. We demonstrate this in Section 6.4, revealing patches for multiple CVEs were in fact incomplete.

5 IMPLEMENTATION

CapFuzz and CapSan (Section 3) are written in 7K LoC of C. Our example capability aggregation and scoring system (Section 4.1) is written in 1.5K LoC of Python. We make our code available at https://github.com/HexHive/Evocatio to help future studies in this area.

CapSan leverages ASan's API for accessing crash details at runtime. When a new crash is encountered, the capabilities of that crash are deduplicated with those previously encountered. This means comparisons between capabilities is a frequent operation, and is thus required to be as efficient as possible. During fuzzing, capabilities are stored and compared through their hashes.

Prior to fuzzing, Evocatio performs the dual-phase inference process described in Section 3.2.1. The first phase (single byte inference) is an exhaustive search and always runs to completion. The second phase (sequence inference) is fuzzing based and thus has a variable execution time. We empirically determined 10 minutes to be a reasonable default.

Capability exploration uses seeds that introduce new capabilities (discovered during the inference phase) as the initial corpus,

and the critical bytes mapping to guide mutation. The capability-driven fuzzing engine is built on top of AFL++ [19], which has been modified to use the capability hash as a guidance metric.

While our current prototype is tailored to memory safety bugs, Evocatio is implemented so other bug types and threat models can also be explored. For example, rather than building on ASan, CapSan could be modified to leverage UBSan and thus explore capabilities associated with undefined behavior bugs. Similarly, other severity scoring systems can be constructed based on other threat models. We leave this for future work.

6 EVALUATION

We evaluate Evocatio by answering the following research questions:

- **RQ1** Does Evocatio discover a greater range of capabilities (compared to other crash exploration tools)? (Section 6.1)
- **RQ2** How do the different design components affect Evocatio's performance? (Section 6.2)
- **RQ3** Can Evocatio be applied to severity scoring? (Section 6.3)
- **RQ4** Can Evocatio be applied to patch testing? (Section 6.4)

Finally, we use a case study to illustrate the practical benefits of Evocatio (Section 6.6).

Benchmark suite. Our benchmark is composed of 38 bugs belonging to six different bug types: 34 CVEs and four open issues across eight targets. Table 1 summarizes the target programs. We selected these targets due to their popularity in fuzzing research [2, 21, 29], functionality, code diversity, development activeness, and varying code base sizes (ranging from 50k to 2M LoC).

Baseline. Capability/crash exploration is an under-explored research topic; AFL++'s "crash exploration mode"(`afl-cexp`) is the only tool we are aware of. `Afl-cexp` is similar to CapFuzz: it takes a single PoC as input and mutates it in an attempt to generate new crashes. We compare Evocatio to `afl-cexp` in our evaluation.

Performance metrics. We evaluate performance by considering (a) the number of new capabilities discovered over time, and (b) the diversity of these capabilities.

Testing time. Unlike traditional fuzzers, crash exploration tools cannot run for long periods of time; doing so introduces lengthy delays in bug fixing. In our evaluation, we found eight hours to be a suitable compromise between execution time and capability discovery. We run Evocatio and `afl-cexp` for eight hours unless otherwise stated. Experiments are repeated five times to ensure statistical significance.

Table 1: Benchmark summary. Bug types are: heap buffer overflow (HOF), integer overflow (IOF), use-after-free (UAF), stack buffer overflow (SOF), global buffer overflow (GOF), and off-by-one (OBO).

Program	Description	LoC	Bugs	
			Type	#
Libtiff	Image library	84K	HOF, IOF	11
Libming	Flash library	114K	HOF, UAF	9
Binutils	Binary tools	2.1M	HOF, UAF, SOF, OBO	6
Libsixel	Image library	37K	HOF, SOF	3
Jasper	Image manipulation	53K	HOF	5
Libsndfile	Audio manipulation	85K	HOF	1
Nasm	x86 asssembler	122K	UAF	1
Fig2dev	Graphics creation	47K	UAF, SOF, GOF	2

Experiment environment. All experiments were conducted on an Ubuntu 18.04 LTS server, with 200 GiB of RAM on an Intel® Xeon® Gold 6254 3.10 GHz CPU with 60 cores.

6.1 Capability Discovery (RQ1)

Table 2 summarizes the capabilities discovered across our benchmark.[5] Here, we report the *bug effect*; that is, the measurable consequence that follows from the bug's root cause. For example, the root cause of *CVE-2016-9532* is an integer overflow. However, the *effect* of this overflow is a heap-overflow (HOF)/ stack-overflow (SOF). This distinction allows us to highlight additional bug capabilities.

Overall, Evocatio successfully discovers new capabilities for 50 % of the bugs in Table 1. For example, both integer overflow (root cause of CVE-2016-9532) and off-by-one errors (root cause of CVE-2021-3156) lead to OOB accesses. Thus, we use the bug type in Table 2 to show how many different effects can be discovered by Evocatio. This experiment also shows Evocatio is not limited to memory corruption errors, but can also identify other bug types (e.g., UAF in CVE-2016-10092).

Prior work has shown more flexible overflows simplify exploitation, resulting in higher security impact [12, 71]. The correlation between capability diversity and new bug types in Table 2 empirically confirms this: all bugs for which a new effect was discovered had a wide range of *origin*, *size* and *origin offset* values. The capabilities found by Evocatio help developers at prioritizing those bugs with a more diverse range of capabilities.

6.1.1 CapFuzz vs. `afl-cexp`. Here we compare Evocatio against AFL++'s crash exploration mode (`afl-cexp`). Table 3 summarizes the total number of capabilities and seeds obtained after 8 hours of execution,[6] while Fig. 2 shows the evolution of capabilities/seeds over time for 8 CVEs that we choose as representative of the bugs effect in our benchmark. This allows us to compare the results of the two tools across targets and bug types.

Table 3 shows the number of different discovered unique crash capabilities (e.g., unique array access index, unique size of the read-/write, or the bug type). Evocatio finds 10× more capabilities than `afl-cexp`. Evocatio generates significantly more unique seeds than `afl-cexp`, not only because Evocatio is better at discovering new capabilities, but also because `afl-cexp` will wrongly discard some inputs (`afl-cexp` may exercise a capability but discard the corresponding inputs due to its seed selection strategy).

Figure 2 shows `afl-cexp`'s ability to uncover new capabilities quickly plateaus. While the number of crashing seeds increases, the number of capabilities remains low. This is due to `afl-cexp`'s code-coverage-based feedback mechanism; it is unable to detect changes in data flow. As a result, `afl-cexp` finds new seeds, but they trigger the same capabilities along different execution paths. In contrast,

[5] We applied an extra eight hours (on average) of manual analysis with the aid of the original bug reports to verify the discovered capabilities.

[6] We considered different invalid memory addresses. We also merged the consecutive addresses with ASLR disabled. Therefore, the numbers are larger than Table 2 in which memory addresses are not shown.

