Everything Old is New Again: Binary Security of WebAssembly

Daniel Lehmann, *University of Stuttgart;* Johannes Kinder, *Bundeswehr University Munich;* Michael Pradel, *University of Stuttgart*

https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.

Everything Old is New Again: Binary Security of WebAssembly

Daniel Lehmann
University of Stuttgart

Johannes Kinder
Bundeswehr University Munich

Michael Pradel
University of Stuttgart

Abstract

WebAssembly is an increasingly popular compilation target designed to run code in browsers and on other platforms safely and securely, by strictly separating code and data, enforcing types, and limiting indirect control flow. Still, vulnerabilities in memory-unsafe source languages can translate to vulnerabilities in WebAssembly binaries. In this paper, we analyze to what extent vulnerabilities are exploitable in WebAssembly binaries, and how this compares to native code. We find that many classic vulnerabilities which, due to common mitigations, are no longer exploitable in native binaries, are completely exposed in WebAssembly. Moreover, WebAssembly enables unique attacks, such as overwriting supposedly constant data or manipulating the heap using a stack overflow. We present a set of attack primitives that enable an attacker (i) to write arbitrary memory, (ii) to overwrite sensitive data, and (iii) to trigger unexpected behavior by diverting control flow or manipulating the host environment. We provide a set of vulnerable proof-of-concept applications along with complete end-to-end exploits, which cover three WebAssembly platforms. An empirical risk assessment on real-world binaries and SPEC CPU programs compiled to WebAssembly shows that our attack primitives are likely to be feasible in practice. Overall, our findings show a perhaps surprising lack of binary security in WebAssembly. We discuss potential protection mechanisms to mitigate the resulting risks.

1 Introduction

WebAssembly is an increasingly popular bytecode language that offers a compact and portable representation, fast execution, and a low-level memory model [32]. Announced in 2015 [19] and implemented by all major browsers in 2017 [65], WebAssembly is supported by 92% of all global browser installations as of June 2020.[1] The language is designed as a compilation target, and several widely used compilers exist, e.g., Emscripten for C and C++, or the Rust compiler, both based on LLVM. Originally devised for client-side computation in browsers, WebAssembly's simplicity and generality has sparked interest to use it as a platform for many other domains, e.g., on the server side in conjunction with Node.js, for "serverless" cloud computing [33–35, 64], Internet of Things and embedded devices [31], smart contracts [44, 53], or even as a standalone runtime [4, 23]. WebAssembly and its ecosystem, although still evolving, have already gathered significant momentum and will be an important computing platform for years to come.

WebAssembly is often touted for its safety and security. For example, both the initial publication [32] and the official website [12] highlight security on the first page. Indeed, in WebAssembly's core application domains, security is paramount: on the client side, users run untrusted code from websites in their browser; on the server side in Node.js, WebAssembly modules operate on untrusted inputs from clients; in cloud computing, providers run untrusted code from users; and in smart contracts, programs may handle large sums of money.

There are two main aspects to the security of the WebAssembly ecosystem: (i) *host security*, the effectiveness of the runtime environment in protecting the host system against malicious WebAssembly code; and (ii) *binary security*, the effectiveness of the built-in fault isolation mechanisms in preventing exploitation of otherwise benign WebAssembly code. Attacks against host security rely on implementation bugs [16, 59] and therefore are typically specific to a given virtual machine (VM). Attacks against binary security—the focus of this paper—are specific to each WebAssembly program and its compiler toolchain. The design of WebAssembly includes various features to ensure binary security. For example, the memory maintained by a WebAssembly program is separated from its code, the execution stack, and the data structures of the underlying VM. To prevent type-related crashes and attacks, binaries are designed to be easily type-checked, which they are statically before execution. Moreoever, WebAssembly programs can only jump to designated code locations, a form of fault isolation that prevents many classic control flow attacks.

