

This paper presents Fuzzm which is a tool that finds memory bugs in WebAssembly programs without needing source code. It works by adding canaries (special markers) to the stack and heap to detect problems like buffer overflows. The tool tested many real Wasm programs and found real crashes and bugs. It also showed that WebAssembly is more vulnerable than native code because it has no built-in protections like stack canaries. So we can use it to confirm that memory bugs can break the sandbox and gives us ideas to test WAMR for similar problems.

## Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly

Daniel Lehmann\*  
University of Stuttgart,  
Germany  
mail@dlehmann.eu

Martin Toldam Torp\*  
Aarhus University,  
Denmark  
torp@cs.au.dk

Michael Pradel  
University of Stuttgart,  
Germany  
michael@binaervarianz.de

### Abstract

WebAssembly binaries are often compiled from memory-unsafe languages, such as C and C++. Because of WebAssembly’s linear memory and missing protection features, e.g., stack canaries, source-level memory vulnerabilities are exploitable in compiled WebAssembly binaries, sometimes even more easily than in native code. This paper addresses the problem of detecting such vulnerabilities through the first binary-only fuzzer for WebAssembly. Our approach, called Fuzzm, combines canary instrumentation to detect overflows and underflows on the stack and the heap, an efficient coverage instrumentation, a WebAssembly VM, and the input generation algorithm of the popular AFL fuzzer. Besides as an oracle for fuzzing, our canaries also serve as a stand-alone binary hardening technique to prevent the exploitation of vulnerable binaries in production. We evaluate Fuzzm with 28 real-world WebAssembly binaries, some compiled from source and some found in the wild without source code. The fuzzer explores thousands of execution paths, triggers dozens of crashes, and performs hundreds of program executions per second. When used for binary hardening, the approach prevents previously published exploits against vulnerable WebAssembly binaries while imposing low runtime overhead.

### 1 Introduction

WebAssembly is an increasingly important bytecode language [20, 55, 56] with low-level semantics, fast execution, and a multitude of source languages compiling to it. It is widely supported in browsers<sup>1</sup> and used by diverse web applications [24], for “serverless” cloud computing [21, 52], in smart contract platforms [23, 36, 46], to sandbox libraries in native applications [40, 58], and even as a universal bytecode by standalone WebAssembly runtimes [1, 2, 13].

Given its importance, the security of WebAssembly is also becoming more and more relevant. While WebAssembly prevents some security issues by design, source-level vulnerabilities may still propagate to WebAssembly binaries [8, 30, 37].

Recent work [30] has shown that, surprisingly, memory vulnerabilities in WebAssembly binaries can sometimes be even more easily exploited than when the same source code is compiled to native architectures. One reason is the lack of mitigations, such as stack canaries, page protection flags, or hardened memory allocators [30].

To find vulnerabilities, *greybox fuzzing* has proven to be an effective technique [9, 22, 32, 47, 59]. For example, Google’s OSS-Fuzz project has found thousands of vulnerabilities in widely used software [3, 51]. A greybox fuzzer automatically generates inputs that explore the target program and eventually trigger a vulnerability. For that, it requires (i) lightweight feedback from the execution, e.g., coverage information, to guide the input generation, and (ii) runtime oracles that make a vulnerability apparent, e.g., by crashing the program.

A greybox fuzzer for WebAssembly would be highly desirable, but several characteristics of WebAssembly must be taken into account. First, WebAssembly is a compilation target for multiple source languages, including C, C++, Rust, Go, and many others [24]. A fuzzer aimed at a specific source language hence could analyze only a fraction of all real-world binaries. Second, the source code of a WebAssembly binary may not be available, e.g., when analyzing third-party websites, third-party libraries, or in-house legacy applications. Even if the source code is available, adopting a fuzzer into the development workflow is made harder if it requires changes to the build system, or specific (versions of) compilers. Third, even when compiling from the same source code, the security-relevant behavior of a program compiled to WebAssembly may differ from the same program being compiled to native code [30]. As we illustrate in Section 3, whether a vulnerability can be exploited depends on how the semantics of the source language are compiled and which mitigations the target platform provides. As a result, fuzzing a program compiled for another platform, e.g., x86 [16], is insufficient to expose memory bugs in WebAssembly. Taken together, these characteristics motivate a fuzzer targeted at WebAssembly binaries. However, despite the overall success of greybox fuzzing and the increasing importance of WebAssembly, such a fuzzer currently does not exist.

\*Both authors contributed equally to the paper.

<sup>1</sup>> 94% support as of October 2021, see <https://caniuse.com/wasm>.

This paper presents Fuzzm<sup>2</sup>, the first binary-only greybox fuzzer for WebAssembly. Its main components, shown in Figure 1, address several interesting technical challenges. First, unlike native programs, WebAssembly lacks several built-in oracles that native fuzzers rely on for finding suspicious program behavior. For example, none of the current compilers targeting WebAssembly adds stack canaries [14, 45], and due to WebAssembly’s linear memory, overflows from, e.g., stack to heap data remain unnoticed [30]. While tools like AddressSanitizer [50] can instrument source code to detect memory-related misbehavior, they do not apply to binaries. Instead, our stack and heap canary instrumentation rewrites binaries to detect over- and underflows on the stack and heap. Besides fuzzing, the canaries are also useful for retroactively hardening existing WebAssembly binaries in production.

Second, in a binary fuzzer we cannot rely on compiler-inserted code to track coverage, which is what AFL and other fuzzers do [9, 32, 47]. Even though there are dynamic instrumentation approaches for binaries, e.g., AFL’s QEMU mode, they often suffer from high overheads, and are architecture-dependent and not applicable to WebAssembly. Our coverage instrumentation instead applies to unmodified, production WebAssembly binaries and tracks coverage efficiently.

The final challenge, especially when fuzzing bytecode programs, is efficiency. WebAssembly binaries are executed in a virtual machine (VM), which may cause a naive approach to suffer from high start-up time and makes fuzzing impractical. Instead, we integrate a WebAssembly VM that executes the target program with the tried-and-tested input generation of AFL. Here, WebAssembly’s sandboxing can actually be an opportunity rather than a drawback: The memory of the target application and AFL can reside in a single address space, without the need for different processes separating the two.

The result of addressing the above challenges is a practical, effective, and efficient end-to-end fuzzer for WebAssembly binaries. Our evaluation applies Fuzzm to 28 programs, of which ten are well-known programs compiled from source code to WebAssembly, and 18 are WebAssembly binaries without source code found in the wild. We find our approach to be effective, covering 1,232 unique execution paths and triggering 40 unique crashes on average during 24 hours of fuzzing. The majority of the triggered crashes are due to our canary-based oracles. In terms of efficiency, Fuzzm performs hundreds of program executions per second, comparable to AFL, despite requiring only a binary as input and running the program in a VM. Finally, we show that the canaries inserted by our instrumentations effectively prevent three previously published exploits against vulnerable WebAssembly binaries [30]. Due to their low runtime overhead (1.05x and 1.06x, for stack and heap canaries, respectively) the canary instrumentation serves, beyond fuzzing, as a standalone hardening tool for existing, vulnerable WebAssembly binaries.

<sup>2</sup>“Fuzzm” is a portmanteau word of “fuzzing” and “Wasm”.

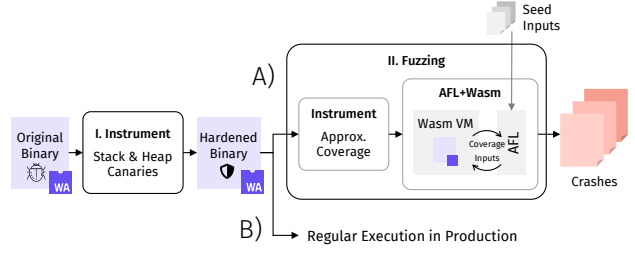


Figure 1: Overview of the main components of Fuzzm.

**Contributions.** In summary, this paper contributes:

- The first *binary-only fuzzer for WebAssembly* programs.
- A binary instrumentation that inserts stack and heap canaries, which can be used to *harden existing WebAssembly programs* and as an *oracle in our fuzzer* (Section 4).
- Integration of the AFL fuzzer and its tried-and-tested input generation, a binary-only instrumentation that provides compatible coverage information, and a WebAssembly VM, for *efficient end-to-end fuzzing* (Section 5).
- Empirical evidence that Fuzzm *effectively explores paths and finds crashes* in well-known programs compiled to WebAssembly, and in large, real-world WebAssembly binaries without source code (Section 6), and
- Empirical evidence that binaries hardened with our canary instrumentations run with *low runtime overhead* and *effectively thwart previously published exploits* (Section 6).

## 2 Background on WebAssembly

We briefly introduce features of WebAssembly most relevant to this paper, and some of the security aspects of the language. WebAssembly is an assembly-like language designed as a portable compilation target from different source languages, e.g., from C/C++ with Emscripten or Clang, or from Rust. Beyond the browser platform, which was the first to widely support WebAssembly, there now are several other platforms, e.g., Node.js and the standalone Wasmtime VM<sup>3</sup>.

**Types.** WebAssembly is a stack-based, statically typed language. There are four types in WebAssembly: `i32` (`i64`) for 32-bit (64-bit) integers and `f32` (`f64`) for 32-bit (64-bit) floating point numbers. Typed instructions push and pop values from an implicit *operand stack*. For example, `i32.const N` pushes the 32-bit constant integer  $N$ , and `f64.add` pops two 64-bit floating point numbers and then pushes their sum.

