2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)

MEMFUZZ: Using Memory Accesses to Guide Fuzzing

Nicolas Coppik, Oliver Schwahn, Neeraj Suri
DEEDS Group, TU Darmstadt
Darmstadt, Germany
{nc, os, suri}@cs.tu-darmstadt.de

***Abstract*—Fuzzing is a form of random testing that is widely used for finding bugs and vulnerabilities. State of the art approaches commonly leverage information about the control flow of prior executions of the program under test to decide which inputs to mutate further. By relying solely on control flow information to characterize executions, such approaches may miss relevant differences. We propose augmenting evolutionary fuzzing by additionally leveraging information about memory accesses performed by the target program. The resulting approach can leverage more sophisticated information about the execution of the target program, enhancing the effectiveness of the evolutionary fuzzing. We implement our approach as a modification of the widely used AFL fuzzer and evaluate our implementation on three widely used target applications. We find distinct crashes from those detected by AFL for all three targets in our evaluation.**

I. INTRODUCTION

Fuzzing is an established form of random testing that has proven to be highly capable of finding bugs and vulnerabilities in numerous widely used programs and applications. It is commonly used to examine application software, libraries, as well as system level code and has been applied in practice to a wide variety of targets, ranging from smart contracts [1] to operating system kernels [2], [3]. In recent years, fuzzing has emerged as an effective technique for finding bugs that involve memory safety violations for software written in memory unsafe languages such as C or C++. This category of software bugs is of particular interest as such bugs frequently have security implications. Some of the most well known vulnerabilities of recent years belong to this category [4]–[6].

The state of the art for fuzzing is *coverage-guided, evolutionary fuzzing*. A typical fuzzing workflow is illustrated in Figure 1. It consists of the following steps:

a) A set of seed inputs is selected (for instance, from the test suite of the program under test) and used to initialize the input queue.
b) The program under test is instrumented, typically during compilation, to gather coverage information at runtime. This step is marked ① in Figure 1.
c) The fuzzer picks an input from the queue, mutates it, and runs the instrumented program under test with the mutated input while monitoring its execution.
d) The resulting coverage information is evaluated to determine whether the program under test exhibited *interesting* behavior with the latest input. Typically, this means checking whether the program crashed or new edges in the control flow graph or new basic blocks were reached during that execution. This step is marked ② in Figure 1. Non-crashing, interesting inputs are put back into the queue for further mutation, whereas crashing inputs are reported.

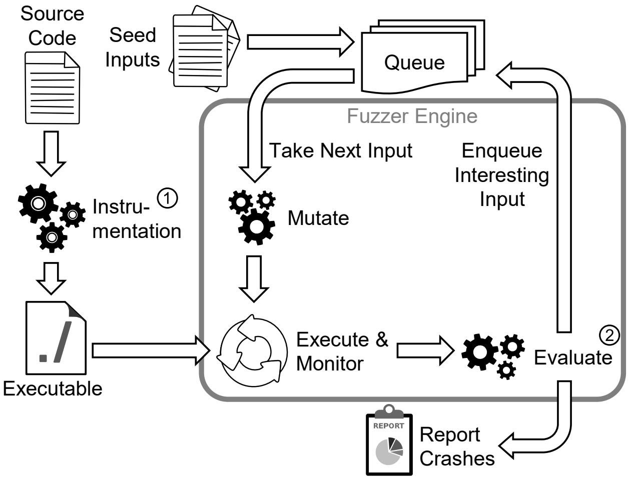


Fig. 1. Evolutionary Fuzzing

The last two steps of this process (grey box in Figure 1) are repeated as often as possible with the available time budget, which is set by the user. Fuzzing emphasizes speed of individual executions and favors efficiency over complex program analysis techniques. Therefore, the inserted instrumentation (①) should be lightweight and incur little runtime overhead; and the evaluation (②) should be fast to allow as many executions of the target program as possible in the given time budget.

For these reasons, widely used coverage-guided, evolutionary fuzzing techniques generally rely on basic block or edge coverage data: It can be gathered with relatively little runtime overhead and compact representations enable quick comparisons in the evaluation step.

Relying on such control-flow centered coverage information to distinguish between executions of a fuzzing target is an efficient approach with a proven track record. However, relying solely on edge or basic block coverage information may lead to missing important differences in how programs process

978-1-7281-1736-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ICST.2019.00015

```
 1  int main(int argc, char **argv) {
 2      int buffer[512] = {0};
 3      FILE* fp = fopen(argv[1], "r");
 4      int a = 0, b = 0, c = 0, d = 0;
 5      fscanf(fp, "(%d, %d); (%d, %d)",
 6              &a, &b, &c, &d);
 7      int x = (a * b) & 511;
 8      int y = (c * d) & 511;
 9      printf("%d %d %d\n", buffer[x],
10              buffer[y], buffer[x + y]);
11      return 0;
12  }
```

Fig. 2. A Motivating Example

different inputs. For instance, consider the example shown in Figure 2. The presented program reads four integers from an input file using `fscanf` (lines 3 – 6). Crucially, it does not perform any error checking and silently falls back to default values in the presence of invalid input. It then uses these integers to compute indices (lines 7 and 8) which are used to access a fixed-size buffer (lines 9 and 10). The last of these accesses (`buffer[x + y]` in line 10) is potentially out of bounds, which constitutes a memory safety violation bug.

This bug is straightforward to find by manual inspection and easy to detect using more heavyweight techniques like symbolic execution. Conventional coverage-guided, evolutionary fuzzing, however, struggles to find a crashing input for this program: There is only one path through the program, so the edge coverage such techniques rely on fails to provide meaningful feedback. Without any feedback, the resulting blind fuzzing is unlikely to produce a crashing input entirely by chance.

We fuzzed the program shown in Figure 2 using AFL 2.45b [7], a widely used, state of the art fuzzing tool, and a seed input `(1, 1); (1, 1)` for more than 24 hours without finding a crash. This is due to the fuzzer trimming the seed input prior to mutating it and, being entirely reliant on edge coverage, discarding most of it. Absent a working feedback mechanism, the fuzzer would then need to reconstruct the structure of the input entirely by chance to find a crash, which is highly unlikely.

Similar issues can arise due to the use of conditional moves or complex address calculations. Generally, the effectiveness of conventional, coverage-guided fuzzing is limited for code in which safe executions and unsafe, crashing executions exist with the same control flow, and in which the mutation of multiple parts of the input is necessary to generate an unsafe input from a safe one. In such cases, any intermediate inputs would be discarded by the fuzzer without further mutation as no new control flow behavior is exhibited, and the fuzzer would fail to find a crashing input.