Table 2: Capabilities discovered by EVOCATIO. *Bug effect* refers to the bug effect witnessed during a crash, expressed as "original bug effect[new bug effect]". HOF = heap overflow, UAF = use-after-free, SOF = stack overflow, GOF = global buffer overflow, #W = wild address read, #N = null pointer dereference, #A = allocation size too bug, #U = unknown crash. Newly introduced effects are in blue. The *sizes* column contains the size of the access, *origins* the count of different objects accessible from a valid memory access, *origin sizes* the size of these objects, and *origin offsets* the offsets from these objects. The *sizes*, *origin sizes*, and *origin offsets* columns differentiate between *read* and *write* capabilities. For these columns we use the notation `[min..x..max] y` to indicate a range (rather than total count). Here, `x` is the value of the capability for the original PoC, and `y` the number of unique values observed within this range. Finally, *origins* differentiates between *stack* and *heap*, reporting the total count of origin objects found in each location.

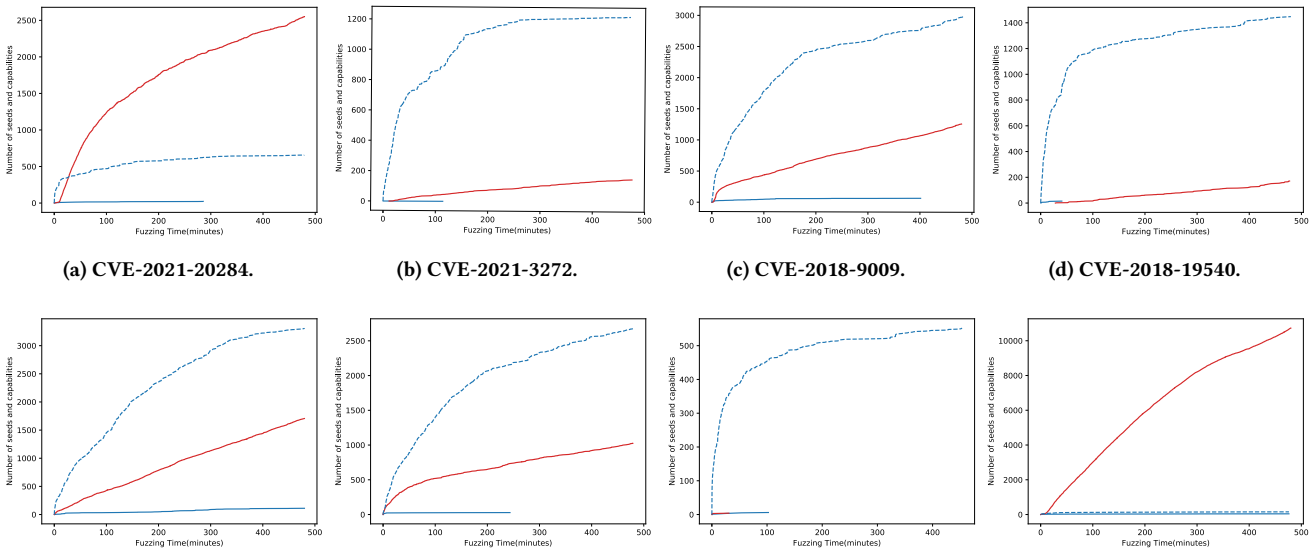
CVE	Bug Effect	Size		Origin		Origin Size		Origin Offset	
		Read	Write	Stack	Heap	Read	Write	Read	Write
issue-278	HOF[-]	[2^{0}..2^{13}] 1542	[2^{3}..2^{12}] 4	0	34	[2^{3}..2^{20}..2^{27}] 39263	[2^{8}..2^{19}] 5	[2^{1}..2^{3}..2^{8}] 214	[2^{0}] 1
issue-277	HOF[-]	[2^{0}..2^{2}] 3	-	0	1	[2^{0}..2^{3}..2^{5}] 97	-	[2^{0}] 1	-
issue-275	HOF[UAF]	[2^{0}] 1	[2^{0}] 1	0	2	[2^{9}..2^{19}] 25560	[2^{0}..2^{18}] 7387	[2^{0}] 1	[2^{0}] 1
issue-269	HOF[-]	[2^{0}..2^{2}] 3	-	0	2	[2^{0}..2^{3}..2^{6}] 78	-	[2^{0}] 1	-
CVE-2018-17795	HOF[-]	-	[2^{2}] 1	0	1	-	[2^{6}] 1	-	[2^{0}] 1
CVE-2018-12900	HOF[#U]	[2^{0}] 1	[2^{0}] 1	0	5	[2^{0}..2^{25}] 43852	[2^{5}..2^{23}..2^{24}] 2178	[2^{0}..2^{13}] 1602	[2^{0}] 1
CVE-2018-8905	HOF[-]	-	[2^{0}] 1	0	1	-	[2^{3}] 1	-	[2^{0}] 1
CVE-2016-10272	HOF[-]	[2^{0}..2^{0}] 1	[2^{0}] 1	0	5	[2^{14}..2^{15}] 6	[2^{8}..2^{28}] 33440	[2^{2}..2^{2}..2^{4}] 4	[2^{0}..2^{5}] 33
CVE-2016-10092	HOF[UAF]	[2^{0}] 1	[2^{0}..2^{6}] 259	0	11	[2^{0}..2^{27}] 611	[2^{3}..2^{27}] 8746	[2^{0}..2^{1}] 4	[2^{0}..2^{14}] 1817
CVE-2016-9532	HOF[SOF]	[2^{0}..2^{3}] 2	[2^{0}] 1	1	9	[2^{0}..2^{20}] 65374	[2^{0}] 1	[2^{0}..2^{8}] 2	[2^{0}] 1
CVE-2016-9273	HOF[SOF]	[2^{3}] 1	[2^{2}..2^{3}] 2	0	1	[2^{3}] 1	[2^{3}..2^{3}] 1	[2^{0}] 1	[2^{7}..2^{7}] 1
CVE-2020-11895	HOF[#A|HOF|UAF]	[2^{0}..2^{1}..2^{3}] 4	[2^{2}..2^{3}] 6	0	370	[2^{0}..2^{11}..2^{13}] 204	[2^{2}..2^{18}] 13	[2^{0}..2^{4}..2^{11}] 74	[2^{0}] 1
CVE-2020-11894	HOF[#W|UAF]	[2^{0}..2^{3}] 5	[2^{0}..2^{3}] 10	0	346	[2^{0}..2^{11}..2^{13}] 152	[2^{2}..2^{18}] 11	[2^{0}..2^{3}..2^{11}] 45	[2^{0}] 1
CVE-2020-6628	HOF[#W|UAF|#N]	[2^{0}..2^{3}] 5	[2^{3}..2^{6}] 7	0	170	[2^{0}..2^{11}..2^{13}] 90	[2^{2}..2^{18}] 8	[2^{3}..2^{11}] 26	[2^{0}] 1
CVE-2019-16705	HOF[#W|UAF]	[2^{0}..2^{0}..2^{10}] 3	[2^{5}..2^{5}] 1	0	42	[2^{0}..2^{8}..2^{12}] 81	[2^{15}..2^{18}] 11	[2^{3}..2^{10}] 44	[2^{0}] 1
CVE-2019-9114	HOF[-]	-	[2^{0}] 1	0	26	-	[2^{2}..2^{4}..2^{5}] 49	-	[2^{0}] 1
CVE-2018-20591	HOF[UAF|#N]	[2^{0}..2^{0}..2^{10}] 3	[2^{2}..2^{3}] 6	0	168	[2^{0}..2^{3}..2^{13}] 157	[2^{2}..2^{18}] 14	[2^{0}..2^{3}..2^{11}] 44	[2^{0}] 1
CVE-2018-9009	UAF[#W|HOF]	[2^{0}..2^{0}..2^{3}] 3	[2^{2}..2^{3}] 5	0	93	[2^{0}..2^{5}..2^{14}] 241	[2^{2}..2^{3}] 2	[2^{6}..2^{6}..2^{8}] 11	[2^{0}] 1
CVE-2018-8964	UAF[HOF]	[2^{3}] 1	[2^{2}..2^{3}] 5	0	45	[2^{0}..2^{8}] 9	[2^{2}..2^{3}] 2	[2^{5}..2^{5}] 7	[2^{0}] 1
CVE-2018-7871	HOF[#W|UAF|#N]	[2^{0}..2^{3}] 4	[2^{2}..2^{3}] 6	0	408	[2^{0}..2^{14}] 216	[2^{2}..2^{18}] 13	[2^{3}..2^{10}] 54	[2^{0}] 1
CVE-2021-45078	HOF[-]	-	[2^{3}] 1	0	1	-	[2^{4}] 1	-	[2^{0}] 1
CVE-2021-3156	HOF[-]	-	[2^{0}..2^{10}] 694	0	2	-	[2^{2}..2^{4}..2^{5}] 31	-	[2^{0}..2^{10}] 2
CVE-2021-20294	SOF[-]	-	[2^{9}..2^{9}] 2	1	1	-	[2^{0}] 1	-	[2^{8}..2^{8}] 1
CVE-2021-20284	HOF[UAF|#N]	[2^{0}..2^{3}] 3	-	0	19	[2^{0}..2^{12}] 841	-	[2^{0}..2^{5}] 11	-
CVE-2020-35493	HOF[-]	[2^{0}] 1	-	0	20	[2^{2}..2^{13}] 2072	-	[2^{0}] 1	-
CVE-2020-16592	UAF[-]	[2^{3}..2^{6}] 95	-	0	1	[2^{2}..2^{7}] 200	-	[2^{2}..2^{7}] 181	-
CVE-2020-21050	SOF[HOF]	-	[2^{0}..2^{1}] 2	1	2	-	[2^{5}..2^{27}] 11	-	[2^{0}..2^{14}] 2
CVE-2019-20094	HOF[-]	-	[2^{0}] 1	0	1	-	[2^{11}..2^{25}..2^{26}] 2655	-	[2^{0}] 1
CVE-2019-20024	HOF[-]	-	[2^{9}] 1	0	1	-	[2^{26}] 1	-	[2^{0}] 1
CVE-2021-26926	HOF[-]	[2^{0}] 1	-	0	1	[2^{2}] 1	-	[2^{0}] 1	-
CVE-2021-3272	HOF[-]	[2^{3}] 1	-	0	1	[2^{3}] 1	-	[2^{0}] 1	-
CVE-2020-27828	HOF[#N]	[2^{0}] 1	[2^{3}] 1	0	1	-	[2^{9}] 1	-	-
CVE-2018-19543	HOF[UAF|#N]	[2^{3}] 1	-	0	1	[2^{0}] 1	-	[2^{0}] 1	-
CVE-2018-19540	HOF[-]	-	[2^{0}] 1	0	1	-	[2^{0}] 1	-	[2^{0}] 1
CVE-2021-3246	HOF[-]	-	[2^{1}] 1	0	2	-	[2^{8}..2^{12}..2^{15}] 5640	-	[2^{0}] 1
CVE-2020-24241	UAF[-]	[2^{0}..2^{3}] 6	-	0	1	[2^{5}..2^{6}] 6	-	[2^{5}] 1	-
CVE-2020-21676	SOF[GOF]	[2^{0}] 1	-	0	1	[2^{6}] 1	-	[2^{0}] 1	-
CVE-2020-21675	GOF[SOF]	[2^{0}] 1	[2^{0}..2^{6}] 105	2	2	[2^{6}] 1	-	[2^{0}] 1	[2^{11}..2^{14}] 2