[1] https://caniuse.com/#search=WebAssembly

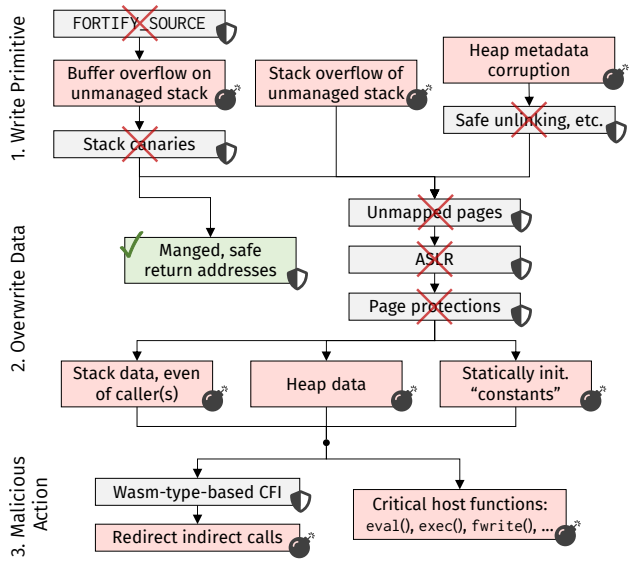


Figure 1: An overview of attack primitives (💣) and (missing) defenses (🛡) in WebAssembly, later detailed in this paper.

Despite all these features, the fact that WebAssembly is designed as a compilation target for languages with manual memory management, such as C and C++, raises a question: *To what extent do memory vulnerabilities affect the security of WebAssembly binaries?* The original WebAssembly paper addresses this question briefly by saying that "at worst, a buggy or exploited WebAssembly program can make a mess of the data in its own memory" [32]. A WebAssembly design document on security [1] concludes: "common mitigations such as data execution prevention (DEP) and stack smashing protection (SSP) are not needed by WebAssembly programs."

This paper analyzes to what extent WebAssembly binaries can be exploited and demonstrates that the above answers miss important security risks. Comparing the exploitability of WebAssembly binaries with native binaries, e.g., on x86, shows that WebAssembly re-enables several formerly defeated attacks because it lacks modern mitigations. One example are stack-based buffer overflows, which are effective again because WebAssembly binaries do not deploy stack canaries. Moreover, we find attacks not possible in this form in native binaries, such as overwriting string literals in supposedly constant memory. If such manipulated data is later interpreted by critical host functions, e.g., as JavaScript code, this can lead to further system compromise. Our work mostly focuses on binaries compiled with LLVM-based compilers, such as Emscripten and Clang for C and C++ code, or the Rust compiler, since they are currently the most popular compilers targeting WebAssembly.

After our analysis of the deployed (and missing) security features in WebAssembly, we take the position of an active adversary and identify a set of attack primitives that can later be used to build end-to-end exploits. Our attack primitives span three dimensions: (i) obtaining a write primitive, i.e., the ability to write memory locations in violation of source-level semantics; (ii) overwriting security-relevant data, e.g., constants or data on the stack and heap; and (iii) triggering a malicious action by diverging control flow or manipulating the host environment. Figure 1 provides an overview of the attack primitives and defenses discussed.

To show that our attack primitives are applicable in practice, we then discuss a set of vulnerable example WebAssembly applications and demonstrate end-to-end exploits against each one of them. The attacked applications cover three different kinds of platforms that support WebAssembly: browser-based web applications, server-side applications on Node.js, and applications for stand-alone WebAssembly VMs.

In our quantitative evaluation, we then estimate the feasibility of attacks against other binaries. We collect a set of binaries from real-world web applications and compiled from large C and C++ programs of the SPEC CPU benchmark suite. Regarding data-based attacks, we find that one third of all functions make use of the unmanaged (and unprotected) stack in linear memory. Regarding control-flow attacks, we find that every second function can be reached from indirect calls that take their target directly from linear memory. We also compare WebAssembly's type-checking of indirect calls with native control-flow integrity defenses.

Our work improves upon initial discussions of WebAssembly binary security in the non-academic community [20, 25, 28, 45] by providing a systematic analysis, a generalization of attacks, and data on real binaries (see Section 8 for a more detailed comparison).