We propose addressing this blind spot by using information about the memory addresses a program accesses, either in addition to or instead of edge coverage, to characterize executions. The resulting approach, which we call MEMFUZZ, should be able to distinguish between executions with the same control flow, based on differences in the accessed memory addresses. We implement a prototype of this approach based on AFL 2.45b. To summarize, our paper makes the following contributions:

- A new characterization of program executions for feedback-guided fuzzing
- MEMFUZZ, an instrumentation and static analysis approach to allow for efficient collection of memory coverage data
- A prototype implementation based on the state of the art AFL fuzzer
- An extensive evaluation of the proposed approach, highlighting both the strengths and limitations of the MEMFUZZ approach

The rest of this paper is structured as follows: We discuss related work in Section II. Our approach and implementation are presented in Section III and evaluated in Section IV. We discuss our results and threats to validity in Section V and present concluding remarks in Section VI.

II. RELATED WORK

Fuzzing is a wide, active research area, which renders a comprehensive overview infeasible. Therefore, we focus on a discussion of how our work relates to other research on evolutionary fuzzing and omit an in-depth discussion of other related techniques (e.g. approaches using symbolic execution).

Recent research in this area aims at improving the various steps of the evolutionary fuzzing process. We focus our discussion on related work dealing with seed selection and minimization, instrumentation, and guidance mechanisms in this section. Recent work on search strategies, scheduling, and queue prioritization [8]–[12] is largely orthogonal to our approach as it does not alter how executions are distinguished. Therefore, we omit a detailed discussion.

A. Seed Selection

The selection of seed inputs is usually one of the first steps in fuzzing workflows (cf. Figure 1) and one that requires substantial manual effort. Research in this area has dealt with automating this step, achieving effective seed corpora by optimizing seed selection [13] or generation [14]. As our proposed approach does not involve alterations to the seed selection step, most of this work is orthogonal to ours.

However, it is common to minimize seed test cases prior to fuzzing as the mutation and execution of smaller inputs is more efficient. AFL [7], being the current state of the art fuzzer in this area, even trims new inputs on its own to reduce their size and additionally ships with a separate tool, `afl-tmin`, to perform more sophisticated test case minimization.

Other approaches to test case minimization can also be applied to minimize seed inputs for fuzzing. Groce, Alipour, Zhang, *et al.* propose an extension of delta debugging [16] to support test case minimization while retaining arbitrary properties of executions of a given target program. When used for seed minimization in the context of guided fuzzing, such approaches may benefit from additionally applying memory instrumentation to avoid cases like the one discussed in the example in Figure 2 in Section I. The MEMFUZZ modifications we propose for AFL involve changing the way the fuzzer trims new inputs to avoid this issue (but not the standalone `afl-tmin` tool). Other test case minimization tools could be adjusted in a similar way to leverage the MEMFUZZ memory instrumentation.

B. Instrumentation and Guidance

Research on instrumentation and guidance mechanisms for fuzzing either deals with devising new mechanisms or optimizing existing approaches for gathering coverage information or with the collection and usage of entirely different types of information to guide the fuzzing process.

Petsios, Zhao, Keromytis, *et al.* propose SlowFuzz [17] to automatically detect algorithmic complexity vulnerabilities. Like MEMFUZZ, SlowFuzz employs a different guidance mechanism from conventional evolutionary fuzzing, which does not exclusively rely on a notion of edge coverage. Instead, SlowFuzz uses resource usage, specifically the number of executed instructions, to guide the fuzzing process.

Hsu, Wu, Hsiao, *et al.* propose INSTRIM [18], an approach for reducing the number of basic blocks that need to be instrumented to gather edge coverage information in order to reduce runtime overhead. We pursue a similar objective with MEMFUZZ's static analysis described in Section III-B, but for memory access instrumentation rather than basic block instrumentation.

Gan, Zhang, Qin, *et al.* tackle the problem of *edge collisions* in AFL. The original AFL fuzzer may assign the same identifier to multiple edges, which results in inaccurate coverage information. Their solution, CollAFL [19], outperforms AFL in terms of coverage and crashes found.

AFLGo [20] is a modification of AFL for directed fuzzing that aims to extensively fuzz specific parts of the program under test for use cases like patch testing. AFLGo extends AFL's feedback and guiding mechanism to consider the distance to the targeted parts of the program under test.

Other approaches do not investigate different types of information as basis to guide the fuzzing but different ways of gathering edge coverage information. For instance, PTFuzz [21] and kAFL [3] rely on the Intel Processor Trace functionality of newer Intel x86 CPUs to obviate compile-time instrumentation and dynamic binary instrumentation. Such optimizations are not applicable to the instrumentation MEMFUZZ requires as the mechanisms they use are restricted to control flow tracing.

III. APPROACH

In the following, we present the design and implementation of MEMFUZZ, our approach to distinguish meaningfully different program executions with the same control flow based on the sets of memory addresses used in *read* and *write* memory accesses. MEMFUZZ provides novel guiding strategies to enhance the traditional coverage-guided evolutionary fuzzing and is implemented as an extension to the state of the art AFL fuzzer. It requires a compile-time instrumentation of the targeted programs, which we discuss in Section III-B, as well as certain runtime support, which we discuss in Section III-C. MEMFUZZ extends the information the fuzzing feedback loop can rely on to guide the fuzzing process, which we detail in Section III-D. We begin our discussion with an overview of MEMFUZZ and its integration with AFL in Section III-A.

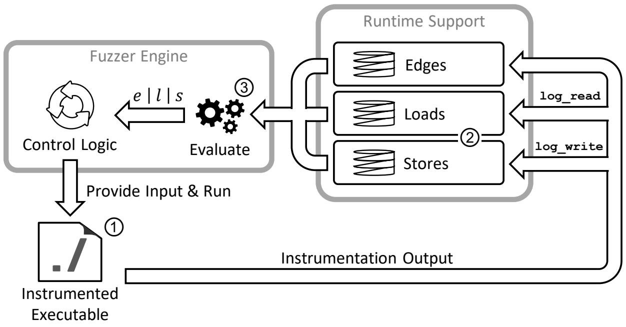


Fig. 3. MEMFUZZ: Design Overview

A. Overview

The overall design goal of MEMFUZZ is closing the blind spot of traditional coverage-guided fuzzing, which we illustrated with the example in Figure 2, where executions cannot be meaningfully distinguished by means of control flow only. In order to be of practical use, the MEMFUZZ implementation must be simple to apply to new targets and maintain good enough fuzzing performance. This is why we designed MEMFUZZ as extension to the state of the art AFL fuzzer, which is widely used and known for its performance.