the number of capabilities found by EVOCATIO steadily increases. For EVOCATIO, the number of seeds equals the number of capabilities; each newly-saved seed corresponds to a new capability. CAPFUZZ continues to discover new capabilities, reaching a total number that is generally orders-of-magnitudes larger than the number found by `afl-cexp`. Section 6.5 explains the advantage of EVOCATIO over `afl-cexp` when it comes changes in bug type.

6.2 Design Validation (RQ2)

We perform an ablation study to measure the contributions of CAPFUZZ's core components (critical byte inference and seed selection). We randomly select four CVEs and use the three configurations to measure these contributions: (i) *retention only*: the dual-phase critical bytes inference is disabled, as a consequence all other elements but the seed retention strategy are also removed; (ii) *critical bytes only*: the most energetic sequence is removed. This implies disabling the seed selection and the energy scheduling strategies. Additionally, the mutation strategy will randomly mutate critical bytes instead of prioritizing them by their contribution; and (iii) *full design*: all design elements are enabled.

Figure 3 summarizes the capability discovery rate across these three configurations. On average, *critical bytes only* improves performance over *retention only*. While this improvement is small for



(a) CVE-2021-20284. (b) CVE-2021-3272. (c) CVE-2018-9009. (d) CVE-2018-19540.

(e) CVE-2018-20591. (f) CVE-2019-9114. (g) CVE-2021-45078. (h) CVE-2020-16592.

Figure 2: Capabilities discovered over time by CapFuzz and `afl-cexp` (for `afl-cexp` the seeds are also reported). Colors indicate the number of capabilities discovered by: CapFuzz; `afl-cexp`; and the number of seeds generated by `afl-cexp` (dashed line).

CVE-2020-21675, it is significant for the other three CVEs, ranging from a factor of 2× (CVE-2018-7871) to 10× (CVE-2018-12900). Adding the most energetic sequence in addition to the critical byte inference further improves performance. Indeed, the *full design* performs twice as good as the *critical bytes only* configuration (on average). These results confirm the importance of the two main design elements of CapFuzz: critical byte inference and most energetic sequences.

6.3 Bug Severity Assessment (RQ3)

We investigate whether the capabilities discovered by Evocatio assist developers in assessing bug severity. As described in Section 4.1, the severity of a bug is based on a specific threat model. Here we use the threat model presented in Section 4.1.1 to *quantitatively* derive a severity score.

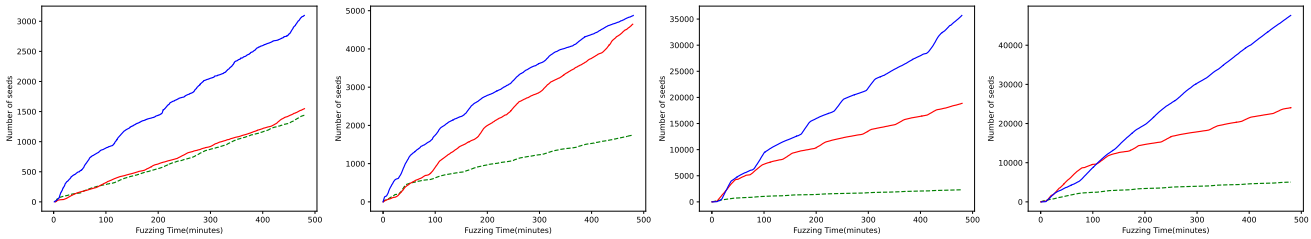
We compare our scoring system against CVSS; specifically, the *impact* and *base* metrics. The impact metric "*reflects the direct consequence of a successful exploit*" [20], while the base metric "*represents the intrinsic characteristics of a vulnerability that are consistent over time and across user environments*" [20]. The impact metric also contributes to the base metric, providing a comprehensive representation of the bug [54]. Both scores range between zero and ten (with a higher score corresponding to higher severity). We select the impact metric because it is conceptually closer to our scoring system. In contrast, the base metric provides a broader perspective that also depends on non-quantifiable metrics (e.g., the context in which a technology is deployed). Importantly, both metrics require a thorough analysis of the bug and likely exploit, often resulting in substantial manual efforts. In contrast, Evocatio is fully automatic.