Contributions In summary, this paper contributes:

- An in-depth *security analysis of WebAssembly's linear memory* and its use by programs compiled from languages such as C, C++, and Rust, which common memory protections are missing from WebAssembly, and how this can make some code less secure than when compiled to a native binary (Section 3).
- A set of *attack primitives*, derived from our analysis and generalized from previous work, along with a discussion of mitigations that the WebAssembly ecosystem does, or does not, provide (Section 4).
- A set of *example vulnerable applications* and *end-to-end exploits*, which show the consequences of our attacks on three different WebAssembly platforms (Section 5).
- *Empirical evidence* that both data and control-flow attacks are likely to be feasible, measured on WebAssembly binaries from real-world web applications and compiled from large C and C++ programs (Section 6).
- A discussion of possible *mitigations* to harden WebAssembly binaries against the described attacks (Section 7). We make our attack primitives, end-to-end exploits, and analysis tool publicly available[2] to aid in this process.

[2] https://github.com/sola-st/wasm-binary-security

2 Background on WebAssembly

Since WebAssembly is still relatively new, we briefly give an introduction to its syntax, execution model, and the ecosystem. More comprehensive information is available in the official documentation and specification [12, 14].

Overview WebAssembly is a binary format. The binaries are designed to be compact and quick to parse. Unlike for x86, static disassembly is simple and reliable. A human-readable, exact text representation of binaries exists, called `wat`. Figure 2 shows a simple WebAssembly program. One module corresponds to one file. A module contains functions, globals, and at most one linear memory and indirect call table. Program elements, such as functions or locals, are identified by integer indices. For convenience, indices can be written as `$name` in the text format, but those labels are lost in the binary.

WebAssembly bytecode is executed on a stack-based virtual machine. Instructions pop their inputs from and push their results to the implicit evaluation stack. There are no registers. Individual values can be stored in an unlimited number of global variables, whose scope is the entire module, and local variables, which are only visible to the current function. Functions cannot access local variables or the evaluation stack of other functions, also not of their caller or callees. The evaluation stack, globals, and locals are managed by the VM.

Types Unlike in most native architectures, WebAssembly globals, locals, and the arguments and results of functions and instructions are typed. Binaries are statically type-checked before being executed. There are four primitive types: 32 and 64 bit integers (`i32`, `i64`) and single and double precision floats (`f32`, `f64`). More complex types, such as arrays, records, or designated pointers do not exist. Source-level types are thus lowered to these primitive types during compilation.

Control-Flow Unlike native code or Java bytecode, WebAssembly has only structured control-flow. Instructions in a function are organized into well-nested blocks. Branches can only jump to the end of surrounding blocks, and only inside the current function. Multi-way branches can only target blocks that are statically designated in a branch table. Unrestricted gotos or jumps to arbitrary addresses are not possible. In particular, one cannot execute data in memory as bytecode instructions. Many classical attacks are thus ruled out in WebAssembly, e.g., injecting shellcode or abusing unrestricted indirect jumps, e.g., `jmp *%reg` in x86.

Indirect Calls To implement function pointers and virtual functions, WebAssembly has indirect calls. Figure 3 illustrates how they work. The `call_indirect` instruction on the left pops a value from the stack, which it uses to index into the so called `table` section. Table entries map this index to a function, which is subsequently called. Thus, a function can only be indirectly called if it is present in the table. Functions

```
(module
  ;; Import function from host environment.
  (import "print" (func $print (param i32)))
  ;; Global variable, 32-bit integer, initialized to 42.
  (global $g i32 (i32.const 42))
  ;; Function in the binary with type [i32] -> [i64].
  (func $f (param $arg i32) (result i64)
    (local $var i32) ;; Declaration of a local variable.
    i32.const 8      ;; Push constant on stack.
    local.get $arg   ;; Copy function argument to stack.
    i32.add          ;; Pop inputs from stack, push result.
    local.tee $var   ;; Copy result to local variable.
    if               ;; Is top == 0?
      i32.const 1024   ;; Pointer to string in memory.
      call $print      ;; Call imported function.
    end              ;; Structured control-flow.
    local.get $var   ;; Push local value as address for...
    i64.load         ;; ...8 byte read from linear memory.
  )
  ;; Explicitly initialized memory at offset 1024.
  (data (i32.const 1024) "some string\00"))
```

Figure 2: Example of a WebAssembly binary, represented in the (slightly simplified) text format.