Figure 3 provides an overview of the MEMFUZZ design, which consists of three essential parts: the compile-time instrumentation (marked ① in the figure), the runtime support library (marked ②), and the fuzzer feedback mechanism that categorizes an execution as interesting or uninteresting (marked ③). The MEMFUZZ instrumentation, which focuses on loads and stores, provides information on performed memory accesses in addition to AFL's edge coverage information. The accessed memory addresses are tracked within the runtime support library that employs two bloomfilters to efficiently track which addresses have been accessed to read or write memory. This extended pool of coverage information is then used by the fuzzer engine to evaluate the past execution and decide whether the used input is eligible for further mutation. The MEMFUZZ extension to AFL's evaluation mechanism allows the usage of all three information sources either separately or in certain combinations.

B. Instrumentation

To capture the addresses used in *read* and *write* memory accesses, we instrument such accesses at compile-time. As AFL already includes an LLVM-based mode and ships with an LLVM-instrumentation pass, we chose to build onto that feature for MEMFUZZ. In the LLVM intermediate representation (LLVM-IR), memory accesses are explicit and can only be performed using a small number of specific instructions, such as load for reading and store for writing memory locations. It is therefore especially well suited for the instrumentation of memory accesses as all instrumentation sites are apparent. Moreover, instrumenting memory accesses at compile time incurs lower runtime overhead than techniques using, e.g., dynamic binary instrumentation, and allows us to perform static analysis as discussed in Section III-B2.

1) Basic Memory Instrumentation: We start from the existing AFL LLVM instrumentation pass and enhance it to also instrument load and store instructions. By extending the existing pass, we ensure backwards compatibility as we leave the original edge instrumentation functionality untouched. The enhanced instrumentation pass inserts calls to logging functions, which reside in the support runtime library (cf. Section III-C), prior to loads and stores. An example of the memory instrumentation is shown in Figure 4. The small C function in Figure 4a reads from a computed memory location (d[i]) and writes to the global variable g. Figure 4b lists a simplified version of the corresponding LLVM-IR. Note that not only the read and write accesses are explicitly represented (lines 5 and 6) but also the address calculation (line 4) for the read access. The possible instrumentation sites are the load in line 5 and the store in line 6. The listing in Figure 4c shows the LLVM-IR after the MEMFUZZ instrumentation has been applied. Here, the load is instrumented by the insertion of a call to log_read (line 6) prior to the actual load. Note that the additional bitcast (line 5) is needed to correctly call log_read as type casts in LLVM are explicit, but it does not impose actual runtime overhead. The store in line 6 is not instrumented in this example due to MEMFUZZ's static analysis optimization that we detail in Section III-B2. The log_read function takes the address of the memory location the program is reading as its argument. The value that is read is not logged. store instructions are treated analogously using the log_write function.

2) Instrumentation Site Filtering: At runtime, MEMFUZZ-instrumented code incurs the overhead of an additional function call as well as the actual logging implementation (cf. Section III-C) for each instrumentation site. It is therefore desirable from a performance standpoint to avoid instrumentation where possible. To this end, our instrumentation pass includes a static analysis component that excludes instrumentation sites that correspond to certain classes of memory accesses that can be safely ignored without losing information beneficial for the guiding of the fuzzing process (i.e., memory accesses for which the address is not computed using the program input). For instance, a load from a global variable will use the same address for every access. Instrumenting such an access does not yield any information beyond the fact that the access has taken place. This fact, however, is already (more efficiently) detectable by means of control flow instrumentation. Our static analysis therefore excludes instrumentation sites that correspond to accesses to globals from instrumentation. The store in Figure 4b (line 6) is an example of this, and consequently, there is no call to log_write prior to the store in Figure 4c. Additionally, we also exclude accesses to stack variables allocated in the current frame unless they use a dynamically computed offset.

A simplified version of the algorithm we use to decide whether to filter out a given instrumentation site is given in Figure 5. Note that the pseudocode omits loop detection and result caching. The analysis is intraprocedural and conservative, which allows us to substantially reduce the number of instrumentation sites without incurring large compile time overhead or erroneously skipping relevant instrumentation sites.

C. Runtime

The log_read and log_write functions that are invoked by the inserted instrumentation code as described earlier are implemented within a runtime support library. The two functions implement the actual tracking of memory addresses. As the instrumentation and the runtime library are decoupled, a wide variety of design choices for the implementation of these functions are viable. As we aim to maintain backwards compatibility with AFL's edge coverage instrumentation, we extend AFL's runtime library with the runtime support functions our MEMFUZZ instrumentation requires. Moreover, we choose to focus on performance and favor simplicity in our implementation. The runtime library therefore only keeps track of the sets of memory addresses that are used to read or write memory locations. It does not track the order of accesses or the number of accesses to the same address. In order to avoid the need for heap allocations in the runtime, we employ bloom filters to represent the sets of memory addresses. As these are fixed-size data structures, memory overhead is independent of the number of accesses. Furthermore, adjusting bloom filter parameters allows tuning of the runtime for different application scenarios or target applications. Since bloom filters do not allow the retrieval of set members, it is not possible to reconstruct the exact set of addresses that were accessed during an execution afterwards. However, as we will discuss in Section III-D, this is not a drawback in this application context as we only need to decide whether a memory address has been seen before in an execution. We use the xxHash64 [22] algorithm for our bloom filter implementation.

D. Fuzzer

Our instrumentation and runtime provide the fuzzer with additional feedback for each execution. In addition to the edge coverage information, which the original AFL implementation relies on exclusively, our MEMFUZZ enabled fuzzer can use the two bloom filters containing load and store addresses to choose which inputs to mutate further. As discussed in Section III-C, bloom filters do not allow the retrieval of entries. However, by keeping track of the bloom filters seen during previous executions of the target program, the fuzzer can tell whether an execution has resulted in a reading or writing memory access that has not been seen during any previous execution. This is necessarily the case if any bit in the bloom filter is set that has not been set during any prior execution.

After each execution of the target program, there are three predicates on the execution available to the fuzzer:

```
int g = 0;

void f(int* d, long i) {
  g = d[i];
}
```

(a) C-Code

```
@g = global i32 0

define void @f(i32*, i64) {
 %4 = gep i32, i32* %0, i64 %1
 %5 = load i32, i32* %4
 store i32 %5, i32* @g
 ret void
}
```

(b) LLVM-IR Representation

```
@g = global i32 0

define void @f(i32*, i64) {
 %4 = gep i32, i32* %0, i64 %1
 %5 = bitcast i32* %4 to i8*
 call void @log_read(i8* %5)
 %6 = load i32, i32* %4
 store i32 %6, i32* @g
 ret void
}
```

(c) LLVM-IR with Memory Instrumentation

Fig. 4. A simplified example showing the operation of the instrumentation pass and the static analysis component.

```
procedure SkipAddr($a$: Addr)
    if IsGlobal($a$) then
        return true;
    else if IsLocalAlloca($a$) then
        return true;
    else if IsCast($a$) then
        $iv \leftarrow$ IncomingValue($a$);
        return SkipAddr($iv$);
    else if IsGEP($a$) then
        return SkipGEP($a$);
    else if IsPHI($a$) then
        return SkipPHI($a$);
    else
        return false;
    end if
end procedure

procedure SkipGEP($g$: GEPInst)
    if HasSafeConstantOffset($g$) then
        $a \leftarrow$ IncomingValue($g$);
        return SkipAddr($a$);
    else
        return false;
    end if
end procedure

procedure SkipPHI($p$: PHIInst)
    for all $iv \in$IncomingValues($p$) do
        if $\neg$SkipAddr($iv$) then
            return false;
        end if
    end for
    return true;
end procedure
```

Fig. 5. Instrumentation Site Filtering

- Whether the input resulted in new edge coverage (e);
- whether any previously unseen memory addresses were written to (s); and
- whether any previously unseen memory addresses were read from (l).