Table 4 shows the scores assigned by the different metrics. Scores are highlighted according to CVSS risk levels: low (0.1–3.9), medium (4.0–6.9), high (7.0–8.9), and critical (9.0–10.0). Evocatio and the CVSS base metric place ∼50 % of the bugs in the same risk level. The remaining bugs are typically only one level lower in severity. In contrast, Evocatio "underestimates" the severity of 17 bugs when compared to CVSS's impact metric. We use our results in Table 2 as a proxy to validate our findings: the range of capabilities exposed by the original PoC were fewer than those discovered by CapFuzz, potentially biasing the original CVSS analysis. We discuss some of these results below.

CVE-2018-17795. Evocatio ranks this bug as medium risk (score = 4.9). In contrast, the CVSS base metric ranks the bug as high risk (score = 8.8). The description provided by CVSS for this bug states that it "*possibly has other unspecified impact*" [52]. This vague message does not provide an indication of the security impact of the bug. Per Table 2, the bug causes a single byte write with limited address flexibility. CVSS assigned a high score because of an unspecified possible additional impact; Evocatio found no such impact. This bug is an example of overestimation due to imprecise knowledge.

CVE-2021-3246. This bug presents another score difference between the Evocatio and CVSS base metrics: 4.9 and 8.8, respectively. According to the CVSS report, this bug is high risk because it effects a network service and requires no privileges. Our system is purely based on capabilities and does not take into account attack surfaces nor required privileges. Per Evocatio, this bug grants a single-byte write with little address flexibility (Table 2).

CVE-2019-16705. The CVSS base metric considers this bug critical (scoring it 9.1), while the impact metric assigns a medium score (5.3). In contrast, Evocatio ranks this bug as high risk. Here, Evocatio would help an analyst assign a more representative impact



(a) CVE-2020-21675. (b) CVE-2018-7871. (c) CVE-2018-12900. (d) Libtiff-issue-278.

Figure 3: Comparison of different CapFuzz strategies (one seed represents one new capability). Colors indicate different strategies: retention only; critical bytes only; and the full design.

Table 3: Capabilities discovered in eight hours by CapFuzz and `afl-cexp`. For `afl-cexp`, the number of seeds are also provided. "Inc. Rate" represents the capability increment rate, computed as (CapFuzz − `afl-cexp`)/`afl-cexp`.

Bug	Capabilities (#)		Seeds (#)	Inc. Rate
	CapFuzz	`afl-cexp`	`afl-cexp`	
issue-275	45,825	192	775	37.7
issue-277	1,855	1	1,224	1,854
issue-278	37,689	1	3,051	37,688
issue-269	779	86	3,153	8.1
CVE-2018-17795	59	9	1,252	5.6
CVE-2018-12900	60,188	1	1,154	60,187
CVE-2018-8905	16	1	990	15
CVE-2016-10272	42,454	187	5,178	227.0
CVE-2016-10092	17,004	330	4,726	50.5
CVE-2016-9532	73,209	440	14,900	165.4
CVE-2016-9273	12	9	1,044	0.3
CVE-2020-11895	13,672	19	350	718.6
CVE-2020-11894	4,645	24	3,650	193.5
CVE-2020-6628	1,692	119	2,921	13.2
CVE-2019-16705	1,785	20	336	88.3
CVE-2019-9114	963	29	2,684	32.2
CVE-2018-20591	2,966	98	3,313	29.3
CVE-2018-9009	1,269	74	2,988	16.1
CVE-2018-8964	946	177	3,236	4.3
CVE-2018-7871	17,561	194	3,652	89.5
CVE-2021-45078	5	5	553	0
CVE-2021-3156	2352	1	478	2351
CVE-2021-20294	7,327	21	428	347.9
CVE-2021-20284	2,552	24	659	105.3
CVE-2020-35493	3,411	5	186	681.2
CVE-2020-16592	10,728	41	155	260.7
CVE-2020-21050	10	0	27	+∞
CVE-2019-20094	2,176	1	11	2,175
CVE-2019-20024	0	0	105	0
CVE-2021-26926	0	0	734	0
CVE-2021-3272	152	2	1,223	75
CVE-2020-27828	1	0	596	+∞
CVE-2018-19543	0	0	1,338	0
CVE-2018-19540	172	16	1,449	9.8
CVE-2021-3246	5,602	0	102	+∞
CVE-2020-24241	78	28	1,820	1.8
CVE-2020-21676	1	1	482	0
CVE-2020-21675	2,534	15	396	167.9

score. Interestingly, both this and CVE-2020-6628 are the only bugs Evocatio considers as high risk for both read and write scores. From this result (and others in Table 4), we observe the following trend: Bugs with high read and write scores have higher CVSS scores. In general, if both scores are above 6, then the bug is likely to be considered high risk by CVSS. Our scores are two dimensional; when considered together Evocatio's scores a good approximation of the CVSS score.

Finally, Evocatio ranks CVE-2016-9532 and CVE-2018-20591 higher than CVSS's base and impact metrics. This is because Evocatio discovered new bug types that go beyond those originally reported in the CVSS base and impact metrics (Table 2), resulting in an underestimation of severity.

To summarize, Evocatio's metrics are automatic and provide a fast estimate of a bug's impact, along with a score to enable comparisons between bugs. While there are variations between our scores and those reported by CVSS, our scores are quick to compute and can support human analysis with more concrete evidence. Our scores are also oblivious to how the software is run and how much effort is invested into exploitation. For example, a vulnerability where an analyst invests hundreds of hours to demonstrate its exploitability may receive a higher CVSS score than a similarly-severe bug discovered and automatically reported by `syzkaller`. Instead, our scores, provide a quantitative estimate of severity, precisely capturing the read and write powers of an attacker. This focus on quantitative results prevents both over-estimation (e.g., CVE-2018-17795) and under-estimation (e.g., CVE-2016-9536, CVE-2018-20591) of severity. Analysts may leverage Evocatio as a fast and automatic reporter of bug capabilities, using the results to enrich their understanding of a bug.

6.4 Patch Violation (RQ4)

Our results demonstrate Evocatio's ability to quickly and efficiently discover new capabilities. *Can this new knowledge about a bug be used to test the efficacy of bug fixes/patches*?

To answer this question, we selected the 16 bugs from our benchmark where we could identify a single commit that (supposedly) fixed the given bug (and modified no other code). We applied these patches and replayed all Evocatio-generated and `afl-cexp`-generated PoCs to test the efficacy of each patch.

Table 5 summarizes the results of this process for Evocatio: seven of the 16 patches (44 %) were violated (i.e., a crash occurred) by a Evocatio-generated PoC. This is despite the bug being "fixed". To verify this result, we replayed the original PoCs through the patched program; none of these PoCs violated the patch. In other words, 44 % *of these bugs were incorrectly fixed.*

Table 4: Evocatio read and write severity scores, and CVSS's impact and base scores. Colors indicate risk level: low (green), medium (yellow), high (orange), and critical (red).