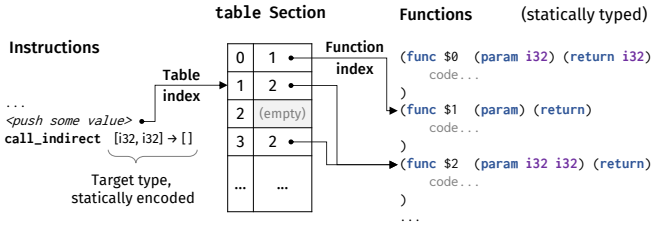


Figure 3: Indirect function calls via the `table` section.

can be referenced multiple times in the table and not every entry in the table must be filled. To ensure type-correctness, the VM checks before executing the call that the target function is type-compatible with the statically declared type in the indirect call instruction and aborts execution otherwise.

Linear, Unmanaged Memory In contrast to other bytecode languages, WebAssembly does not provide managed memory or garbage collection. Instead, the so called *linear memory* is simply a single, global array of bytes. Load and store instructions can access arbitrary addresses within the currently allocated memory. The memory is addressed by 32-bit pointers, and `i32` serves as the pointer type. A WebAssembly program can request the VM to increase the linear memory with the `memory.grow` instruction. For efficient dynamic memory allocation, a WebAssembly program typically includes its own allocator, which manages the linear memory, e.g., by providing `malloc` and `free` to the program.

Host Environment WebAssembly modules are executed in a host environment. Without the host environment, WebAssembly programs cannot, for example, perform I/O or access the network. Instead, such functionality is provided by the

host through functions that can be imported by the WebAssembly module. In browsers, all APIs available to JavaScript-based client-side web applications can be imported, such as `XmlHttpRequest`, `eval`, or `document.write`. Other host environments are also emerging, e.g., Node.js for server-side applications, and stand-alone VMs, which provide their own APIs to WebAssembly modules. For example, modules running in Node.js may invoke `exec` to execute shell commands, and modules running on a stand-alone VM may interact with the local file system through the WebAssembly system interface (WASI) [9]. Non-primitive data, e.g., strings or objects, must be passed between host and WebAssembly module through linear memory, which can be accessed by both.

Compilers and Tooling As a low-level bytecode, WebAssembly is a compilation target for higher-level programming languages. There are several compilers for different languages, e.g., C, C++, Rust, Go, and AssemblyScript, and for different host environments. In addition to the source program, compilers also add their own, host-environment-specific implementation of the standard libraries of the compiled language. For example, when Emscripten compiles C code for the browser, it will add JavaScript implementations such that `printf` outputs to the browser console.

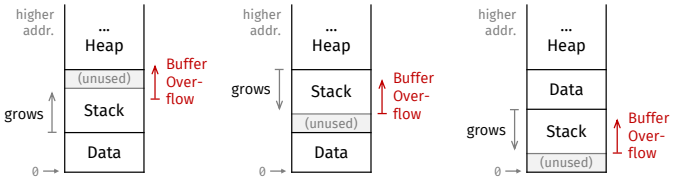
3 Security Analysis of Linear Memory

We now begin our security analysis of WebAssembly binaries and focus first on one of their key components: linear memory. We analyze how compilers arrange program data in linear memory and investigate how and which standard memory protection mechanisms are applied.

3.1 Managed vs. Unmanaged Data

We distinguish managed and unmanaged data in WebAssembly. *Managed* data, i.e., local variables, global variables, values on the evaluation stack, and return addresses, reside in dedicated storage handled directly by the VM. WebAssembly code can only interact with managed data implicitly through instructions, but not directly modify its underlying storage. E.g., `local.get 0` reads local 0, but at no point is the actual, underlying address of the local visible to the program. *Unmanaged* data is all data that resides in linear memory. It is completely under the control of the program and typically organized by compiler-generated code.

There are several reasons for putting unmanaged data in linear memory. Since WebAssembly has only four types and because managed data can hold instances of only those primitive types, all non-scalar data, such as strings, arrays, or lists, must be stored in linear memory. Because managed data has no address, any variable whose address is ever taken in the source program, e.g., out parameters, must also be stored in linear memory. Because many non-scalar types occur in the



(a) `emcc` 1.39.7 (*fastcomp* backend, deprecated). (b) `emcc` 1.39.7 (*upstream* backend), `clang` 9 (WASI). (c) `clang` 9 (WASI with `stack-first`), `rustc` 1.41 (WASI).