We use this information to determine whether an execution exhibited novel behavior, and therefore, whether the corresponding input should be mutated further. We do not change other parts of the fuzzer such as queue prioritization.

MemFuzz allows the use of any boolean expression over the aforementioned three predicates to be used as a novelty criterion by the fuzzer. We call such expressions strategies. Our prototype implementation supports both a conjunction or disjunction over any subset of predicates. As our goal is a more fine-grained rather than coarse-grained distinction between executions, we focus on logical disjunctions (e.g. $e \vee s \vee l$ or $s \vee l$) as guiding strategies. With such strategies, the fuzzer considers an execution to have exhibited novel behavior if, for instance, it resulted in new edge coverage *or* a store to a new memory address. As a result, the set of executions considered novel by any such strategy that also takes edge coverage into account is a superset of the set of executions considered novel by conventional coverage-guided fuzzing. This is in line with our goal of a more fine-grained distinction.

IV. Evaluation

To evaluate the proposed approach, we apply our prototype implementation to three evaluation targets and investigate the following research questions:

RQ 1 Does MemFuzz find different crashes from conventional, coverage-guided evolutionary fuzzing?

RQ 2 What runtime overhead does our prototype implementation impose?

RQ 3 How does MemFuzz affect the edge coverage of the generated inputs?

RQ 4 How much can the static analysis component reduce the number of necessary instrumentation sites?

In the following, we first describe our experimental setup and evaluation targets in Section IV-A. Then, we address the four research questions in Sections IV-B to IV-E.

A. Experimental Setup

Evaluation Targets: We evaluate our approach on the three target applications listed in Table I. The table lists for each target the used versions, the number of `load`/`store` instrumentation sites, the number of tokens for the fuzzing dictionary, and the number and size ranges for the fuzzing seed inputs. All three targets are widely used, parse complex file formats and are primarily implemented in C, which does not guarantee memory safety. We selected older versions of our targets to ensure that the fuzzer is able to find crashes in a reasonable amount of time.

Execution Environment: We use machines with an Intel Core i7-4790 CPU, 16 GiB of RAM, and a 500 GB SSD running Debian 8.10 with a Linux 4.9 kernel for all experiments.

TABLE I
OVERVIEW OF EVALUATION TARGETS

Application	Version	Description	Instr. Sites (Loads/Stores)	Dict-Tokens	Seed-Inputs
ffmpeg	2.0.1	Audio/video processing	360 684 (65.5 % / 34.5 %)	992	196 (55 B – 99 KiB)
ImageMagick	6.7.5-10	Bitmap image processing	295 373 (66.4 % / 33.6 %)	90	15 (41 B – 262 KiB)
libxml2	2.7.0	XML parser library/toolkit	139 572 (72.6 % / 27.4 %)	60	69 (5 B – 40 KiB)

TABLE II
OVERVIEW OF FUZZING CONFIGURATIONS

Designation	Fuzzer	Strategy	ASAN	Seed/Dict
AFL	AFL	Edge	No	Null
AFL+A	AFL	Edge	Yes	Null
MEM	AFLm	Mem	No	Null
MEM+A	AFLm	Mem	Yes	Null
HYB	AFLm	Mem + Edge	No	Null
HYB+A	AFLm	Mem + Edge	Yes	Null
AFL^S	AFL	Edge	No	Seed & Dict
AFL+A^S	AFL	Edge	Yes	Seed & Dict
MEM^S	AFLm	Mem	No	Seed & Dict
MEM+A^S	AFLm	Mem	Yes	Seed & Dict
HYB^S	AFLm	Mem + Edge	No	Seed & Dict
HYB+A^S	AFLm	Mem + Edge	Yes	Seed & Dict

Experiment Configurations and Execution: Table II lists all configurations we use for fuzzing each of our targets. We use the original AFL fuzzer in version 2.45b (being the version our implementation is based on) that relies exclusively on edges (predicate e, cf. Section III-D) as guiding strategy in the AFL configurations. For our modified AFL (AFLm in the table), we distinguish two configurations that use different guiding strategies: The first, MEM, relies exclusively on memory accesses to distinguish interesting executions ($l \vee s$). The second, HYB, is a hybrid strategy taking both edges and memory accesses into account ($l \vee s \vee e$). Additionally, we fuzz our targets both with and without AddressSanitizer (ASAN) [23] as well as with simple Null seed inputs without dictionaries and with a larger corpus of seed inputs and dictionaries to cover a variety of realistic scenarios. Overall, this amounts to a total of 12 distinct fuzzing configurations.

We execute each of these 12 configurations for each target for three hours using eight parallel instances (one master and seven secondary instances, totalling 24 h computation time). This process is repeated five times for a total of 180 experiments.

Dictionaries and Seed Inputs: Our Null seed input is identical for all targets and consists of a single file containing one byte with value 0x00. For the seeded configurations, we constructed an individual corpus of seed inputs and a token dictionary for each target (cf. Table I). The seed input corpus for ImageMagick and libxml2 is constructed from test case and example files that ship with the respective target. For ffmpeg, the seed input corpus is constructed from small sample files from the *FFmpeg Automated Testing Environment (FATE)*[1]. Starting with these files, we applied AFL's corpus and file minimization tools afl-cmin and afl-tmin to remove redundant input files and file contents which do not contribute to interesting program behavior in terms of edge coverage.

The token dictionaries were constructed by merging the dictionaries for the file types relevant for the respective targets that are included with AFL. The only exception is ffmpeg since AFL does not include dictionaries for video and audio files. We used libtokencap, which ships with AFL, on the samples from FATE to automatically extract tokens for all file types included in the corpus.