**Control flow.** Unlike many other assembly-like languages, WebAssembly features structured control flow, which is encoded using nested blocks. A block is a sequence of instructions that either begins with a block or loop and ends with an end instruction.<sup>4</sup> Blocks may be nested arbitrarily deep. Within a block, a `br` (`br_if`)  $L$  instruction (conditionally)

<sup>3</sup><https://wasmtime.dev/>

<sup>4</sup>There are also `if` and `else` blocks, but they add no expressive power over regular blocks and branches, so we do not explain them here for brevity.

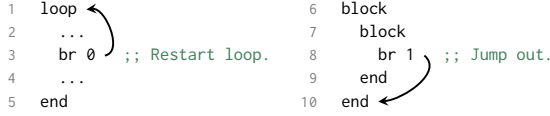


Figure 2: Branching in WebAssembly illustrated

jumps to the end (or for loops, to the beginning) of the  $L$ th block, where 0 is the block containing the `br_if` instruction, 1 is its parent, and so on.  $L$  can be thought of as a numerical, relative block *label*. For example, the `br` instruction on line 3 of Figure 2 jumps to the beginning of the loop block on line 1, and similarly, the `br` on line 8 jumps out two blocks to the end on line 10. A `br_table  $L_0 \dots L_t, D$`  instruction implements jump tables. It consumes the top-most integer from the operand stack  $i$  and jumps to  $L_i$ , or a default  $D$  if  $i > t$ .

**Functions and variables.** A function in WebAssembly has typed parameters, typed local variables, and a sequence of instructions. Parameters and local variables are read using `local.get  $N$`  and written using `local.set  $N$` , where  $N$  refers to the  $N$ th local variable or parameter. Most instructions, including direct calls, are statically type checked; indirect calls are type checked at runtime. There are also global variables, which are read (written) with `global.get` (`global.set`).

**Memory.** Unlike native programs, WebAssembly uses a byte-addressable *linear memory* for storing long-lived objects. The memory is initialized with a certain size when the module is instantiated and can be grown at runtime using the `memory.grow` instruction. The 32-bit address space has no holes, so every pointer  $\in [0, \text{size}]$  is valid. The memory is fully program-organized, i.e., there is no garbage collection. Load and store instructions take `i32` values as addresses. For example, `i64.store` consumes two elements from the operand stack: an `i64` value and an `i32` address, at which it stores the eight byte value in linear memory.

**WASI.** WebAssembly does not have a standard library or I/O functions by default. Instead all interaction with the underlying host system needs to happen through imports. WASI (the WebAssembly System Interface) specifies a syscall interface of functions for performing I/O, filesystem access, etc.<sup>5</sup> A program written in some high-level language (typically C, C++, or Rust) can be compiled to a WASI binary, and then executed by a WASI-compliant runtime, such as Wasmtime. There are other ways of executing WebAssembly, e.g., in the browser or on Node.js, but we focus on WASI in this work.

**Security.** WebAssembly has a two-sided security story. On the one hand, WebAssembly programs execute in a sandboxed environment, which isolates them well from the memory and code of the underlying host system. This prevents many attacks in the browser or cloud setting. On the other hand, protection of WebAssembly program’s *own* memory is very limited [8, 18, 30], even compared to native binaries. As

<sup>5</sup><https://wasi.dev/>

the operand stack only stores primitive values, linear memory must be used for all non-primitive values on the stack, all static data, and the heap. Current compilers do not insert canaries into the stack in linear memory or any other protection mechanism for detecting buffer overflows at runtime. This issue is aggravated by the fact that there are no guard pages between memory regions, allowing, e.g., a buffer overflow on the stack to run over static data or data on the heap. There is also no way of marking parts of linear memory as read-only; it is always writable everywhere. Due to these issues, WebAssembly programs can be exploited in practice, causing cross-site scripting, remote code execution, and other malicious behaviors [30], which motivates the need for tools to both discover memory-related vulnerabilities in WebAssembly binaries, and mitigate exploits at runtime.

### 3 Overview and Motivating Example

Our approach consists of two main components, as shown in Figure 1. First, we present a novel *binary-only canary instrumentation* (Section 4) that hardens WebAssembly applications by adding stack and heap canaries. Second, we present a *binary-only fuzzer for WebAssembly* (Section 5). It integrates several components into an effective and efficient end-to-end fuzzer: novel instrumentation to gather coverage information directly from WebAssembly binaries, a WebAssembly VM, and the input generation abilities of the proven AFL tool. The remainder of this section illustrates our approach with a motivating example, subsequent sections fill in the details.

**Example.** The program in Figure 3 suffers from a textbook buffer overflow on the stack (line 3) that can be potentially triggered by the right inputs (lines 10 and 3). Because of differences in compilers, system libraries, and protection features, the vulnerability is not exploitable when compiled to a modern native architecture, such as x86-64, but it can be exploited when compiled to WebAssembly [30]. Figure 4a and b show the stack layout of the program when executed as a native binary (compiled with GCC) and as a WebAssembly binary (compiled with Emscripten). Because the variables `input1` and `input2` are stored in a different order on the stack, an attacker overflowing `input1` cannot change the program behavior in the native binary, but *can* do so in WebAssembly. It is thus important to fuzz test the WebAssembly binary, and not only a native binary compiled from the same source.

**Canary instrumentation.** To detect executions that exploit vulnerabilities like the above example, we present a binary-only instrumentation technique that adds protections in the form of stack and heap canaries. The approach instruments every function in the binary with code that inserts a canary onto the stack frame upon entry and checks it upon function exit. Beyond stack overflows, the instrumentation also detects memory violations on the heap by surrounding heap chunks with canaries. Figure 4c shows the inserted canary on the stack of the example program. An attack writing

```

1 void vulnerable() {
2     char input1[8];
3     scanf("%16s", input1); // Buffer overflow!
4     char input2[8];
5     scanf("%8s", input2);
6     /* more code... */ }
7 int read_int() { int i; scanf("%d", &i); return i; }
8 int main(int argc, char** argv) {
9     char data[10] = "some data";
10    if (read_int() == 42) // Input, figured out by fuzzer.
11        vulnerable();
12    if (strcmp(data, "some data") == 0)
13        puts("equal");
14    else puts("not equal"); }

```

Figure 3: Example program with a vulnerability (simplified).

beyond the buffer will overwrite the canary, which the instrumented binary will detect and abort execution.

The canary instrumentation serves two purposes, marked with A) and B) in Figure 1. A) The primary purpose explored in this paper is as an oracle during fuzz-testing. If a fuzzer successfully generates an input that causes an overflow (e.g., of `input1` in the example), it might remain unnoticed, unless the overflow causes a crash. Analogous to dynamic checks for memory corruptions in native programs [16, 45, 48], our stack and heap canary instrumentation provides a precise test oracle that warns about memory corruptions observed during an execution. B) Beyond fuzzing, our instrumentation also serves as a hardening technique for binaries running in production. The instrumentation mitigates exploits by detecting overflows during an execution, where it can terminate the program and hence prevent exploitation. As we show in our evaluation, this protection comes with low overhead and can be applied to large, real-world binaries compiled from C, C++, and Rust.

**Fuzzing WebAssembly.** The second main component of our approach is the actual fuzzer. We use a greybox fuzzing approach based on the widely used AFL fuzzer and its proven input generation. The high-level workflow is that of standard greybox fuzzing: Starting from some seed input(s), repeatedly execute the program, and gather coverage feedback, which is used to mutate inputs until triggering a crash. In the example program, AFL’s input generation eventually figures out to start the input with “42” (line 9) to explore more behavior in function `vulnerable`. However, in said function, native AFL fails to detect a vulnerability due to the different stack layouts between the native and WebAssembly binaries, whereas Fuzzm finds a crashing input after few minutes of fuzzing.

Applying AFL-style greybox fuzzing to WebAssembly is non-trivial for two reasons. First, the fuzzer requires coverage information, which native AFL obtains by inserting code when compiling the program from source. However, we want to fuzz WebAssembly binaries without requiring access to the source code. We hence present a novel binary-only instrumentation technique to gather AFL-compatible coverage information from WebAssembly binaries. Second, to be practical, fuzzers typically perform hundreds of executions of

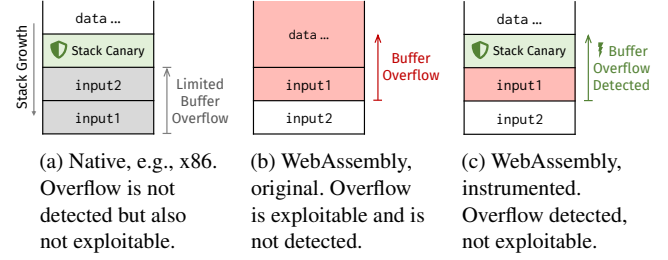


Figure 4: Stack layouts of the example program.

a program per second. We present a set of WebAssembly-specific adaptations of the original AFL that achieve this level of efficiency.