Bug	Evocatio		CVSS	
	Read	Write	Impact	Base
issue-269	6.3	3.1	–	–
issue-275	7.3	6.8	–	–
issue-277	5.1	5.1	–	–
issue-278	6.2	6.2	–	–
CVE-2016-9273	6.5	5.5	3.6	5.5
CVE-2016-9532	7.3	6.1	3.6	5.5
CVE-2016-10092	6.2	7.2	5.9	7.8
CVE-2016-10272	5.6	5.7	5.9	7.8
CVE-2018-7871	7.1	6.5	5.9	8.8
CVE-2018-8905	0	4.8	5.9	8.8
CVE-2018-8964	6.7	6.1	3.6	6.5
CVE-2018-9009	6.9	6.3	5.9	8.8
CVE-2018-12900	5.9	5.3	5.9	8.8
CVE-2018-17795	0	4.9	5.9	8.8
CVE-2018-19540	0	4.2	5.9	8.8
CVE-2018-19543	3.6	0	5.9	7.8
CVE-2018-20591	7.5	6.3	3.6	6.5
CVE-2019-20024	0	4.7	3.6	6.5
CVE-2019-20094	0	4.8	5.9	8.8
CVE-2019-16705	7.5	7.0	5.2	9.1
CVE-2019-9114	0	5.7	5.9	8.8
CVE-2020-6628	7.0	7.0	5.9	8.8
CVE-2020-11894	7.0	6.4	5.2	9.1
CVE-2020-11895	7.1	6.5	5.2	9.1
CVE-2020-16592	6.2	0	3.6	5.5
CVE-2020-21050	0	6.5	3.6	6.5
CVE-2020-21675	6.2	6.5	3.6	5.5
CVE-2020-21676	4.8	0	3.6	5.5
CVE-2020-24241	5.1	0	5.2	5.5
CVE-2020-27828	0	4.3	5.9	7.8
CVE-2020-35493	5.6	0	3.6	5.5
CVE-2021-3156	0	7.2	5.9	7.8
CVE-2021-3246	0	4.9	5.9	8.8
CVE-2021-3272	5.1	0	3.6	5.5
CVE-2021-20284	6.9	0	3.6	5.5
CVE-2021-20294	0	5.0	5.9	7.8
CVE-2021-26926	3.8	0	5.2	7.1
CVE-2021-45078	0	5.0	5.9	7.8

We further investigated the efficacy of these patches by replaying the Evocatio-generated crash-inducing PoCs through the latest version of the given program. We found one case (CVE-2018-7871) where a crash still triggered due to other changes in the code. While the bug was believed to have been fixed in March 2018, in reality the bug remained unfixed for almost 4 years. This result confirms the need for tools like Evocatio. With the information provided by our system, the developers could have implemented a more complete patch. We have reported the incomplete fix to the developers and are carefully checking the other partial patches. At the time of submission, all of the incomplete patches—except for Libming, which is no longer maintained—have been fixed in newer versions of the code.

Only a single patch violation was found among the `afl-cexp`-generated PoCs (CVE-2018-7871). This patch was also violated by Evocatio. For the other six patches Evocatio successfully violated (but `afl-cexp` failed to violate), we found that Evocatio discovered new bug types and access types that `afl-cexp` was unable to discover. We discuss the importance of this capability in Section 6.5.

Table 5: Leveraging CapFuzz to violate existing patches. For each patch a reference is official patch link is provided. Correct patches are marked with ✓, incomplete patches with ✗.

Program	Bug	Fix details	Complete
Libtiff	CVE-2018-12900 [6]	Limit variable size	✓
Libtiff	CVE-2016-10272 [57]	Increase buffer size	✓
Libtiff	CVE-2016-10092 [56]	Increase buffer size	✗
Libtiff	CVE-2016-9532 [59]	Add boundary check	✗
Libtiff	CVE-2016-9273 [58]	Recompute buffer size	✗
Libming	CVE-2018-9009 [37]	Add function to check var.	✓
Libming	CVE-2018-8964 [36]	Add check before using var.	✗
Libming	CVE-2018-7871 [38]	Add boundary check	✗
Binutils	CVE-2021-45078 [49]	Change var. to unsigned	✓
Binutils	CVE-2021-20294 [48]	Disable `sprintf` to buffer	✓
Binutils	CVE-2021-20284 [15]	Add check before using var.	✓
Libsixel	CVE-2020-21050 [62]	Add boundary check	✗
Jasper	CVE-2020-27828 [77]	Add boundary check	✗
Jasper	CVE-2018-19543 [32]	Change boundary check	✓
Jasper	CVE-2018-19540 [31]	Add boundary check	✓
Fig2dev	CVE-2020-21675 [42]	Increase buffer size	✓

Listing 3: CVE-2018-7871 code snippet.

```
1  char *getName(struct SWF_ACTIONPUSHPARAM *act) {
2    switch (act->Type) {
3    case PUSH_CONSTANT:
4      t = malloc(strlenext(pool[act->p.Constant8])+1);
5      strcpyext(t, pool[act->p.Constant8]);
6      // ...
7    case PUSH_CONSTANT16:
8      t = malloc(strlenext(pool[act->p.Constant16])+1);
9      strcpyext(t, pool[act->p.Constant16]);
10     // ...
11   }
12   return t;
13 }
```

6.5 Case Study: Escalating Bug Types

Evocatio found new bug types for 18 out of 38 evaluated CVEs. Moreover, Evocatio converted 10 out of 38 CVEs from a read primitive into a write primitive. Evocatio can do this because of its ability to distinguish different capabilities (via CapSan). In contrast, `afl-cexp` is limited to knowledge gleaned from the fuzzer's coverage map. Here we showcase two case studies demonstrating the power of these abilities.

6.5.1 CVE-2018-7871 (Libming). Illustrated in Listing 3, a heap buffer overflow occurs when `act->p`'s `Constant8` or `Constant16` is greater than the size of `pool` (Lines 4 and 8, respectively). This overflow is due to an incorrect check of the relationship between `act->p.Constant8`, `act->p.Constant16`, and `pool`'s size. Importantly, `act`'s `Type`, `Constant8` and `Constant16` are all attacker controlled. Once overflowed, the `pool` pointer will point to arbitrary memory.

The exact memory location that is accessed when dereferencing `pool` depends on a set of critical input bytes (leading to different values in `act`'s `Constant8`, `Constant16`, etc. fields). We distinguish four possible cases (and their effects): (i) a `null`-pointer dereference; (ii) the address of a free chunk triggers a UAF; (iii) a few bytes after the end of a heap chunk triggers a heap buffer overflow (invalid out-of-bounds read); and (iv) invalid memory access triggers a segmentation fault.