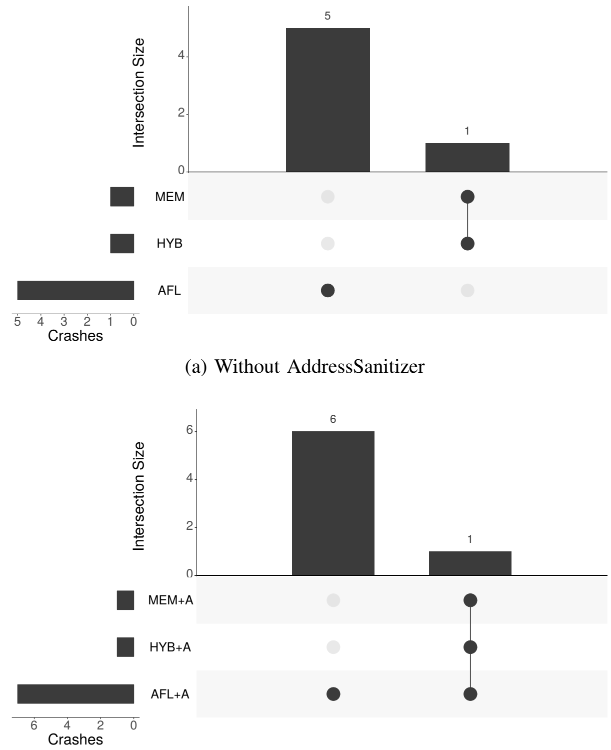
B. RQ 1: Crashes

To determine whether the MEMFUZZ approach is capable of finding different crashes compared to conventional, coverage-guided evolutionary fuzzing, we analyse the crashing inputs generated by each fuzzing configuration for each target. We re-use the same target binaries that were used for AFL+A, as they are built with AddressSanitizer, to reproduce all crashes and gather stack traces. We then bucket crashes using an enhanced form of stack hashing. We hash call stack addresses, program counter, stack pointer as well as base pointer values, and the cause of the crash as reported by AddressSanitizer (read, write, or signal) using the xxHash64 algorithm. In case of crashes caused by signals, we also include the signal type (e.g. SEGV or ABRT) but not the exact memory addresses triggering an AddressSanitizer error or signal. The additional information beyond the call stack and program counter is included to increase precision. We exclude memory addresses to ensure that this technique does not give an advantage to our approach.

The described stack hashing yields a set of hashes for each combination of fuzzing configuration and target (36 sets in total) where each hash corresponds to a distinct crash. We compare these sets for each target and consider all crashes found in at least one of the five repetitions. The results are visualized in Figures 6 to 10 as UpSet plots [24], [25]. The UpSet plots show the number of distinct crashes (horizontal bars at the left), the set intersections (connected circles), and the intersection sizes (vertical bars). Note that the intersections are exclusive and form a partition of the set of all crashes. We omit plots for cases in which there are no set intersections, e.g. because only one fuzzer found any crashes.

For ffmpeg, Figure 6a shows that, without the use of AddressSanitizer, both MEM and HYB found one crash. In the same runtime, AFL found five different crashes, but not the one found by MEM and HYB. With AddressSanitizer (Figure 6b), on the other hand, neither HYB+A nor MEM+A were able to detect crashes not found by AFL+A. In both configurations, MEM and HYB find the same crashes as each other.

[1] https://www.ffmpeg.org/fate.html



(a) Without AddressSanitizer

(b) With AddressSanitizer

Fig. 6. `ffmpeg` crashes

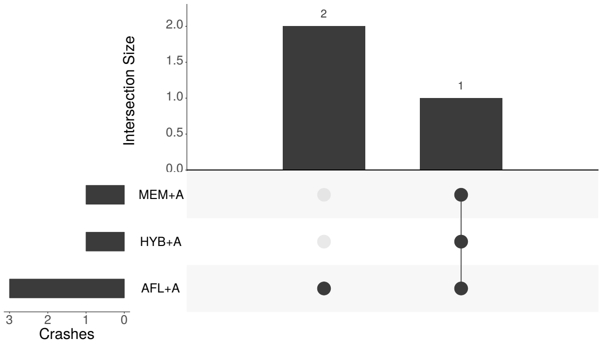
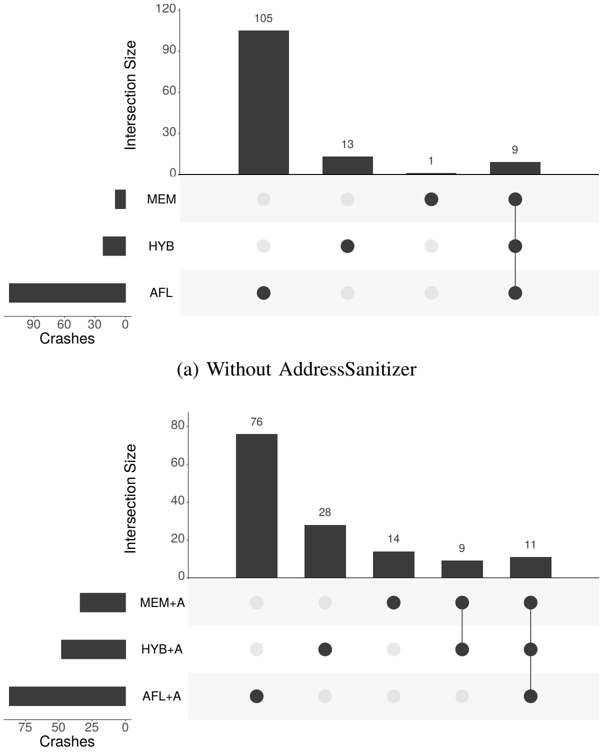


Fig. 7. `ffmpeg` Crashes with AddressSanitizer (seeded)

When using seed inputs without AddressSanitizer, AFL^S finds 5 distinct crashes whereas MEM^S and HYB^S do not find any. With AddressSanitizer (Figure 7), MEM+A^S and HYB+A^S perform as well as in the corresponding unseeded runs, whereas AFL+A^S finds fewer crashes than AFL+A. We attribute this to the use of a large seed input collection leading to longer individual runtimes and keeping the fuzzer from reaching more difficult crashes within the alloted time budget.