## 4 Hardening WebAssembly Programs with a Binary-Only Canary Instrumentation

The lack of virtual memory, page protections, or compiler-inserted mitigations [14] makes WebAssembly programs more vulnerable to buffer overflows than native programs, with unique and surprising consequences such as overwriting supposedly constant data and stack-to-heap overflows [30]. While linear memory is a core part of the language that cannot be changed, over- and underflows on the stack and heap should be detected at runtime. To do so, we present an instrumentation that statically inserts stack and heap canaries in WebAssembly binaries. Similar to prior work on canaries on the stack [19, 45] or the heap [42, 48] for native programs, the basic idea is to surround memory chunks with a special value, called the *canary*, and to check whether this value was overwritten, e.g., before deallocation or when returning from a function. Our approach differs from prior work in three ways. First, to the best of our knowledge, we are the first to present a canary-based protection for WebAssembly. Current compilers do not implement canaries, so many already existing WebAssembly binaries are potentially vulnerable. Second, in contrast to, e.g., compiler-inserted canaries or the popular AddressSanitizer [50], our approach does not require source code but instruments WebAssembly binaries directly. This allows us to retroactively harden existing binaries. Finally, in contrast to binary-only techniques for native programs, such as Valgrind [41] or Intel Pin [33], our approach performs reliable, static instrumentation. As a result, the runtime overhead of our instrumented binaries is negligible instead of the much higher overheads caused by dynamic instrumentation.

### 4.1 Stack Canaries

To detect buffer overflows that write beyond the current stack frame, Fuzzm performs three transformations on each function in a binary, as illustrated in Figure 5. First, we insert a preamble that injects the canary value onto the stack in linear memory. Second, we wrap the original body in a new block and rewrite all returns such that they jump to the end of



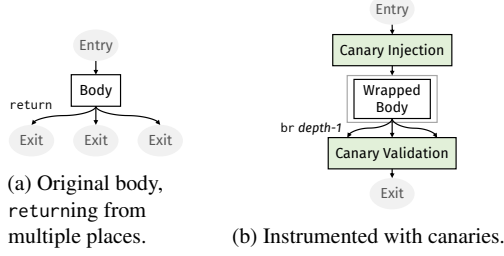


Figure 5: CFGs of a WebAssembly function before and after the instrumentation. The rectangles represent subgraphs.

**Algorithm 1** Stack canary instrumentation.

The ++ operator denotes concatenation.

```

1: procedure INSTRUMENTFUNCTION(body)
2:   canary ← eight randomly generated bytes
3:   body ← INJECTCANARY(canary) ++ body
4:   body ← block ++ body ++ end    ▷ Wrap original body
5:   depth ← 0
6:   for instr in body do
7:     if OPENSBLOCK(instr) then    ▷ Track block depth
8:       depth ← depth + 1
9:     else if CLOSESBLOCK(instr) then
10:      depth ← depth - 1
11:    if instr = return then        ▷ Redirect returns
12:      instr ← br (depth - 1)
13:   body ← body ++ VALIDATECANARY(canary)

```

said block, giving the code a single unique exit point. Finally, we append a canary validation postamble to the function.

Algorithm 1 presents the instrumentation of a given function in more detail. Line 2 generates a random 8-byte canary value. We use eight bytes because it is the largest primitive value supported by WebAssembly, and we use a random value to reduce the probability of missing a buffer overflow that coincidentally matches the canary. Line 3 prepends the canary injection code to the function body. A template of the injected code is shown in Figure 6. The code is parametrized by the canary value <CANARY> and by <SP>, the index of the global WebAssembly variable that holds the stack pointer. In all WASI applications, the stack pointer is the first global variable, i.e., <SP> is 0. For non-WASI applications, heuristics to identify the stack pointer can be used [24]. The code reserves space for the canary on the stack in linear memory (line 1–4, 16 bytes due to stack alignment) and stores the canary value there, i.e., at the beginning of the stack frame (line 5–7).

The fact that a WebAssembly function may return from multiple locations raises the question where to insert code to validate the canary. Unlike in native code, there is no single function epilogue that clears the stack and returns to the caller. One possible approach is to separately instrument every return instruction with a copy of the validation code, but this would increase code size considerably. Instead, we first rewrite the function to return from a single location and then

```

1 global.get <SP>    ;; Reserve stack space for canary value.
2 i32.const 16      ;; WASI has 16-byte stack alignment.
3 i32.sub
4 global.set <SP>
5 global.get <SP>    ;; Store canary at beginning of stack frame.
6 i64.const <CANARY>
7 i64.store

```

Figure 6: Template of INJECTCANARY for Algorithm 1.

```

1 block    ;; Original return value is at top of operand stack.
2 global.get <SP>    ;; Compare canary value against reference.
3 i64.load
4 i64.const <CANARY>
5 i64.eq
6 br_if 0        ;; Jump out of block if correct.
7 unreachable    ;; Otherwise: Overflow detected!
8 end
9 global.get <SP>    ;; Adjust stack pointer.
10 i32.const 16
11 i32.add
12 global.set <SP>
13 return

```

Figure 7: Template of VALIDATECANARY for Algorithm 1.

insert the validation code once there. First, the entire function is wrapped into a new WebAssembly block (line 4 of Algorithm 1). Then, each return instruction in the function body is replaced with a jump to the end of the new block (lines 6–12), keeping the semantics of the original code. For that, the depth variable (line 5) needs to keep track of the number of nested blocks around the current instruction.

Finally, line 13 appends a postamble for validating the canary before the function returns. Figure 7 shows the template of this code. Since all return instructions were rewritten to jump to the end of the wrapped body, the function return value will reside at the top of the WebAssembly operand stack once the postamble starts executing. This return value is untouched by the canary validation code in lines 1–6 of Figure 7, avoiding the introduction of a fresh local. The stack pointer global at this point refers to the original functions stack base, which after our injection is the memory location used for the canary. This value is loaded from memory and compared against the known, correct canary value. If they differ, an unreachable trap is triggered, which halts execution and thwarts potential exploits. The trap location also indicates that a buffer overflow occurred that crosses the stack frame boundary of the function and its caller. If the canary was intact, lines 9–12 deallocate the canary by adjusting the stack pointer. Finally, line 13 returns the only value left on the operand stack from line 1: the untouched function return value.

## 4.2 Heap Canaries

Detecting and preventing memory violations on the heap is important as well. In WebAssembly, this is especially critical, because binaries frequently ship with allocators that are optimized for code size, and thus lack security features such as safe unlinking [30]. To illustrate the problem, Figure 8a

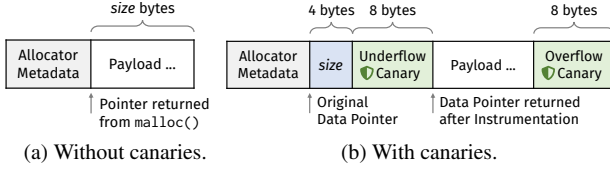


Figure 8: Heap chunks, before and after the instrumentation.

shows the typical layout of a *heap chunk*, i.e., a region of dynamically allocated memory returned by functions like `malloc`. The *payload* is where the user will read and write data to. The *metadata* precedes (or follows) the payload and is used for bookkeeping by the allocator. If an attacker over- or underflows a buffer in the payload and writes into adjacent metadata, this can yield a dangerous arbitrary write primitive, which is much more powerful than a linear overflow [5, 27].

To detect such violations on the heap, Fuzzm instruments heap allocation and deallocation functions in the binary. Our instrumentation inserts canary values before and after the payload, as illustrated by Figure 8b, enabling us to detect both overflows and underflows of the payload. The canaries are inserted into the heap chunk by instrumented versions of allocation functions (Section 4.2.1) and checked by instrumented versions of deallocation functions (Section 4.2.2).

#### 4.2.1 Insert Canaries on Heap Allocation

Heap canaries are inserted by instrumenting all functions that directly allocate heap chunks. Our current implementation instruments the allocation functions from the C standard library, i.e., `malloc`, `calloc`, and `realloc`. Other functions that allocate by calling the low-level libc functions in turn thus profit from our protection as well, as is commonly the case, e.g., for operator `new` in the C++ standard library.

The instrumentation inserts code into allocation functions in two places: a *preamble* in the beginning and a *postamble* at the end, as outlined in Algorithm 2. The added code has three high-level goals. First, the allocation size needs to be increased in order to fit the canaries (line 5). Second, the canary values must be written to memory (line 9). Third, the data pointer returned by the allocator needs to be adjusted before passing it to the user, such that it points to the now shifted payload (line 10). Additionally, we add two new locals to the function (lines 2 and 6) to save data (lines 4 and 8) used later by the inserted code.

Effectively, our canary code is interposed between the original allocator and the client code requesting the allocation, and must be transparent to both. From the allocator’s point of view, the payload is the whole region after the metadata in Figure 8b, including the inserted canaries. For the client code requesting the allocation, the payload is only the region between the canaries, matching the originally requested size. Both are unaware of the data inserted by our added code.

**Details.** In the preamble, we retrieve the originally requested allocation size, save it to a local for later (line 4),

#### Algorithm 2 Instrumentation of heap allocation functions.

```

1: procedure INSTRUMENTALLOCFUNCTION(f)
2:   localreq_size ← ADDFRESHLOCAL(f)
3:   f.body ← ▷ Insert preamble
4:     SAVEALLOCREQUESTSIZE(f, localreq_size) ++
5:     INCREASEALLOCsize(f, localreq_size) ++ f.body
6:   localdata_ptr ← ADDFRESHLOCAL(f)
7:   f.body ← f.body ++ ▷ Insert postamble
8:     SAVEDATAPOINTER(localdata_ptr) ++
9:     WRITESIZEANDCANARIES(localreq_size, localdata_ptr) ++
10:    ADJUSTDATAPOINTER(localdata_ptr)

```

and increase it by 20 bytes (line 5). The additional 20 bytes make space for two 8-byte canaries and a 4-byte size field. The latter is required for the checking code (Section 4.2.2). The exact inserted preamble instructions depend on the allocator function. For `malloc(size_t)` and `realloc(void*, size_t)`, `SAVEALLOCREQUESTSIZE` simply retrieves the first (second) function argument with `local`.get and stores it into a new local. `INCREASEALLOCsize` is just an addition and sets the first (second) function argument to the new value.