Figure 4: WebAssembly linear memory layouts for different compilers and backends.

source program as function-scoped, global, or data with dynamic lifetime, the compiler creates areas for a call stack, a heap, and static data in linear memory. We will refer to the compiler-created call stack in linear memory as the *unmanaged stack* to distinguish it from the managed evaluation stack, which holds intermediate values of instructions, and the managed call stack, which holds locals and return addresses. Importantly, this means a lot of data lies in unmanaged linear memory, not under protection of the VM, but instead under full control of memory write instructions in the program.

3.2 Memory Layout

Native ELF binaries[3] contain sections for zero-initialized data (`.bss`), read- and writable data (`.data`), read-only data (`.rodata`), code (`.text`), a stack, and a heap. The compilers we analyze, Emscripten, Clang, and Rustc, all perform a similar subdivision of the linear memory in WebAssembly binaries (Figure 4). The heap must always be placed at the end of linear memory, such that it can grow towards higher addresses and make use of additional memory when it is requested from the host environment. Below the heap are the stack and static data. Since there is no read-only memory in WebAssembly (more on that in the next section), there is no distinction between `.data` and `.rodata`, and since memory is always zero-initialized, there is no need for a dedicated `.bss` section. In other words, `.data`, `.rodata`, and `.bss` are not explicitly distinguished in WebAssembly. In the following, when we refer to the *data* section in linear memory, we mean all such data that is valid for the whole lifetime of the program, e.g., statically initialized string constants, global arrays, or zero-byte ranges.

The memory layout, i.e., the order of stack, heap, and data in linear memory, depends on the compiler. Figure 4a shows that the *fastcomp* backend of Emscripten (the first WebAssembly backend and thus frequently used until its deprecation in October 2019 [8]) places the static data at the beginning of linear memory, followed by the stack, and then the heap. The stack grows upwards (i.e., towards higher addresses) in this configuration. More recently, LLVM has gained its own,

[3] Other native binary formats, such as PE, have analogous sections, but for readability we compare only with ELF here.

in-tree WebAssembly backend [70], which at the time of writing is used by Emscripten, Clang, and the Rust compiler. That is, in most WebAssembly binaries produced today, the stack grows downwards (similar to ARM and x86). The difference between Figure 4b and 4c is in the relative order of stack and data in linear memory. In Emscripten and Clang, static data comes first by default. In Rust and in Clang with the linker option `-stack-first`, the stack comes first and static data sits between stack and heap.

3.3 Memory Protections

One of the most basic protection mechanisms in native programs is virtual memory with *unmapped pages*. A read or write to an unmapped page triggers a page fault and terminates the program, hence an attacker must avoid writing to such addresses. WebAssembly's *linear* memory, on the other hand, is a single, contiguous memory space without any holes, so every pointer $\in [0, max_mem]$ is valid. As long as the attacker stays within this bound, any read or write will succeed. This is a fundamental limitation of linear memory with severe consequences. Since one cannot install guard pages between static data, the unmanaged stack, and the heap, overflows in one section can silently corrupt data in adjacent sections. Section 4 shows that buffer and stack overflows are thus very powerful attack primitives in WebAssembly.

Virtual memory in native execution also allows to set *page protection flags*, i.e., marking pages exclusively as readable, writable, or executable. In WebAssembly, linear memory is non-executable by design, as it cannot be jumped to. However, WebAssembly does not allow marking memory as read-only; instead, all data in linear memory is always writable. This is another quite surprising limitation of linear memory and enables one of our attack primitives in Section 4.

As an additional probabilistic defense in native execution, *address space layout randomization* (ASLR) [51] randomly arranges the stack, heap, and code in the address space at runtime. For a successful attack, the attacker thus first has to obtain a pointer, e.g., to the heap, via an information disclosure vulnerability. In WebAssembly, there is no ASLR. WebAssembly linear memory is arranged deterministically, i.e., stack and heap positions are predictable from the compiler and program. Even if one were to add some form of ASLR to WebAssembly, linear memory is addressed by 32-bit pointers, which likely does not provide enough entropy for strong protection [58].