Listing 4: CVE-2016-9273 code snippet.

```
static int
TIFFVGetField(TIFF *tif, uint32 tag, va_list ap){
  if (fip->field_passcount) {
    if (fip->field_readcount == TIFF_VARIABLE2)
      // Stack-buffer-overflow write
      *va_arg(ap, uint32*) == (uint32)tv->count;
    else
      *va_arg(ap, uint16*) == (uint16)tv->count;
    // Heap-buffer-overflow write
    *va_arg(ap, void **) = tv->value;
  else
    tstrip_t s, ns = TIFFNumberOfStrips(in);
    uint64 *bytecounts;
    TIFFVGetField(in, tag, &bytecounts);
    for (s = 0; s < ns; s++) {
      // Heap-buffer-overflow read
      if (bytecount[s] > (uint64)bufsize)
        bufsize = (tmsize_t)bytecounts[s];
    }
  }
  // ...
}

int main(){
  uint32 tag = get_tag_from_file(in);
  va_list ap;
  va_start(ap, tag);
  // Invoke TIFFNumberOfStrips to initialize tif
  TIFFVSetField(tif, tag, ap);
  va_end(ap);

  TIFFVGetField(tif, tag, ap);
}
```

EvoCatio successfully uses its critical bytes inference process to efficiently discover these four cases, while CapSan successfully differentiates between the different bug types. In contrast, `afl-cexp` only uncovers the UAF case (by tracing a new path).

6.5.2 CVE-2016-9273 (Libtiff). The root cause of this bug (Listing 4 [7]) lies in `TIFFNumberOfStrips` (a parsing routine), where file content is transformed into a `TIFF` structure [60]. This function returns an incorrect value used that is subsequently used to calculate the buffer size of a structure member. If the program calls `TIFFNumberOfStrips` in different locations, a wrong buffer size will be allocated to different objects, resulting in different overflow effects: if the program tries to read from the victim buffer, an invalid read will be reported; if the program tries to write content to the victim buffer, an invalid write will be reported.

During critical byte inference CapFuzz discovered modifications to one byte in the original PoC that introduces different capabilities. According to our investigation, this byte determines which TIFF structure will be initialized by `TIFFNumberOfStrips` . This single byte causes the program to call `TIFFNumberOfStrips` from different contexts, leading to an incorrectly-sized buffer (a) allocated at different locations (heap or stack) and (b) accessed in different ways (reads or writes).

The original PoC calls `TIFFNumberOfStrips` to calculate the buffer size of a stack-allocated object. The allocated object is later dereferenced, leading to a heap overflow and an out-of-bounds read. CapFuzz discovered the critical byte and commences targeted mutation. This causes the program's processing logic to enter different branches, generating new inputs allocated (with different sizes) on the stack or heap (the buffer will be written to memory later). This eventually leads to a stack/heap buffer overflow through an out-of-bounds write. In our evaluation it took EvoCatio two minutes to discover new bug types and new access types.

[7] Modeled after CVE-2016-273; we have simplified the logic and omitted variable initialization.

6.6 Case Study: sudo

CVE-2021-3156 is a high-severity heap-based buffer overflow vulnerability in the sudo program. The NVD bug description states "`sudo` *before 1.9.5p2 contains an off-by-one error that can result in a heap-based buffer overflow, which allows privilege escalation to root via* `sudoedit -s` *and a command-line argument that ends with a single backslash character*" [53]. We use this vulnerability to demonstrate how EvoCatio (a) discovers additional capabilities from a seemingly low-impact PoC, and (b) accurately predicts the severity of the vulnerability.

By fuzzing a modified `sudo` [41] with AFL++, we obtain a PoC [40] exercising a 1-byte heap buffer overflow (reported by ASan). EvoCatio expands this single PoC to more than 200 new PoCs within 10 minutes. These PoCs result in OOB writes of varying lengths (the largest of which is 120 bytes). Per Section 6.3, EvoCatio scores this bug as high risk. In comparison, AFL++'s crash mode is unable to generate any new PoCs even after 8 hours of fuzzing.[8]

7 DISCUSSION

Here we discuss EvoCatio's limitations and how they can be addressed by future work. Importantly, the goal of EvoCatio is not to replace CVSS. CVSS requires manual analysis, which EvoCatio assists by focusing on "interesting" bugs (based on discovered capabilities). This, along with CVSS, helps decide bug severity.

Critical bytes (identified during the inference phase) are relative to the input length. CapFuzz does not change the input length while fuzzing to avoid breaking these bytes. Consequently, EvoCatio does not support the exploration of capabilities related to input length.

In addition to the bug types supported in this paper, common memory safety bug types include `null`-pointer dereferences and double frees. However, these two bug types do not directly lead to memory corruption, and thus EvoCatio is unable to discover capabilities associated with these bug types. We leave the detection of other bug types as future work.

The scoring system described in Section 4.1 is based on a threat model where the number and diversity of capabilities is the most important measure of "severity". However, more capabilities may not always correlate with increased severity; even an off-by-one error can lead to the compromise of a system. EvoCatio can reason about off-by-one errors provided they lead to a memory safety violation (i.e., they are detectable by CapSan); we demonstrate this in Section 6.6. However, generalizing these bug types (i.e., when they do not result in memory corruption) requires changes to capability detection, which we leave to future work.

8 RELATED WORK

Symbolic tracing engines collect symbolic constraints along a program's execution path. These constraints can then be analyzed to

[8] Section 3.1.1 contains further discussion on why crash mode fails.

infer all possible values of a variable at any point in time [23, 71, 72]. In theory, symbolic tracing enables both the identification of critical bytes and bug capabilities. However, in practice, symbolic tracing suffers from scalability and accuracy issues, making it difficult to apply to real-world applications [1, 4].

Taint analysis tracks the flow of data in a target program from a "taint source" (e.g., a user input function) to a "taint sink" (e.g., a security-critical function). This allows the relationships between input bytes to be determined as taint propagates through the program. Taint analysis has been used to locate: buffer boundary violations [27]; buffer over-read vulnerabilities [50]; and invocations of sensitive library and system calls [22]. It has also been used to improve fuzzer performance [11, 55]. Unfortunately, taint analysis is subject to under- and over-tainting [14, 75]. While solutions to under/over tainting have been proposed, they require manual effort and result in performance degradation [33, 69]. Under/over tainting is particularly pronounced in large, real-world programs, making taint analysis difficult to use for critical byte identification.

Fuzzing-based taint inference has been used to overcome the previously-discussed limitations of taint analysis [2, 21, 39, 45, 76]. Indeed, CapFuzz's byte inference technique is similar to the one proposed by GreyOne [21]. However, there are a number of important distinctions. First, our goals differ: CapFuzz is focused on capabilities, not bugs. Second, GreyOne does not consider *relationships* between bytes (as discussed in Section 3.2.1).

AFL++'s crash exploration mode (`afl-cexp`) has similar goals to CapFuzz: given a PoC, explore the crash's state space (via fuzzing) and generate new PoCs that induce the same crash. However, `afl-cexp` is guided by code-coverage and is unable to detect changes in data flow. In contrast, CapFuzz not only detects data flow changes, but focuses on inducing these data flow changes. Our evaluation (Section 6.1.1) shows this has a significant impact on CapFuzz's ability to discover new capabilities.

Patch validation and testing aims to ensure a patch fixes a given bug while not introducing any new bugs (patch testing [8]) and that the program's behavior remains correct (patch validation [10, 24]). In contrast, Evocatio aims at producing a set of PoCs that trigger nontrivial behaviors in unpatched programs, helping developers build correct patches. We demonstrate how Evocatio assists this process in Section 6.4.