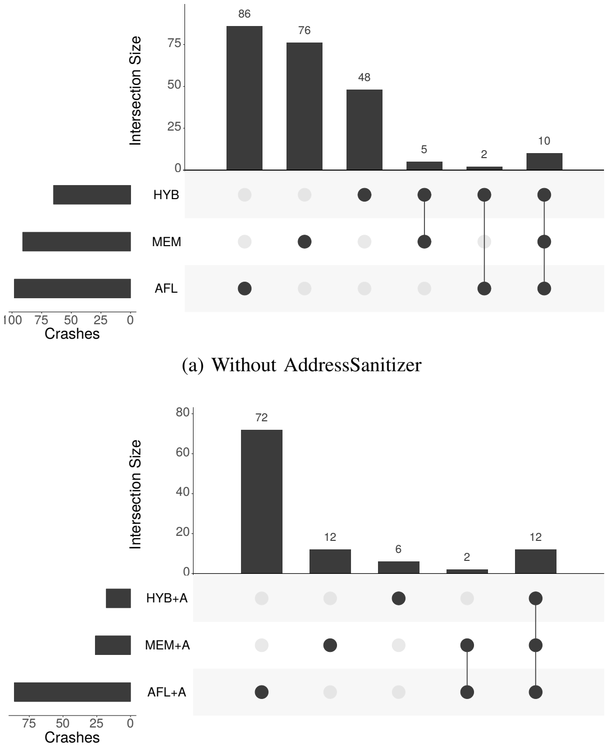
For `ImageMagick`, both with and without AddressSanitizer, MEM and HYB were able to find crashes not found by AFL, as shown in Figure 8. Unlike for `ffmpeg`, the crashes found by MEM and HYB in `ImageMagick` differ between the two fuzzers, with HYB outperforming MEM.



(a) Without AddressSanitizer

(b) With AddressSanitizer

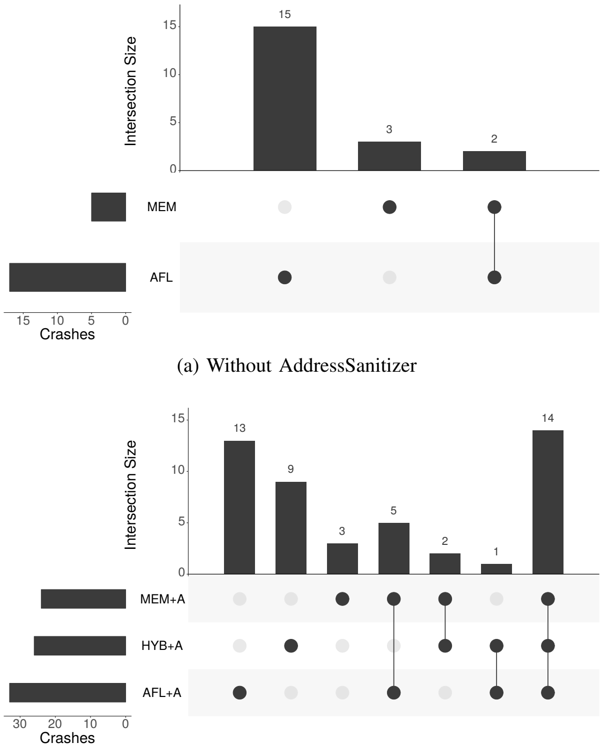
Fig. 8. `ImageMagick` crashes



(a) Without AddressSanitizer

(b) With AddressSanitizer

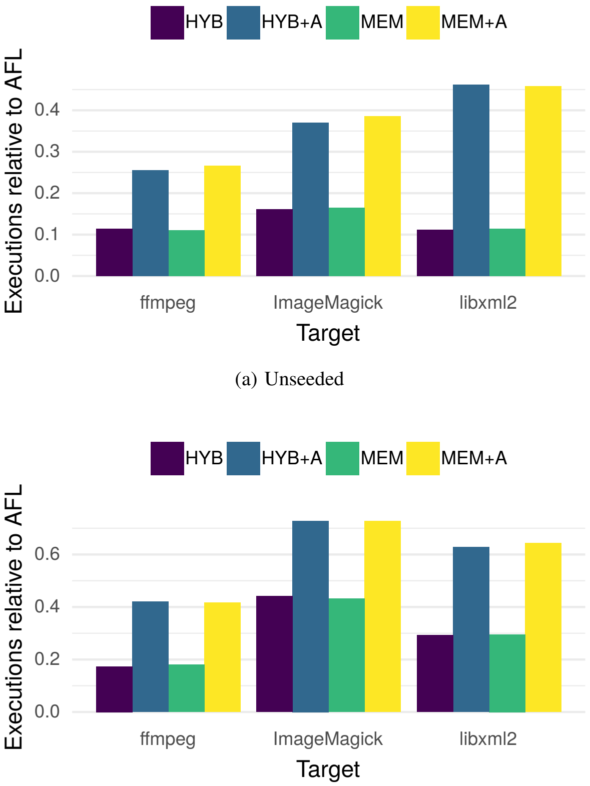
Fig. 9. `ImageMagick` crashes (seeded)



(a) Without AddressSanitizer

(b) With AddressSanitizer

Fig. 10. `libxml2` crashes (seeded)



(a) Unseeded

(b) Seeded

Fig. 11. Mean number of executions over five runs relative to AFL(+A).

With seed inputs HYB^{S} and especially MEM^{S} perform well, finding 48 and 76 crashes not found by any other fuzzer, respectively (Figure 9a). Both exceed the performance of their unseeded counterparts whereas AFL^{S} does worse than AFL. With AddressSanitizer (Figure 9b), on the other hand, both MEM+A^{S} and HYB+A^{S} do worse than their unseeded counterparts, likely due to the increased runtime.

For `libxml2`, without AddressSanitizer, no fuzzing configuration found any crashes. With AddressSanitizer, AFL+A found 149 distinct crashes whereas MEM+A and HYB+A did not find any crashes. MEMFUZZ performs better in the seeded experiments: As shown in Figure 10a, MEM^{S} finds 3 crashes that AFL^{S} does not. Moreover, when using AddressSanitizer, both MEM+A^{S} and HYB+A^{S} find crashes that AFL+A^{S} does not (Figure 10b).

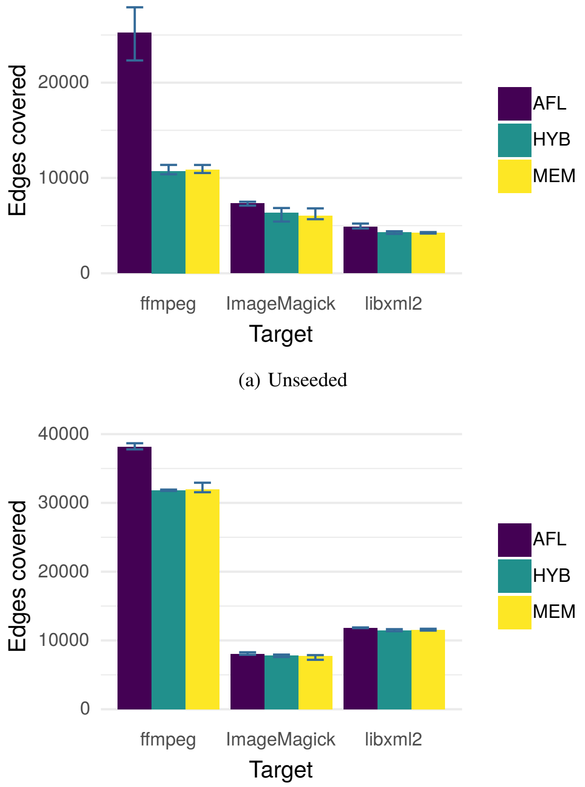
Across all targets for which at least one fuzzing configuration found crashes, the number of crashes found by MEM and HYB is lower than AFL. We hypothesize that this is primarily due to the additional runtime overhead, as discussed in Section IV-C. However, despite finding a lower number of crashes overall, both MEM and HYB find crashes not detected by AFL for all three targets we have evaluated, albeit not in all configurations. We therefore conclude that MEMFUZZ is capable of finding different crashes from conventional, coverage-guided evolutionary fuzzing.