Correctly instrumenting `calloc(size_t nitems, size_t item_size)` is a bit more challenging. As the allocation size is the product of both arguments and neither has to be multiple of 20, the inserted preamble changes the arguments to

$$\text{nitems}_{\text{new}} = 1, \text{ and}$$

$$\text{item\_size}_{\text{new}} = \text{nitems} \times \text{item\_size} + 20$$

Additionally, `INCREASEALLOCsize` checks that the second expression does not result in an integer overflow to ensure that the instrumentation never introduces errors into the program.

After the preamble, the original allocator code performs the regular memory allocation routine. Then follows our inserted postamble, shown in Figure 9. Since the postamble executes after the original body of the allocation function, the top-most element on the operand stack is the original return value, i.e., the pointer to the newly allocated memory. This is saved to a local (line 1). Then, the chunk size and underflow canary are stored before the payload (lines 2–7). The overflow canary is stored after the payload (lines 8–12). Finally, the data pointer is fetched and adjusted to point past the underflow canary (lines 13–15). This value is finally returned to the calling code. The result of this instrumentation is that memory allocation functions create chunks as shown in Figure 8b.

#### 4.2.2 Check Canaries on Heap Deallocation

Fuzzm checks whether the heap canaries are valid whenever a heap chunk gets deallocated. Similar to the allocation functions, the approach requires the list of deallocation functions, and our implementation currently supports the functions provided by the C standard library, i.e., `free` and `realloc`.

The heap canaries are validated by the code in Figure 10, which our approach inserts at the beginning of every deallocation function. The argument to the function is a pointer to

```

1  local.set <DATA_PTR> ;; Save pointer returned by allocator.
2  local.get <DATA_PTR> ;; \
3  local.get <REQ_SIZE> ;; | Write requested allocation size
4  i32.store ;; / at the beginning (data_ptr).
5  local.get <DATA_PTR> ;; \
6  i64.const <CANARY> ;; | Write underflow canary
7  i64.store offset=4 ;; / at data_ptr + 4.
8  local.get <DATA_PTR> ;; \
9  local.get <REQ_SIZE> ;; | Write overflow canary
10 i32.add ;; | at data_ptr + size + 12.
11 i64.const <CANARY> ;; |
12 i64.store offset=12 ;; /
13 local.get <DATA_PTR> ;; \
14 i32.const 12 ;; | Adjust returned pointer to payload.
15 i32.add ;; /

```

Figure 9: Template of the inserted postamble in Algorithm 2.

```

1  local.get <PARAM> ;; The argument passed to, e.g., free().
2  i32.const 12 ;; Adjust the pointer before passing
3  i32.sub ;; it to the allocator.
4  local.set <PARAM>
5  block ;; Check underflow canary.
6    local.get <PARAM>
7    i64.load offset=4
8    i64.const <CANARY>
9    i64.eq
10   br_if 0
11   unreachable ;; Underflow detected!
12 end
13 block ;; Check overflow canary.
14   local.get <PARAM> ;; \ Load payload size from our own
15   i32.load ;; / metadata at beginning of chunk.
16   local.get <PARAM> ;; \
17   i32.add ;; | Load overflow canary from
18   i64.load offset=12 ;; | data_ptr + size + 12.
19   i64.const <CANARY> ;; /
20   i64.eq
21   br_if 0
22   unreachable ;; Overflow detected!
23 end

```

Figure 10: Preamble injected into heap deallocation functions to validate heap canaries.

the payload of the heap chunk. To make the canaries transparent to the deallocation function, this pointer needs to be decremented in the beginning (lines 1–4). Lines 5–12 validate the underflow canary, and lines 13–23 validate the overflow canary. The approach uses the size stored during the canary injection to compute the location of the overflow canary.

When to check the heap canaries is a trade-off between performance, complexity of the instrumentation, and the likelihood of catching buffer overflows. Fuzzm performs this check during deallocation, which is inexpensive, as every canary is checked at most once, but has the disadvantage of not catching overflows in chunks that are never deallocated. Others have proposed more aggressive techniques that check canaries at every memory read or write [50] or validate canaries at every syscall [42]. While these approaches may detect more attacks in production, they also impose a larger runtime overhead, which makes them less suited for binary hardening and fuzzing.

## 5 Binary-Only Fuzzing for WebAssembly

This section presents the first binary-only fuzzer for WebAssembly. We take a greybox fuzzing approach and build upon the popular AFL framework, enabling us to reuse its effective input generation abilities. Because AFL usually targets programs with source code available and does not support WebAssembly, there are two key challenges to address. The first challenge is gathering AFL-compatible coverage information during the execution of a WebAssembly program. Section 5.1 describes how Fuzzm addresses this challenge via a novel static instrumentation of WebAssembly binaries. The second challenge is how to integrate executions of WebAssembly on a VM with the existing AFL framework in a way that allows for performing hundreds of executions of a program within a second, which is the level of efficiency AFL provides for natively compiled code. Section 5.2 describes how Fuzzm addresses this challenge through a set of novel techniques that connect AFL to WebAssembly.

### 5.1 Coverage Instrumentation

Greybox fuzzing is effective because it relies on lightweight feedback during program execution to steer the fuzzer. To collect that feedback, native AFL compiles applications from source, inserting code to track an approximate form of path coverage [9], which is stored into a *trace bits* array. Unlike native AFL, we fuzz WebAssembly binaries without access to their source code, and hence, cannot instrument during compilation. AFL also offers a QEMU mode for dynamic binary instrumentation, but it comes with a high performance overhead and naturally is architecture-specific, offering no WebAssembly implementation. Instead, Fuzzm gathers coverage via static binary instrumentation that inserts code at all branches to extract AFL-compatible coverage information. Hence, instrumentation cost is a one-time effort.

As a prerequisite for instrumentation, the approach determines all branches. The structured control flow of WebAssembly enables Fuzzm to precisely identify all branching points in a program. Algorithm 3 summarizes this step, traversing each function of a binary and marking instructions that correspond to branches. This includes `br_if` (line 12), but also `if`, `else` and `loop` blocks (line 9) since they also correspond to a branch. Furthermore, the algorithm keeps track of the depth of each instruction, i.e., depth is incremented at `block`, `if` and `loop` instructions (line 7) and decremented at an `end` instruction (line 19). Keeping track of the depth is necessary to compute which end blocks are targets of branches. Whenever the algorithm encounters a conditional break (either `br_if` on line 10 or `br_table` on line 13), it adds the target depth of the branch instruction to the targets set. At every end instruction, the algorithm then checks whether the depth of that end instruction is in the targets set (line 15). In case it is present, the end is a target of some branch, and is therefore also marked for instrumentation (line 17). In addition to

**Algorithm 3** Insertion of AFL coverage instrumentation.

```

1: procedure AFL_INSTRUMENT_FUNCTION(f)
2:   depth  $\leftarrow$  0
3:   targets  $\leftarrow$  {}
4:   for instr in f.body do
5:     if instr  $\in$  {block, if, else, loop} then
6:       if instr  $\in$  {block, if, loop} then
7:         depth  $\leftarrow$  depth + 1
8:       if instr  $\in$  {if, else, loop} then
9:         mark(instr)
10:    else if instr = br_if n then
11:      targets  $\leftarrow$  targets  $\cup$  {depth - n}
12:      mark(instr)
13:    else if instr = br_table(jmp_targets) then
14:      targets  $\leftarrow$  targets  $\cup \bigcup_{t \in \text{jmp\_targets}} \{t - n\}$ 
15:    else if instr = end then
16:      if depth  $\in$  targets then
17:        mark(instr)
18:        targets  $\leftarrow$  targets  $\setminus$  {depth}
19:      depth  $\leftarrow$  depth - 1
20:  mark(f.body[0])

```

---

```

1  i32.const <CUR_LOCATION>           ;; Id of current branch.
2  global.get <PREV_LOCATION>         ;; Id of previous branch.
3  i32.xor
4  global.get <TRACE_BITS>
5  i32.add
6  local.tee $1                       ;; Set local without pop.
7  local.get 1
8  i32.load8_u                        ;; Unsigned load of 1 byte.
9  i32.const 1
10 i32.add
11 i32.store8                         ;; Store counter in trace_bits.
12 i32.const <CUR_LOCATION> >> 1>    ;; Shift right once.
13 global.set <PREV_LOCATION>

```

Figure 11: AFL-style coverage instrumentation in Wasm.

the instructions marked by Algorithm 3, Fuzzm also marks the beginning of every function (line 20) since an indirect function call also represents a branch.

Given the branching points identified by Algorithm 2, Fuzzm inserts instrumentation code into each of them. A template of the instrumentation code is shown in Figure 11. It adapts the coverage mechanism described in the AFL documentation<sup>6</sup> to WebAssembly. The basic idea is to maintain a global array of counters, the trace bits array, that indicates how often each consecutive pair of branches has been taken. Every branch target is assigned a random identifier. Whenever a branch is taken, the instrumentation code computes an index that combines the identifier of the current branch, <CUR\_LOCATION>, and the identifier of the previous branch, <PREV\_LOCATION> (lines 1–3). The code then increments the corresponding index of the trace bits array (lines 4–11). To initialize the trace bits array, Fuzzm also injects code into the `_start` function of the binary, which is the WASI entry point.