Automatic Exploit Generation (AEG) shares similar goals to Evocatio. For example, Revery [71] uses fuzzing to extend a PoC's capabilities, but focuses on relatively strong capabilities (e.g., hijacking control flow), ignoring diversity. Evocatio's goals differ from AEG engines such as Revery—where finding a single exploitable capability is sufficient: Evocatio aims to collect a large set of diverse capabilities.[9] Similarly, Gollum [30] and MAZE [72] are heap-manipulating AEG engines. Gollum leverages fuzzing and requires manually-specified templates describing the target's input syntax, while MAZE uses symbolic execution to solve complex constraints. While automated heap manipulation enables AEG, we target more general, "lower-level" capabilities. In summary, Evocatio explores bug capabilities, while AEG builds exploits. While severe vulnerabilities may facilitate an exploit, their development requires knowledge about the program/environment; this is beyond our goals.

Bug severity assessment systems commonly make use of qualitative metrics (e.g., attack complexity) [64, 67, 68] extracted from vulnerability descriptions to rank bugs. Prior work has leveraged machine learning—which is more amenable to qualitative metrics—to predict bug severity [18, 28, 34]. These systems use vulnerability descriptions and reports, rather than program analyses, to determine bug severity. In contrast, Evocatio's focus is not to use known information to determine severity, but to provide a framework for reasoning about severity based on the set of capabilities discovered by Evocatio. For example, capabilities can be used in a human-in-the-loop scenario to generate a CVSS-style severity score, similar to our sudo case study (Section 6.6).

Kernel-space fuzzing shares similar design constraints to user-space fuzzing. In particular, a bug is represented by a single PoC that may only exercise a limited set of capabilities, resulting in a limited understanding of the bug. SyzScope [78] combines fuzzing, static analysis, and symbolic execution to analyze the security impact of fuzzer-exposed bugs in the kernel. CapFuzz focuses on user-space programs, which are not supported by SyzScope. Moreover, due to its dependency on symbolic execution, SyzScope will inevitably suffer from state explosion and complex constraint solving, potentially leading to poor performance on larger kernel modules.

9 CONCLUSIONS

Developers are overwhelmed by the large number of discovered bugs. In order to prioritize their fixes, developers must understand the *capabilities* these bugs expose. By understanding these capabilities, a more accurate assessment of a bug's impact can be performed. We developed Evocatio for this purpose. Evocatio uses a novel capability-guided fuzzer, which, given an initial PoC, explores the surrounding state space for new capabilities. In our evaluation, Evocatio outperformed existing crash exploration tools (in particular, AFL++'s crash exploration mode). We demonstrated Evocatio's utility by applying it to automatic bug severity assessment and patch testing. Notably, Evocatio was able to generate PoCs that violate existing patches.

Our open-source release of the Evocatio prototype implementation is available at https://github.com/HexHive/Evocatio.

ACKNOWLEDGEMENTS

We thank our shepherd Marcel Böhme, the anonymous reviewers, and Ruilin Li for their careful feedback which greatly improved the clarity of this paper. This project has received funding from the European Research Council (ERC) under the H2020 grant 850868, SNSF PCEGP2_186974, AFRL FA8655-20-1-7048, DARPA HR001119S0089-AMP-FP-034, the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (61972224), Beijing National Research Center for Information Science and Technology (BNRist) under Grant BNR2022RC01006, China Postdoctoral Science Foundation 2021M701942, and the National Science Foundation (NSF) under Grant CNS1942793. Any findings are those of the authors and do not necessarily reflect the views of our sponsors.

[9] We contacted the authors of Revery to facilitate a side-by-side comparison. They confirmed our evaluated programs are too complex for Revery.

REFERENCES

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-driven compositional symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 367–381.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence.. In *Network and Distributed Systems Security (NDSS, Vol. 19)*. 1–15.

[3] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic Exploit Generation. *Commun. ACM* 57, 2 (feb 2014), 74–84. https://doi.org/10.1145/2560217.2560219

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3, Article 50 (may 2018), 39 pages. https://doi.org/10.1145/3182657

[5] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. 2013. The Page-Fault Weird Machine: Lessons in Instruction-less Computation. In *Workshop on Offensive Technologies (WOOT)*. USENIX.

[6] Thomas Bernard. 2018. CVE-2018-12900 Patch. https://gitlab.com/libtiff/libtiff/-/merge_requests/60.

[7] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. 2020. AURORA: Statistical Crash Analysis for Automated Root Cause Explanation. In *USENIX Security (SEC)*. USENIX, 235–252.

[8] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Computer and Communications Security (CCS)*. ACM, 2329–2344. https://doi.org/10.1145/3133956.3134020

[9] Rich Campagna. 2020. The 3 Reasons CVSS Scores Change Over Time. https://securityboulevard.com/2020/05/the-3-reasons-cvss-scores-change-over-time/.

[10] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and precise on-the-fly patch validation for all. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1123–1134.

[11] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *Security and Privacy (S&P)*. IEEE, 711–725.

[12] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities. In *USENIX Security (SEC)*. USENIX, 1093–1110.

[13] Yueqi Chen and Xinyu Xing. 2019. Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1707–1722.

[14] Zheng Leong Chua, Yanhao Wang, Teodora Baluta, Prateek Saxena, Zhenkai Liang, and Purui Su. 2019. One Engine To Serve 'em All: Inferring Taint Rules Without Architectural Semantics. In *Network and Distributed Systems Security (NDSS)*.

[15] Nick Clifton. 2020. CVE-2021-20284 Patch. https://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git;h=f60742b2a1988d276c77d5c1011143f320d9b4cb.

[16] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (jul 2019), 62–70. https://doi.org/10.1145/3338112

[17] Thomas Dullien. 2020. Weird Machines, Exploitability, and Provable Unexploitability. *Transactions on Emerging Topics in Computing* 8, 2 (2020), 391–403. https://doi.org/10.1109/TETC.2017.2785299

[18] Clément Elbaz, Louis Rilling, and Christine Morin. 2020. Fighting N-day vulnerabilities with automated CVSS vector prediction at disclosure. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*. 1–10.

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *Workshop on Offensive Technologies (WOOT)*. USENIX.

[20] FIRST. 2019. Common Vulnerability Scoring System v3.1: Specification Document. https://www.first.org/cvss/specification-document.

[21] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *USENIX Security (SEC)*. USENIX, 2577–2594.

[22] Vijay Ganesh, Tim Leek, and Martin Rinard. 2009. Taint-based directed whitebox fuzzing. In *International Conference on Software Engineering (ICSE)*. IEEE, 474–484.

[23] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *Transactions on Software Engineering and Methodology* 30, 2 (2021), 1–27.

[24] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[25] Google. 2019. Default configuration of AddressSanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer.

[26] Google. 2022. Google syzbot. https://syzkaller.appspot.com/upstream.

[27] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for overflows: a guided fuzzer to find buffer boundary violations.. In *USENIX Security (SEC)*. USENIX, 49–64.

[28] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. 2017. Learning to predict severity of software vulnerability using only vulnerability description. In *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 125–136.

[29] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[30] Sean Heelan, Tom Melham, and Daniel Kroening. 2019. Gollum: Modular and greybox exploit generation for heap overflows in interpreters. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1689–1706.

[31] Max Kellermann. 2018. CVE-2018-19540 Patch. https://github.com/jasper-maint/jasper/pull/38.

[32] Max Kellermann. 2018. CVE-2018-19543 Patch. https://github.com/jasper-maint/jasper/pull/38/commits/69bba1480fb4b1f1e2ab75a14a00721f4cf16e50.