C. *RQ 2: Overhead*

To determine the runtime overhead caused by the additional instrumentation and analysis required for MEMFUZZ, we compare the number of executions of each target configuration achieved by MEM and HYB to AFL.

For unseeded runs, the mean number of executions over five runs achieved by HYB and MEM relative to AFL is shown in Figure 11a. Without AddressSanitizer, MEM and HYB achieve between 11 % and 17 % of the executions achieved by AFL. With AddressSanitizer, they achieve between 25 % and 47 % of AFL+A executions. We hypothesize that this increase is due to the overhead incurred by AddressSanitizer instrumentation, which affects all fuzzers equally. In both cases, the overhead incurred by our MEMFUZZ implementation is highest for `ffmpeg`.

Overheads are lower for seeded runs, with MEM^{S} and HYB^{S} achieving 18 % and 17 % of AFL^{S} executions, respectively, on `ffmpeg`, an increase of 6 – 7 percentage points compared to the unseeded runs. On `ImageMagick`, the improvement is even larger at 27 – 28 percentage points, and on `libxml2`, performance relative to AFL^{S} is more than doubled compared to the unseeded runs. Configurations using AddressSanitizer see larger improvements still. On `ImageMagick`, MEM+A^{S} and HYB+A^{S} achieve 73 % of the executions of AFL+A^{S}. On `libxml2`, they achieve 64 % and 63 %, respectively. We hypothesize that



(a) Unseeded

(b) Seeded

Fig. 12. Mean edge coverage achieved by the different fuzzers over five runs. Error bars indicate minimum and maximum values.

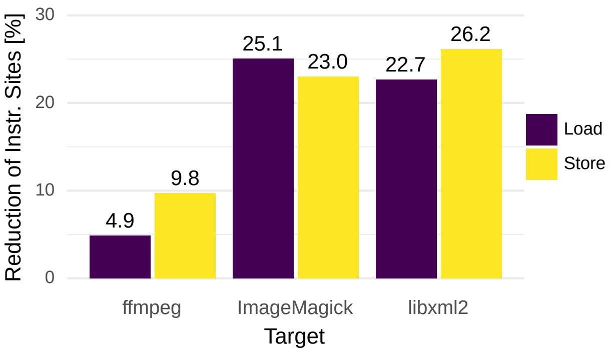


Fig. 13. Reduction of Load/Store Instrumentation Sites Due to Static Analysis

the reduced slowdown for seeded runs is the result of longer execution durations for all fuzzers amortizing the more complex evaluation step after each execution.

We conclude that, while our implementation does incur substantial runtime overhead, the slowdown is highly variable between targets and configurations, ranging from 1.36x to more than 5x. The effect is lower when using appropriate seed inputs and AddressSanitizer, both of which are generally desirable in a wide variety of fuzzing use cases.

D. RQ 3: Coverage

As we modify the fuzzing engine and introduce additional instrumentation and consequently overhead (as discussed in Section IV-C), we expect our MEMFUZZ implementation to achieve a reduced edge coverage relative to conventional, coverage-guided evolutionary fuzzing. To assess the extent of the impact, we check the inputs generated by all fuzzers and compare the covered edges. Coverage for all fuzzers is checked on the same binary for each target. An edge is considered covered by a fuzzer if it was covered during at least one of the five executions by any of the eight parallel instances.

Results for the unseeded runs without AddressSanitizer are shown in Figure 12a. AFL consistently outperforms both MEM and HYB for all targets, but the size of the effect varies substantially between targets. On ffmpeg, AFL covers more than twice as many edges as MEM or HYB, whereas the difference on the other targets is much smaller, with both MEM and HYB achieving over 80 % of the coverage achieved by AFL. MEM and HYB perform similarly well on all three targets, with HYB having a slight edge on ImageMagick and libxml2 while MEM does marginally better on ffmpeg.

For the seeded runs (Figure 12b), HYB and MEM are much more closely matched with AFL. On ffmpeg, both go from below 50 % of AFL's coverage to above 80 %. On ImageMagick and libxml2, MEM and HYB cover more than 96 % of the edges covered by AFL.

While some impact on coverage is to be expected due to the overheads reported in Section IV-C, our results show that the extent of this impact is highly dependent on the specific program under test and the quality of the seed inputs. We conclude that our approach does adversely affect edge coverage but the effect can be mitigated with careful seed selection.

E. RQ 4: Static Analysis

In Section III-B, we describe our technique for reducing the number of instrumentation sites that we employ to reduce overall runtime overhead as each instrumentation point imposes a runtime cost. To assess how well our technique works in practice, we log both the number of instrumentation sites where instrumentation was applied and where it could be elided during the execution of our LLVM instrumentation pass. We report the percentage of load and store instructions in each of our fuzzing targets for which instrumentation could be elided by our technique in Figure 13.

For both ImageMagick and libxml2 the number of necessary load instrumentations could be reduced by over 22 % and the number of store instrumentations by over 23 %. For ffmpeg, on the other hand, load instrumentations could be reduced by only about 5 % and store instrumentations by about 10 %. To put these percentages into perspective, Table I reports the absolute total numbers of instrumentation sites per target. Overall, our technique saves 72 048 instrumentation points for ImageMagick, 33 016 for libxml2, and 23 681 for ffmpeg. As these numbers demonstrate, our static analysis and instrumentation elision technique is capable of significantly reducing the number of necessary instrumentation points, which translates to a reduction of runtime overhead.

V. Discussion and Threats to Validity

A. Discussion

1) Instrumentation and Overhead: While the evaluation shows that our static analysis is capable of significantly reducing the number of instrumentation sites (cf. Section IV-E), our approach nonetheless incurs a substantial runtime overhead. Moreover, the evaluation shows that the target for which the static analysis was least effective at reducing the number of instrumentation points (`ffmpeg`) is not the target with the largest overhead. We hypothesize that this is due to our evaluation of the static analysis considering instrumentation sites at compile time, whereas runtime overhead is determined by the number of instrumentation sites encountered at runtime. A more powerful static analysis could further reduce instrumentation sites and runtime overhead. We consider this as future work.