<sup>6</sup>[https://lcamtuf.coredump.cx/afl/technical\\_details.txt](https://lcamtuf.coredump.cx/afl/technical_details.txt)

## 5.2 Integrating a WebAssembly VM and AFL

The native version of AFL is heavily optimized towards performing as many executions of the target program within a given time period as possible. The following presents several novel techniques that allow Fuzzm to achieve a similar level of efficiency. Our implementation targets WebAssembly binaries using the WASI syscall interface, i.e., applications running on a compliant VM, such as Wasmer<sup>7</sup> or Wasmtime.

**Avoiding VM restarts.** With the fuzzed program running on a VM, one possible approach is to start a new instance of the VM for each run of the target program. However, doing so may easily result in more time spend on starting the VM and compiling the module to native code than on running the target program. Instead, Fuzzm starts the VM once, lets the VM precompile the target WebAssembly binary, and then reuses both throughout the fuzzing process. Fuzzm uses the Wasmtime C API<sup>8</sup> to separately compile, then instantiate, and finally run WebAssembly modules. For every newly generated input, the fuzzer instantiates the WebAssembly module that was already compiled to native code and then calls the `_start` function, i.e., the entry point of the target binary.

**Accessing the trace bits array.** To generate new inputs, AFL needs to access the coverage information stored in the trace bits array. The native version of AFL starts the target program as a subprocess and accesses the trace bits array via shared memory. Instead, Fuzzm exploits the fact that the VM sandbox allows for the target program and AFL’s own code to share a single address space. To this end, our approach inserts an accessor function into the binary during the coverage instrumentation. This function returns a pointer to the trace bits array in the linear memory of the target program. After each execution of the target program, Fuzzm calls the special function and extracts the 64KB of linear memory, corresponding to the trace bits, using the Wasmtime C API.

**Detecting crashes.** The native version of AFL detects crashes by looking for fatal signals (SIGSEGV, SIGKILL, SIGABRT) in the target program. However, WASI does not support signals, so Fuzzm uses the trap system of WebAssembly to determine when the target program crashes. To this end, the oracles that Fuzzm inserts (Section 4) trigger an unreachable trap when they detect an overflow or underflow. In addition, WebAssembly has other built-in traps that also indicate faulty behavior (Section 2). When the `_start` function terminates, Fuzzm checks if the termination was triggered by a trap, and in that case, marks it as a crash.<sup>9</sup>

**Killing long-running executions.** Randomly generated inputs may trigger long-running or even non-terminating executions. To prevent those from slowing down overall fuzzing, native AFL runs the fuzzed program in a separate process, which

<sup>7</sup><https://wasmer.io/>

<sup>8</sup><https://docs.wasmtime.dev/c-api/>

<sup>9</sup>Programs terminating with a non-zero exit code also trigger a trap in WASI, but Fuzzm takes care to not interpret these specific traps as crashes, as they do not indicate faulty behavior as crashes in native programs do.



is killed after a timeout. However, since the WebAssembly VM and generated code are running in the same process as AFL, Fuzzm implements two mechanisms to stop long-running executions. First, it uses a “soft killing” mechanism based on a separate thread running on the WASI VM that interrupts the thread of the target program after a timeout dynamically set by AFL. Second, to address that the target program may not react to the interrupt, a second “hard killing” mechanism may restart the entire VM after a longer timeout.

## 6 Evaluation

We evaluate Fuzzm along the two use-cases we present in Section 3. First, *end-to-end fuzzing* of WebAssembly binaries:

**RQ1 Effectiveness:** How effective is Fuzzm at covering paths and finding crashes?

**RQ2 Robustness:** How robust is the instrumentation when applied to real-world binaries?

**RQ3 Efficiency:** How efficient is fuzzing with Fuzzm?

Second, *hardening binaries for production* through canaries:

**RQ4 Effectiveness:** How effective are the inserted canaries at preventing previously demonstrated exploits?

**RQ5 Efficiency:** How much overhead do the canaries impose?

For reproducibility, future research, and practitioners, we make our source code, data, and all experimental results available at <https://github.com/fuzzm/fuzzm-project>.

### 6.1 Experimental Setup

**Benchmarks.** We use three sets of benchmarks (Table 1). Benchmarks 1 to 7 are real-world applications and libraries that can be compiled to WebAssembly with WASI. The selected versions of those programs suffer from known memory vulnerabilities, which a fuzzer might help to uncover and fix.

Benchmarks 8 to 10 are from the LAVA-M benchmark [17]. We omit the *who* program in LAVA-M since it reads the list of mounted file systems, which is not yet supported by WASI. AFL, and by extension also Fuzzm, is known to perform poorly on LAVA-M as the fuzzer does not handle the multi-byte constraints of LAVA-M bugs well [47]. Because LAVA-M has been criticized for not being representative of real bugs [28], we would have liked to instead evaluate against the more modern Magma benchmark suite [22]. However, all the Magma benchmarks use features not yet supported by WASI, such as threads, networking, and long jumps.

Benchmarks 11 to 28 are real-world WebAssembly binaries, gathered from public websites, GitHub, and NPM packages by the WasmBench dataset [24]. We select 18 binaries that run without error (before any instrumentation) in the Wasmtime VM. Among them are large applications, such as SQLite and Clang compiled to WebAssembly, but also several smaller binaries, such as a JSON formatter (*canonicaljson*), a template engine (*handlebars-cli*), and an interpreter (*bfi*).

**Compilation.** We compile the source code from the first and second set using a version of Clang that targets WebAssembly<sup>10</sup> and then instrument the binaries as described in Sections 4 and 5. For comparing against native AFL, we compile the benchmarks with the AFL version of GCC. Since AFL’s instrumentation is applied during compilation, this is not completely true to our scenario where source code is not available and binaries are compiled for production. To level the ground, we do not use AddressSanitizer or any other oracle that requires source code, neither for Fuzzm nor AFL. An alternative baseline would be the QEMU mode of AFL, which uses dynamic instrumentation, but since it is much slower than normal AFL, it would give Fuzzm an unfair advantage. For the third benchmark set, we do not have any source code, which highlights the need for a binary-only fuzzer.

**Repetitions and system configuration.** We repeatedly fuzz each benchmark five times for 24 hours, both with Fuzzm and AFL. The repetitions address the variance of results due to the inherent non-determinism of fuzzing [28]. In addition to the mean results across the repetitions, we report 95% confidence intervals. All experiments were performed on two machines, each with two Intel Xeon 12-core 24-thread CPUs running at 2.2 GHz, using 256 GB of system memory, and Ubuntu 18.04 LTS. For AFL, we use version 2.57b.

### 6.2 RQ1: Effectiveness of Fuzzm

We evaluate the end-to-end effectiveness of fuzzing WebAssembly binaries with Fuzzm by measuring how many unique paths are explored, how many unique crashes are triggered, and whether the canary instrumentation helps in finding those crashes. Table 1 gives the results, where the numbers for Fuzzm are presented in the left block. Crashes and paths are counted using AFL’s notion of unique crashes and unique paths, respectively, i.e., two crashes are merged if they are found on the same path. Different crashes as per this metric may sometimes have the same root cause [28].

We find that Fuzzm successfully explores many hundreds of paths through the programs, on average 1,232 unique paths per benchmark after 24 hours of fuzzing. We see that this works even for complex programs such as *flac* (benchmark set 1) or *sqlite* (set 3). For benchmark sets one and two, where the fuzzed programs are known, we provide the fuzzer with seed inputs. For set three, we only provided an empty file as seed input, which could explain the lower average number of discovered paths. In terms of crashes, Fuzzm finds 40.3 crashes per benchmark on average. For example, for *libpng* and *pdfresurrect*, Fuzzm generates crashing inputs that produce the exact stack trace of proof-of-concept exploits against the vulnerabilities, confirming that Fuzzm can find real bugs.

To better understand how Fuzzm exercises a program over a 24-hour period of fuzzing, Figure 12 shows the number of unique paths detected over time, for four programs. As

<sup>10</sup><https://github.com/WebAssembly/wasi-sdk>

Table 1: Benchmarks overview and fuzzing results ( $5 \times 24$  hours). The reported numbers are mean and 95% confidence intervals.