[33] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. 2012. Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Virtual Execution Environments (VEE)*. ACM, 121–132. https://doi.org/10.1145/2151024.2151042

[34] Atefeh Khazaei, Mohammad Ghasemzadeh, and Vali Derhami. 2016. An automatic method for CVSS score prediction using vulnerabilities description. *Journal of Intelligent & Fuzzy Systems* 30, 1 (2016), 89–96.

[35] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Computer and Communications Security (CCS)*. ACM, 2123–2138. https://doi.org/10.1145/3243734.3243804

[36] Hugo Lefeuvre. 2018. CVE-2018-8964 Patch. https://github.com/libming/libming/issues/128.

[37] Hugo Lefeuvre. 2018. CVE-2018-9009 Patch. https://github.com/libming/libming/pull/145/commits/835cdd0776456483466c6d640d251548e7d9dcdb.

[38] Hugo Lefeuvre. 2018. Patch of CVE-2018-7871. https://github.com/libming/libming/pull/125/commits.

[39] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Automated Software Engineering (ASE)*. ACM, 475–485.

[40] LiveOverflow. 2021. Avaiable PoC to trigger the CVE-2021-3156. https://github.com/LiveOverflow/pwnedit/tree/main/episode05.

[41] LiveOverflow. 2021. Modification on sudo to enable it can be fuzzed by AFLplusplus. https://github.com/LiveOverflow/pwnedit/tree/main/episode01.

[42] Thomas Loimer. 2020. CVE-2020-21675 Patch. https://sourceforge.net/p/mcj/tickets/78/.

[43] Steve Mancini. 2020. The subjective nature of a CVSS score. https://eclypsium.com/2020/09/30/the-subjective-nature-of-a-cvss-score/.

[44] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[45] Björn Mathis, Vitalii Avdiienko, Ezekiel O. Soremekun, Marcel Böhme, and Andreas Zeller. 2017. Detecting Information Flow by Mutating Input Data. In *Automated Software Engineering (ASE)*. IEEE, 263–273.

[46] Xianya Mi, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2021. *LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating*. ACM, 62–77.

[47] Ghassan Misherghi and Zhendong Su. 2006. HDD: Hierarchical Delta Debugging. In *International Conference on Software Engineering (ICSE)*. ACM, 142–151. https://doi.org/10.1145/1134285.1134307

[48] Alan Modra. 2020. CVE-2021-20294 Patch. https://sourceware.org/bugzilla/show_bug.cgi?id=26929.

[49] Alan Modra. 2021. Patch of CVE-2021-45078. https://sourceware.org/bugzilla/show_bug.cgi?id=28694.

[50] Matthias Neugschwandtner, Paolo Milani Comparetti, Istvan Haller, and Herbert Bos. 2015. The BORG: Nanoprobing Binaries for Buffer Overreads. In *Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 87–97. https://doi.org/10.1145/2699026.2699098

[51] NIST. 2021. Common Vulnerability Scoring System. https://nvd.nist.gov/vuln-metrics/cvss.

[52] NVD. 2018. CVSS Description of CVE-2018-17795. https://nvd.nist.gov/vuln/detail/CVE-2018-17795.

[53] NVD. 2021. Description of CVE-2021-3156. https://nvd.nist.gov/vuln/detail/CVE-2021-3156.

[54] IBM QRadar. 2021. Common Vulnerability Scoring System (CVSS). https://www.ibm.com/docs/en/qradar-on-cloud?topic=vulnerabilities-common-vulnerability-scoring-system-cvss.

[55] Sanjay Rawat, Vivek Jain, Ashish Kumar, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed Systems Security (NDSS)*.

[56] Even Rouault. 2016. CVE-2016-10092 Patch. https://github.com/vadz/libtiff/commit/9657bbe3cdce4aaa90e07d50c1c70ae52da0ba6a.

[57] Even Rouault. 2016. CVE-2016-10272 Patch. https://github.com/vadz/libtiff/commit/9657bbe3cdce4aaa90e07d50c1c70ae52da0ba6a.

[58] Even Rouault. 2016. CVE-2016-9273 Patch. https://github.com/vadz/libtiff/commit/d651abc097d91fac57f33b5f9447d0a9183f58e7.

[59] Even Rouault. 2016. CVE-2016-9532 Patch. https://github.com/vadz/libtiff/commit/21d39de1002a5e69caa0574b2cc05d795d6fbfad.

[60] Even Rouault. 2016. Pull Request of CVE-2016-9273. https://github.com/vadz/libtiff/commit/d651abc097d91fac57f33b5f9447d0a9183f58e7.

[61] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (mar 2018), 58–66. https://doi.org/10.1145/3188720

[62] Hayaki Saito. 2018. CVE-2020-21050 Patch. https://github.com/saitoha/libsixel/commit/7808a06b88c11dbc502318cdd51fa374f8cd47ee.

[63] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Annual Technical Conference (ATC)*. USENIX, 28.

[64] Mustafizur R Shahid and Hervé Debar. 2021. CVSS-BERT: Explainable Natural Language Processing to Determine the Severity of a Computer Security Vulnerability from its Description. In *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 1600–1607.

[65] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. 2021. Localizing Vulnerabilities Statistically From One Exploit. In *Asia Computer and Communications Security (ASIA CCS)*. 537–549. https://doi.org/10.1145/3433210.3437528

[66] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints.. In *Network and Distributed Systems Security (NDSS)*.

[67] Georgios Spanos and Lefteris Angelis. 2018. A multi-target approach to estimate software vulnerability characteristics and severity scores. *Journal of Systems and Software* 146 (2018), 152–166.

[68] Georgios Spanos, Lefteris Angelis, and Dimitrios Toloudis. 2017. Assessment of vulnerability severity using text mining. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics*. 1–6.

[69] The Clang Team. 2022. DataFlowSanitizer Design Document. https://clang.llvm.org/docs/DataFlowSanitizerDesign.html.

[70] Guido Vranken. 2020. CVE-2020-1104. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-11104.

[71] Yan Wang, Chao Zhang, Xiaobo Xiang, Zixuan Zhao, Wenjie Li, Xiaorui Gong, Bingchang Liu, Kaixiang Chen, and Wei Zou. 2018. Revery: From Proof-of-Concept to Exploitable. In *Computer and Communications Security (CCS)*. ACM, 1914–1927. https://doi.org/10.1145/3243734.3243847

[72] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. 2021. MAZE: Towards Automated Heap Feng Shui. In *USENIX Security (SEC)*. USENIX, 1647–1664.

[73] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. 2019. {KEPLER}: Facilitating control-flow hijacking primitive evaluation for Linux kernel vulnerabilities. In *28th USENIX Security Symposium (USENIX Security 19)*. 1187–1204.

[74] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *USENIX Security (SEC)*. USENIX, 781–797.

[75] Babak Yadegari and Saumya Debray. 2014. Bit-level taint analysis. In *Source Code Analysis and Manipulation (SCAM)*. IEEE, 255–264.

[76] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *IEEE Security and Privacy (S&P)*. IEEE. https://doi.org/10.1109/SP.2019.00057

[77] Yuan. 2018. CVE-2020-27828 Patch. https://github.com/jasper-software/jasper/pull/253.

[78] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. 2022. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel. In *USENIX Security (SEC)*. USENIX, Boston, MA.