2) Overhead, Coverage, and Crash Finding Ability: We find that the ability of our approach to find crashes is not directly linked to its overhead, the number of executions relative to AFL or the edge coverage relative to AFL. For instance, HYB+A and MEM+A perform better in terms of executions relative to AFL+A and coverage on `libxml2` than on the other two evaluation targets. At the same time, they do not find any crashes in `libxml2` when not using seed inputs while AFL+A finds 149. Despite higher overheads and lower coverage, the performance in terms of crashes found relative to AFL+A is better on `ImageMagick` and `ffmpeg`. This suggests that our feedback mechanism is ill-suited for overcoming the early parsing stages in `libxml2`. When using seed inputs, this problem disappears, and consequently, MEM+A^S and HYB+A^S perform much better than their unseeded counterparts in terms of their ability to find crashes. These results highlight the importance of picking suitable seed inputs. Moreover, coverage appears to be a poor proxy for crash finding ability in general.

3) Future Work: There is potential to further reduce overheads through more powerful static analysis or a more optimized implementation. Another promising avenue for future work is the use of memory access instrumentation as part of the queue prioritization or for more targeted mutations in the fuzzer rather than just as an indicator for novelty. Exploring other strategies and combinations of memory access-guided and conventional coverage-guided fuzzing in a parallel fuzzing setting, such as running AFL, MEM, and HYB in parallel on a shared queue may also be worthwhile.

B. Threats to Validity

1) Choice of Evaluation Targets: We have chosen three evaluation targets. All three targets are widely used and well-suited for fuzzing due to being written in a language that does not provide memory safety and handling complex data structures. Our results may not generalize to other targets, and to applications or libraries that do not have these properties.

Moreover, the number of instrumentation sites in a target application and the achievable reduction in that number depends not just on the target application but also on the employed compiler version and compiler optimizations. In other settings, the number or reduction in instrumentation sites may differ. Consequently, overhead may differ as well.

2) Crash Bucketing: To determine whether our proposed approach finds different crashes from conventional, coverage-guided evolutionary fuzzing, we use stack hashing to bucket the crashing inputs found by each fuzzer and assign a unique identifier to each bucket. However, crash bucketing using standard techniques is known to be imprecise [26]. We first filter crashes using AFL's unique crashes heuristic as part of the fuzzing process, then apply our stack hashing variant, but both techniques have known shortcomings and may suffer from both over- and underapproximation. Without manual inspection, it is not possible to precisely map crashes to underlying bugs in the general case. Therefore, it is possible that we over- or underapproximate the number of crashes. In the former case, we may erroneously map crashes with the same underlying cause to different buckets. In the latter case, crashes with different underlying causes get the same identifier. Both issues would affect both the total number of crashes reported as well as the comparisons between techniques performed in Section IV-B. As discussed in Section IV-B, we attempt to mitigate this concern by omitting specific memory addresses connected to crashes from the hash to ensure that deduplication does not grant an advantage to our proposed technique.

3) Experiment Durations: All results reported in our evaluation have been gathered with the setup and experiment duration described in Section IV-A. Substantially different experiment durations may affect how different fuzzers perform relative to each other. While we have attempted to mitigate the effect of nondeterministic runtime behavior by repeating all experiments five times, additionally assessing the impact of longer experiment durations is not feasible with the computational resources we have available.

VI. Conclusion

We have introduced MemFuzz, an approach for using memory access instrumentation instead of or in addition to control flow information to guide evolutionary fuzzing. We have implemented a prototype of MemFuzz based on AFL. It comprises a static analysis and instrumentation component, a runtime component, and a modified version of the fuzzer capable of taking memory access information into account as a novelty criterion according to different, user-specified strategies. Our evaluation shows that, despite incurring non-negligible runtime overhead, the approach is capable of finding crashes that coverage-guided fuzzing does not find within the same time budget for all three targets in our evaluation.

Acknowledgments

We thank Raik Joachim for his contributions to the MemFuzz implementation.

This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP at TU Darmstadt.

REFERENCES

[1] B. Jiang, Y. Liu, and W. K. Chan, "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection," in *Proc. of ASE*, 2018, pp. 259–269.

[2] *Syzkaller*. [Online]. Available: https://github.com/google/syzkaller (visited on 10/09/2018).

[3] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *Proc. of USENIX Security*, 2017.

[4] *Heartbleed*. [Online]. Available: http://heartbleed.com/ (visited on 10/11/2018).

[5] *Cloudbleed*. [Online]. Available: https://bugs.chromium.org/p/project-zero/issues/detail?id=1139 (visited on 10/11/2018).

[6] *CVE-2014-7186*. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7186 (visited on 10/11/2018).

[7] M. Zalewski, *American Fuzzy Lop*. [Online]. Available: http://lcamtuf.coredump.cx/afl/ (visited on 10/09/2018).

[8] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. of NDSS*, 2017.

[9] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, 2018.

[10] C. Lemieux and K. Sen, "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage," in *Proc. of ASE*, 2018, pp. 475–485.

[11] C. Lemieux, R. Padhye, K. Sen, and D. Song, "PerfFuzz: Automatically Generating Pathological Inputs," in *Proc. of ISSTA*, 2018, pp. 254–265.

[12] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *Proc. of IEEE S&P*, 2018, pp. 711–725.

[13] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, "Optimizing seed selection for fuzzing.," in *Proc. of USENIX Security*, 2014, pp. 861–875.

[14] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proc. of IEEE S&P*, 2017, pp. 579–594.

[15] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, "Cause reduction: Delta debugging, even without bugs," *Software Testing, Verification and Reliability*, vol. 26, no. 1, pp. 40–68,

[16] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proc. of FSE*, 2002, pp. 1–10.

[17] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities," in *Proc. of CCS*, 2017, pp. 2155–2168.

[18] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing," *Workshop on Binary Analysis Research (BAR)*, 2018.

[19] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *Proc. of IEEE S&P*, 2018, pp. 679–696.

[20] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. of CCS*, 2017, pp. 2329–2344.

[21] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min, "PTfuzz: Guided Fuzzing With Processor Trace Feedback," *IEEE Access*, vol. 6, 2018.

[22] *xxHash*. [Online]. Available: http://xxhash.com (visited on 10/09/2018).

[23] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proc. of USENIX ATC*, 2012, pp. 309–318.

[24] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister, "UpSet: Visualization of Intersecting Sets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 1983–1992, 2014.

[25] J. R. Conway, A. Lex, and N. Gehlenborg, "UpSetR: an R package for the visualization of intersecting sets and their properties," *Bioinformatics*, vol. 33, no. 18, pp. 2938–2940, 2017.

[26] R. van Tonder, J. Kotheimer, and C. Le Goues, "Semantic crash bucketing," in *Proc. of ASE*, 2018, pp. 612–622.