#	Benchmark	Fuzzm (WebAssembly binaries)								AFL (native, built from source)							
		Paths		Crashes, of those: caused by Canaries				Execs/sec		Paths		Crashes		Execs/sec			
				Total	Stack Can.	Heap Can.											
Benchmark set 1 – Real-world applications and libraries:																	
1	abc2mtex	1678.6±	22.5	267.9±	6.6	224.5±	5.5	4.2±	1.6	359.4±	42.4	2999.2±	82.6	820.1±	63.2	879.6±	212.1
2	flac	607.5±	11.5	0.0±	0.0	0.0±	0.0	0.0±	0.0	497.0±	7.1	1617.1±	75.1	0.0±	0.0	1228.1±	391.3
3	jbig2dec	2199.1±	13.8	0.1±	0.2	0.0±	0.0	0.0±	0.0	66.2±	51.6	3330.1±	49.8	0.0±	0.0	437.7±	264.8
4	libpng	727.0±	10.4	96.1±	4.0	0.0±	0.0	77.2±	3.8	430.9±	67.6	1123.4±	25.3	176.4±	2.8	692.5±	481.6
5	libtiff	860.3±	9.1	0.0±	0.0	0.0±	0.0	0.0±	0.0	868.5±	64.3	2542.5±	33.6	0.0±	0.0	953.7±	467.8
6	openjpeg	5322.2±	3611.0	90.3±	39.3	7.7±	4.5	8.6±	10.4	457.3±	242.8	1779.7±	39.8	90.7±	3.4	605.4±	435.4
7	pdfresurrect	840.1±	207.0	54.5±	8.4	15.1±	3.4	17.2±	5.4	228.1±	194.1	1011.0±	226.3	129.9±	29.2	701.5±	369.2
Benchmark set 2 – From LAVA-M [17]:																	
8	base64	200.6±	7.2	34.4±	1.5	0.0±	0.0	0.0±	0.0	225.8±	83.7	355.8±	29.1	0.1±	0.2	514.3±	276.6
9	md5sum	395.2±	20.6	0.0±	0.0	0.0±	0.0	0.0±	0.0	324.6±	202.7	317.9±	8.9	0.0±	0.0	202.4±	60.7
10	uniq	213.1±	22.8	0.0±	0.0	0.0±	0.0	0.0±	0.0	678.2±	109.6	113.0±	3.7	0.3±	0.4	415.0±	337.5
Benchmark set 3 – Real-world WebAssembly binaries from WasmBench [24]:																	
11	bf	271.4±	27.5	0.0±	0.0	0.0±	0.0	0.0±	0.0	28.6±	7.4	N/A  (As those samples are binary-only WebAssembly programs, there is no native counterpart to fuzz with AFL.)					
12	bfi	2158.0±	108.8	97.8±	26.1	0.0±	0.0	30.8±	12.5	286.4±	40.1						
13	canonicaljson	357.4±	15.8	180.4±	6.9	0.0±	0.0	0.0±	0.0	428.2±	193.4						
14	clang	6.0±	0.6	0.0±	0.0	0.0±	0.0	0.0±	0.0	36.8±	0.7						
15	colcrt	231.0±	7.6	0.0±	0.0	0.0±	0.0	0.0±	0.0	83.6±	50.3						
16	handlebars-cli	882.2±	69.5	39.4±	0.7	0.0±	0.0	39.4±	0.7	222.0±	136.8						
17	hq9_plus_rs	227.0±	15.5	42.8±	0.9	0.0±	0.0	42.8±	0.9	111.0±	42.0						
18	jq	1.0±	0.0	0.0±	0.0	0.0±	0.0	0.0±	0.0	334.0±	7.3						
19	libxml2	177.8±	5.0	0.0±	0.0	0.0±	0.0	0.0±	0.0	70.0±	1.5						
20	qjs	9640.0±	96.0	140.4±	61.4	3.4±	3.0	11.6±	8.6	387.0±	20.0						
21	qr2text	1.0±	0.0	0.0±	0.0	0.0±	0.0	0.0±	0.0	84.6±	0.4						
22	rev	161.4±	6.1	0.0±	0.0	0.0±	0.0	0.0±	0.0	140.4±	20.1						
23	save	23.2±	0.7	0.0±	0.0	0.0±	0.0	0.0±	0.0	46.4±	1.2						
24	sqlite	4284.2±	738.3	2.4±	4.2	0.0±	0.0	0.0±	0.0	359.4±	116.5						
25	viu	12.4±	1.8	0.0±	0.0	0.0±	0.0	0.0±	0.0	707.2±	7.6						
26	wasi-example	213.4±	7.1	50.2±	0.9	0.0±	0.0	0.0±	0.0	764.4±	81.8						
27	wasm-interface	5.2±	0.4	0.0±	0.0	0.0±	0.0	0.0±	0.0	622.2±	10.3						
28	zxing_barcode	2799.2±	236.2	32.2±	5.6	0.0±	0.0	17.4±	5.2	131.4±	62.2						
Average (only sets 1 and 2)		1304.4		54.3		24.7		10.7		413.6		1518.9		121.7		663.0	
Average (all)		1232.0		40.3		9.0		8.9		320.7							

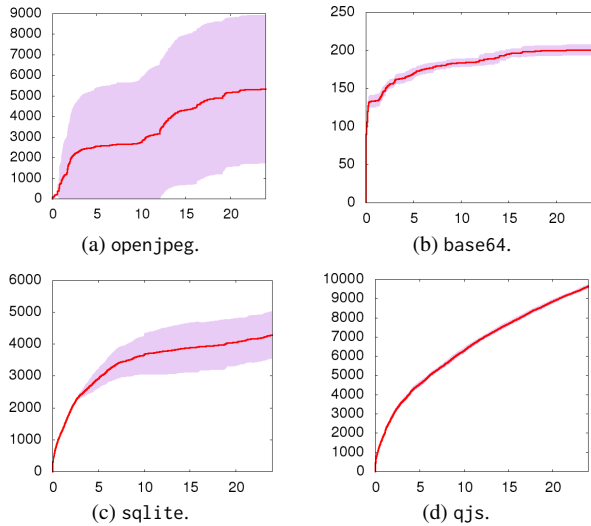


Figure 12: Average unique discovered paths (y-axis) over 24 hours of fuzzing (x-axis), with 95% confidence intervals.

the plots for found crashes strongly correlate with explored paths, we omit the former for space reasons. (Both plots for all programs are available online.) As is typical for fuzzers, the majority of behaviors are detected early on, usually within the first couple of hours (12a/b/c). Then, the number of new paths and crashes often saturates, especially for the LAVA-M benchmarks (b). For some benchmarks, e.g., *qjs*, Fuzzm still finds new paths when given more time (d). The confidence intervals are generally small, except for *openjpeg* (a) and *pdfresurrect*, where the results vary considerably across runs. Overall, these findings are consistent with previous work, and show that running a fuzzer multiple times is important to obtain statistically meaningful results [28].

**Comparison.** As Fuzzm is the first binary-only fuzzing approach for WebAssembly, we cannot directly compare to any baseline. However, to put the number of paths and crashes into perspective, we also present results for native AFL on the right side of Table 1. This is only meant as a rough frame of reference, as a fair comparison is impossible for several reasons. First, Fuzzm requires only the binary as input, whereas AFL

applies its instrumentation during compilation from source. Second, our notion of branches may differ from branches considered by AFL, simply due to different compilers and their target-dependent optimizations and codegen. Third, unlike in native binaries, all libraries (including *libc*) are statically linked in WebAssembly, which increases the amount of code Fuzzm instruments and thus has to fuzz. Fourth, the number of explored paths naturally depends on the execution speed, which is in principle lower on WebAssembly compared to native (see also Section 6.4). Finally, benchmark set three is only available as WebAssembly binaries, which is why we cannot compare against AFL for these benchmarks.

From the data on benchmark sets one and two, we can see that Fuzzm discovers on average a similar number of paths compared with AFL (1304 vs. 1519). In terms of crashes, AFL triggers 122 on average, which is roughly twice as many as Fuzzm’s 54. One outlier is *base64* where Fuzzm triggers 34 unique crashes but AFL triggers only one crash in one of the performed runs. The 34 WebAssembly crashes are triggered by a built-in sanity check of the executing VM, which is not present in native binaries and explains why these crashes are not detected by AFL. For programs where Fuzzm does not find any crashes (e.g., *flac*), AFL does not either.

For the LAVA-M benchmarks, both Fuzzm and AFL fail to trigger any of the bugs injected by the LAVA tool. This observation is surprising since other papers that compare with AFL report at least some bugs detected for *uniq* and also sometimes for *base64* [10, 60]. Manually investigating some of the LAVA-M bugs shows that they resemble use-after-free bugs and that a crash depends on the memory allocator allocating a new chunk at the exact location of some previously freed chunk. We attribute the fact that neither Fuzzm nor AFL finds these bugs to differences in (versions of) the used memory allocator and to differences across versions of AFL.

**Effectiveness of Canaries.** Besides being the first approach for fuzzing WebAssembly binaries, Fuzzm contributes canary-based oracles to detect stack and heap over- and underflows. We measure how much these oracles contribute to the crashes detected by Fuzzm by distinguishing crashes caused by the oracles from other crashes. The three “Crashes” columns of Table 1 show the results. The stack and heap canaries are responsible for 22.2% and 22.0% of all detected crashes, respectively, on all benchmarks, and 45.5% and 19.7% on benchmark sets one and two. This indicates that both contribute significantly to the effectiveness of Fuzzm.

**Summary:** Applied to well-known applications, libraries, and real-world WebAssembly binaries, Fuzzm triggers an average of 40 unique crashes and 1,232 unique paths within 24 hours of fuzzing, which are similar results as AFL applied to native programs. Our canary-based oracles detect about half of all detected crashes, and hence, contribute significantly to the effectiveness of Fuzzm.

### 6.3 RQ2: Robustness of Instrumentation

To effectively fuzz a program, our instrumentation should not affect the semantics of the program, except in the presence of overflows, where the canaries should terminate the program. To validate the robustness of Fuzzm’s instrumentation, we compare the output generated by the non-instrumented and the instrumented versions of the benchmarks. For each benchmark in the first two sets, we collect at least ten different inputs, totaling 138 test cases (Table 2). We sample these inputs from different websites<sup>11</sup>, and for programs where we could not find sufficiently many examples online, e.g., *pal2rgb*, we generate inputs by, e.g., converting images to the pal format. As benchmark set three is only available in binary form without source code or documentation, we do not generate test inputs for those programs. For the binaries and test inputs shown in Table 2, we verify that the outputs produced by the instrumented binaries are equivalent to the outputs produced by the uninstrumented binaries for all 138 test cases. As additional evidence for the robustness of our instrumentation, we find that all binaries we instrumented pass built-in WebAssembly validation, which performs, e.g., type-checking of instructions and functions.

**Summary:** Test runs of the benchmarks and the static validation applied to each WebAssembly module before its execution show that the binary instrumentation applied by Fuzzm preserves the semantics of the original program.

### 6.4 RQ3: Efficiency of End-to-End Fuzzing

Effective fuzzing requires many repeated executions of the target program in limited time. This also applies to Fuzzm, where fast execution is even more challenging due to WebAssembly being a bytecode language and applying our instrumentations at the binary level.

Table 1 lists the average program executions per second during fuzzing in the column “Execs/sec”. With an average speed of 321 executions per second, Fuzzm is able to quickly explore many paths. As already described in Section 6.2, comparisons between Fuzzm and AFL are possible in broad strokes only. This is especially true for performance, because even uninstrumented WebAssembly binaries can execute on average up to 50% slower than native code [25]. Despite this, on benchmark sets one and two, Fuzzm achieves 414 executions per second on average, which is only 37% slower compared with 663 native executions per second in AFL. We believe this is fast enough for practical fuzzing of WebAssembly binaries and respectable, given execution of the target program in a VM. Finally, as improvements to Wasmtime are orthogonal to our approach, further optimizations of the young VM might also speed up Fuzzm in the future.

Besides the program execution in a VM, other sources of slowdown in Fuzzm can come from the applied binary instrumentation. To evaluate the runtime overhead of the added

<sup>11</sup>E.g., <https://filesamples.com/>.

Table 2: Robustness and runtime overhead of instrumented binaries (mean over 25 repetitions, 95% confidence intervals).

#	Benchmark	Test Inputs	Execution Time (ms), Uninstrumented	Execution Time, relative to Uninstrumented Binary				
				Coverage	Stack Canaries	Heap Canaries	All Canaries	Cov. + Can.
1	abc2mtex	30	815	1.38 ± 0.03	1.02 ± 0.01	1.02 ± 0.01	1.06 ± 0.01	1.38 ± 0.04
2	flac	10	2,449	1.42 ± 0.01	1.02 ± 0.01	1.00 ± 0.01	1.02 ± 0.01	1.48 ± 0.02
3	jbig2dec	28	4,742	2.05 ± 0.01	1.22 ± 0.01	1.00 ± 0.00	1.22 ± 0.01	2.24 ± 0.01
4	libpng	10	3,480	1.57 ± 0.02	1.03 ± 0.01	1.00 ± 0.01	1.02 ± 0.01	1.58 ± 0.02
5	libtiff	10	899	1.30 ± 0.03	1.13 ± 0.01	1.11 ± 0.01	1.14 ± 0.01	1.40 ± 0.03
6	openjpeg	10	4,750	1.77 ± 0.02	1.05 ± 0.01	1.00 ± 0.01	1.06 ± 0.01	1.84 ± 0.02
7	pdfresurrect	10	2,894	1.16 ± 0.02	1.06 ± 0.01	1.31 ± 0.01	1.35 ± 0.01	1.53 ± 0.02
8	base64	10	256	1.32 ± 0.10	1.02 ± 0.02	0.99 ± 0.01	1.03 ± 0.02	1.31 ± 0.09
9	md5sum	10	272	1.33 ± 0.09	1.03 ± 0.03	1.00 ± 0.01	1.05 ± 0.02	1.30 ± 0.09
10	uniq	10	260	1.34 ± 0.09	1.04 ± 0.03	1.02 ± 0.01	1.11 ± 0.09	1.39 ± 0.10
Average			2,082	1.46 ± 0.04	1.06 ± 0.02	1.05 ± 0.01	1.11 ± 0.02	1.54 ± 0.04

code, we run the benchmark programs with the test inputs used for RQ3, and compare the runtime of the original, uninstrumented binaries against the runtime when the binaries were instrumented. The results are shown in the right part of Table 2. On average over 25 program executions, the coverage instrumentation imposes a runtime overhead of 1.46x over the uninstrumented binary. We will detail the overhead of the canaries in Section 6.6. The overhead of the coverage instrumentation is generally higher than for the canaries, because for every branch in the program it adds 13 instructions (Figure 11). More efficient implementations, e.g., by predicting some branches based on static analysis [7], could further reduce the overhead, which we leave for future work. The last column of Table 2 shows the combined overhead of both the canary instrumentation and the coverage instrumentation.

**Summary:** Fuzzm performs hundreds of program executions per second, which is only 37% slower than native AFL, despite executing the program in a VM. The coverage instrumentation imposes an average overhead of 1.46x, which dominates the overall overhead imposed by Fuzzm’s instrumentation.

## 6.5 RQ4: Effectiveness of the Canaries in Preventing Exploitation

In the previous research questions, we have analyzed Fuzzm as an end-to-end WebAssembly fuzzer. The canary instrumentations from Section 4 are, however, also useful in a stand-alone setting, namely for catching memory errors in production binaries to prevent exploitation. To evaluate this scenario, we apply our canary instrumentation to three previously published, vulnerable WebAssembly applications with proof-of-concept exploits [30]. The applications use WebAssembly in three different settings: on a website in the browser, on Node.js, and a command-line application for standalone WASI VMs. Since the canary instrumentation is platform-independent, we can harden binaries in all three settings. The proof-of-concept inputs exploit two buffer overflow vulnerabilities on the stack, and one buffer overflow on the heap that writes into allocator metadata. We confirm that the

uninstrumented, original WebAssembly binaries can be exploited, which causes cross-site scripting, executes code, and writes to an unintended file, respectively. Then, we successfully instrument all three binaries, without requiring access to the source code or their build process. When given correct and benign inputs, those three instrumented binaries work as before, but when passing the exploit inputs, all three examples are successfully terminated by the inserted canary checks.

**Summary:** The stack and heap canaries inserted by our binary-only instrumentation effectively hardens existing binaries and protects against previously demonstrated exploits.

## 6.6 RQ5: Efficiency of the Inserted Canaries

As demonstrated in the previous section, the inserted canaries can mitigate buffer overflow attacks when applied to existing binaries. For this usage scenario, it is essential that the canaries have only a minimal impact on performance. We evaluate their efficiency by running the benchmark programs with and without instrumentation on the inputs from RQ3. The results are in Table 2, which shows the execution times with different combinations of canaries relative to the execution time of the uninstrumented programs.

Both the stack and heap canary instrumentation only slightly impact performance, with an average execution time of 1.06x and 1.05x relative to the uninstrumented binary. The stack canary overhead is similar to efficient implementations of canaries for native binaries [15], which are employed by default in common compilers (Clang, GCC, MSVC). Some applications, e.g., *jbig2dec* with an execution time of 1.22x, are impacted significantly more than others, e.g., *flac*, where the overhead is negligible. The relative cost of heap canaries depends on the number of memory allocations, especially small ones. While *pdfresurrect*, an analyzer of PDF files, stands out due its frequent allocations, the overhead for the other applications is low or even too small to measure. The overhead with both canary instrumentations applied (column “All Canaries”) is approximately the combined overhead of the individual ones, imposing a moderate overhead of 1.11x.



**Summary:** The overhead imposed by the canary-based oracles is small (1.06x and 1.05x, respectively) and comparable to canary implementations for other languages, which is encouraging for their use to harden production binaries.

## 7 Related Work

**Fuzzing.** Out of the many approaches for greybox fuzzing [10, 12, 22, 28, 35], we build on the popular AFL fuzzer. Unlike its commonly used GCC and LLVM modes, and the majority of other fuzzers [6, 32, 44, 47, 57], we do not require source code access, and hence, also cannot rely on compiler-added oracles [43, 50]. Improvements to the input generation algorithm of AFL [34] will also benefit Fuzzm.

There are several ways to fuzz native binaries. AFL’s QEMU and DynInst modes rely on dynamic instrumentation, which incurs substantial runtime overhead. Dinesh et al. [16] propose static instrumentation of x86-64 binaries for fuzzing, but they cannot handle WebAssembly binaries due to the different architecture and also make several assumptions that do not apply in our setting, e.g., position-independent code, which does not exist in WebAssembly, or the presence of relocation information, which we do not require. Other recent work is about binary instrumentation for coverage [39], but does not provide an oracle instrumentation similar to ours.

We know of one approach for fuzzing WebAssembly<sup>12</sup>, which is a port of LibFuzzer<sup>13</sup>. They require the source code and only support C and C++ code compiled with Emscripten, which then runs in a browser. Instead, Fuzzm is the first to fuzz WebAssembly binaries, with support for WASI applications. Fuzzing has also been used for testing WebAssembly VMs<sup>14</sup>, which is orthogonal to fuzzing WebAssembly programs.

**Binary rewriting and overflow protection.** Statically instrumenting native binaries is challenging due to the undecidability of disassembly [54], with challenges like data inlined in code, variable-length instructions, resolution of indirect jumps, and identification of functions [4]. Many existing tools rely on the symbol table for recovering function boundaries [11, 16, 53]. WebAssembly does not suffer from these problems, a situation we use to our benefit in Fuzzm.

Several papers have presented binary rewriting techniques for protecting the stack in x86 programs [11, 16, 42, 53]. BodyArmor inserts instrumentation that monitors reads and writes relative to pointers [53]. Another technique combines a randomized layout, isolation, and secure allocation for hardening binaries against stack-based vulnerabilities [11]. RetroWrite [16] uses an overflow detection mechanism similar to AddressSanitizer [50], i.e., using shadow memory to mark bytes where access is illegal. These previous three approaches all rely on the symbol table being available.

<sup>12</sup><https://github.com/jonathanmetzman/wasm-fuzzing-demo>

<sup>13</sup><https://llvm.org/docs/LibFuzzer.html>

<sup>14</sup><https://github.com/wasmerio/wasmer/tree/master/fuzz> and <https://github.com/bytecodealliance/wasmtime/tree/main/fuzz>

Prasad et al. use a shadow stack for finding buffer overflows reaching across stack frame boundaries, without requiring access to the symbol table [45]. Their technique is similar to the Fuzzm canaries in terms of the granularity of detected bugs. Using a shadow stack does not affect relative references on the stack, and is thus preferable to canaries on x86 where relative references appear. They suffer from potential false positives and false negatives as most other x86 techniques.

There has also been work on techniques for protecting binaries against heap overflows [16, 42, 48]. Robertson et al. [48] and Nikiforakis et al. [42] both present techniques that, like the Fuzzm heap canaries, instrument the allocation and deallocation functions such that they insert canaries. Nikiforakis et al. check the canaries at system calls, which means it is likely that overflows are detected early, but at the cost of having to check the canaries often. We instead opted for the more efficient option of checking canaries during deallocation, to enable efficient fuzzing and low-overhead hardening.

**WebAssembly.** Several papers examine the security of WebAssembly applications and found that exploits, which are no longer possible in native binaries, may still affect WebAssembly [18, 30]. WebAssembly was initially used heavily for malicious cryptomining [29, 38, 49], but it has recently been shown that today’s WebAssembly binaries are rarely malicious [24]. While WebAssembly is designed to achieve near native speed, it has been shown to still be around 50% slower [26], which explains the performance difference between Fuzzm and AFL. The Wasabi framework allows for creating dynamic analyses for WebAssembly easily [31], but Fuzzm uses its own instrumentation for efficiency.

## 8 Conclusion

WebAssembly programs are becoming more and more prevalent, which increases the need for techniques that can uncover security problems. This paper presents Fuzzm, the first binary-only greybox fuzzer for WebAssembly. The approach combines canary-based binary instrumentation to detect overflows and underflows on the stack and the heap, an efficient coverage instrumentation, a WebAssembly VM running the program, and the input generation algorithm of native AFL. Unlike most other efficient fuzzers, Fuzzm works directly on production binaries, without requiring access to the source code. We show that Fuzzm finds a substantial amount of crashes in real-world WebAssembly binaries, while being efficient enough to perform hundreds of executions per second, even though WebAssembly is a slower, non-native language. Besides as oracles for fuzzing, the canaries also serve as a stand-alone binary hardening technique to prevent exploitation of vulnerable binaries in production. In this scenario, the approach prevents previously published exploits while imposing low overhead. Overall, our work is an important step toward securing the increasingly popular WebAssembly platform against exploitation of memory-related vulnerabilities.

## Acknowledgments

This work was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

## References

- [1] Wasmer – The Universal WebAssembly Runtime. <https://wasmer.io/>, 2019.
- [2] Wasmtime – A small and efficient runtime for WebAssembly & WASI. <https://wasmtime.dev/>, 2020.
- [3] OSS-Fuzz. <https://google.github.io/oss-fuzz/>, 2021.
- [4] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In Thorsten Holz and Stefan Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pages 583–600. USENIX Association, 2016.
- [5] Anonymous. Once upon a free. *Phrack*, 11(9), November 2001.
- [6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.
- [7] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Efficient binary-level coverage analysis. In *FSE*, 2020.
- [8] John Bergbom. Memory safety: old vulnerabilities become new with WebAssembly. <https://www.forcepoint.com/sites/default/files/resources/files/report-web-assembly-memory-safety-en.pdf>, 2018.
- [9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Trans. Software Eng.*, 45(5):489–506, 2019.
- [10] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 711–725. IEEE Computer Society, 2018.
- [11] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.
- [12] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. Patrix: Efficient hardware-assisted fuzzing for COTS binary. *CoRR*, abs/1905.10499, 2019.
- [13] Lin Clark. Standardizing WASI: A system interface to run WebAssembly outside the web. <https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/>, 2019.
- [14] Crispian Cowan. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Aviel D. Rubin, editor, *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*. USENIX Association, 1998.
- [15] Thurston H. Y. Dang, Petros Maniatis, and David A. Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 555–566. ACM, 2015.
- [16] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1497–1511. IEEE, 2020.
- [17] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 110–121. IEEE Computer Society, 2016.
- [18] Brian McFadden Tyler Lukasiewicz Jeff Dileo Justin Engler. Security chasms of WASM. In *NCC Group Whitepaper*. NCC Group, 2018.
- [19] Dongsoo Ha, Wenhui Jin, and Heekuck Oh. REPICA: rewriting position independent code of ARM. *IEEE Access*, 6:50488–50509, 2018.
- [20] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.
- [21] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI '19*, page

- 225–236, New York, NY, USA, 2019. Association for Computing Machinery.
- [22] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *CoRR*, abs/2009.01120, 2020.
  - [23] Ningyu He, Ruiyi Zhang, Lei Wu, Haoyu Wang, Xipu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. Security analysis of eosio smart contracts. *arXiv preprint arXiv:2003.06568*, 2020.
  - [24] Aaron Hilbig, Daniel Lehman, and Michael Pradel. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *The Web Conference 2021 (WWW '21)*, 2021.
  - [25] Abhinav Jangda, Bobby Powers, Emery Berger, and Arjun Guha. Not so fast: Analyzing the performance of webassembly vs. native code. *login Usenix Mag.*, 44(3), 2019.
  - [26] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: Analyzing the performance of webassembly vs. native code. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 107–120, 2019.
  - [27] Michel Kaempf. Vudo – An object superstitiously believed to embody magical powers. *Phrack*, 11(8), November 2001.
  - [28] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 2123–2138. ACM, 2018.
  - [29] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1714–1730. ACM, 2018.
  - [30] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of web-assembly. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 217–234. USENIX Association, 2020.
  - [31] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing WebAssembly. In *ASPLOS*, 2019.
  - [32] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 627–637, 2017.
  - [33] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 190–200. ACM, 2005.
  - [34] Dominik Maier, Heiko Eißfeldt, Andrea Fioraldi, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020.
  - [35] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.
  - [36] Timothy McCallum. Diving into Ethereum’s Virtual Machine (EVM): the future of Ewasm. <https://hackernoon.com/diving-into-ethereums-virtual-machine-the-future-of-ewasm-wrk32iy>, 2019.
  - [37] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. Security chasms of wasm. NCC Group Whitepaper, 2018.
  - [38] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. Thieves in the browser: Web-based cryptojacking in the wild. In *Proceedings of the 14th International Conference on Availability, Reliability and Security, ARES 2019, Canterbury, UK, August 26-29, 2019*, pages 4:1–4:10. ACM, 2019.
  - [39] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1683–1700. USENIX Association, August 2021.
  - [40] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 699–716, 2020.

- [41] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 89–100. ACM, 2007.
- [42] Nick Nikiforakis, Frank Piessens, and Wouter Joosen. Heapsentry: Kernel-assisted protection against heap overflows. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA 2013, Berlin, Germany, July 18-19, 2013. Proceedings*, volume 7967 of *Lecture Notes in Computer Science*, pages 177–196. Springer, 2013.
- [43] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided grey-box fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306. USENIX Association, August 2020.
- [44] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.
- [45] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference, June 9-14, 2003, San Antonio, Texas, USA*, pages 211–224. USENIX, 2003.
- [46] Lijin Quan, Lei Wu, and Haoyu Wang. EVulHunter: detecting fake transfer vulnerabilities for EOSIO’s smart contracts at Webassembly-level. *arXiv preprint arXiv:1906.10362*, 2019.
- [47] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.
- [48] William K. Robertson, Christopher Krügel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th Conference on Systems Administration (LISA 2003), San Diego, California, USA, October 26-31, 2003*, pages 51–60. USENIX, 2003.
- [49] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. Digging into browser-based crypto mining. In *Proceedings of the Internet Measurement Conference 2018, IMC 2018, Boston, MA, USA, October 31 - November 02, 2018*, pages 70–76. ACM, 2018.
- [50] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 309–318. USENIX Association, 2012.
- [51] Kostya Serebryany. Oss-fuzz-google’s continuous fuzzing service for open source software. *USENIX Security*, 2017.
- [52] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.
- [53] Asia Slowinska, Traian Stancescu, and Herbert Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 125–137. USENIX Association, 2012.
- [54] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Differentiating code from data in x86 binaries. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part III*, volume 6913 of *Lecture Notes in Computer Science*, pages 522–536. Springer, 2011.
- [55] Conrad Watt. Mechanising and verifying the web-assembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs*, pages 53–65, 2018.
- [56] WebAssembly Community Group. WebAssembly Specification. <https://webassembly.github.io/spec/core/>, 2021.
- [57] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 745–761. USENIX Association, 2018.
- [58] Alon Zakai. WasmBoxC: Simple, Easy, and Fast VM-less Sandboxing. <https://kripken.github.io/blog/wasm/2020/07/27/wasmbxc.html>, 2020.
- [59] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book. <https://www.fuzzingbook.org/>, 2019.
- [60] Bin Zhang, Jiaxi Ye, Chao Feng, and Chaojing Tang. S2F: discover hard-to-reach vulnerabilities by semi-symbolic fuzz testing. In *13th International Conference on Computational Intelligence and Security, CIS 2017, Hong Kong, China, December 15-18, 2017*, pages 548–552. IEEE Computer Society, 2017.