4 Attack Primitives

This section presents attack primitives that can be used to exploit vulnerabilities in code compiled to WebAssembly. The attack primitives span three dimensions from which a full attack can be constructed. The first dimension is about obtaining a write primitive, i.e., the ability of an attacker to use a vulnerability for unexpected writes to memory. The second dimension corresponds to the data that can be overwritten. The third dimension is about triggering security-compromising behavior by overwriting data. In principle, the primitives in these three dimensions can be freely combined. For example, a write primitive from the first dimension can overwrite any data from the second dimension to trigger any kind of misbehavior from the third dimension.

Figure 1 gives an overview of the three dimensions of attack primitives (💣) and mitigations designed to counter them (🛡). As discussed in detail in the following, many of the standard mitigations used when compiling to native binaries are unused or unavailable when compiling to WebAssembly (shown by crossing out mitigations). Some of the attack primitives described here are based on existing ideas for exploiting vulnerabilities in C/C++ code compiled to native code. The novelty lies in the way these attacks and existing mititations transfer, or do not transfer, to WebAssembly. Other attack primitives (e.g., Section 4.1.2 and 4.2.3) have never been possible in modern native systems with virtual memory and are presented here for WebAssembly for the first time.

4.1 Obtaining a Write Primitive

Given a WebAssembly binary compiled from vulnerable C or C++ code, there are several ways for an attacker to obtain a write primitive. In particular, we discuss those types of attacks for which there are effective mitigations on native platforms, but not in WebAssembly.[4]

4.1.1 Stack-based Buffer Overflow

Stack-based buffer overflows have been widely exploited [50] and, by now, there exist several mitigation techniques. We show that, contrary to current beliefs, stack-based buffer overflows are exploitable in WebAssembly.

Figure 5 shows C code prone to overflow because line 9 fails to perform bounds checking. Figure 5b shows the stack layout when compiling this code with a modern compiler to x86. The stack contains local variables of the current function (`same_frame` and `buffer`), local variables of parent functions (`parent_frame`), saved registers (if any), and the return address. An overflow of `buffer` could overwrite data on the stack, in particular return addresses. However, modern compilers mitigate this kind of attack in several ways. To detect buffer overflows, compilers place stack canaries (or stack cookies) [24] above local data. To minimize the data that could be overwritten, compilers also reorder local variables on the stack. In many cases, the compiler can also prevent potential buffer overflow vulnerabilities through semantics-preserving code transformations. For example, the `FORTIFY_SOURCE` flag allows

[4] We do not discuss attack primitives that are possible in WebAssembly but neither novel nor specific to this platform. E.g., integer overflows exist in WebAssembly just as they do in x86 or ARM.

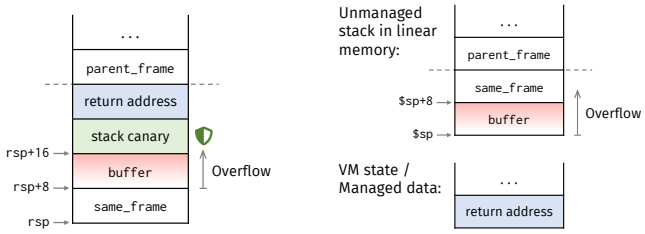
the compiler to replace `strcpy` with `strncpy` if the length of the string is known.

Do stack-based buffer overflows affect WebAssembly? Because the WebAssembly VM isolates managed data, in particular, return addresses, it is tempting to get a strong (and false) sense of security, as illustrated by the quote from WebAssembly's official design document in Section 1. Yet, buffer overflows can compromise data in WebAssembly because parts of the function-scoped data in C is stored on the unmanaged stack in the linear memory (Section 3.1).

Figure 5c illustrates the problem by showing the unmanaged stack in linear memory (top), as well as the internal state of the WebAssembly VM that stores the return addresses of calls (bottom). While the VM-internal state is protected against overwrites by the VM, the unmanaged stack is not. Indeed, an overflow while writing into a local variable on the unmanaged stack, e.g., `buffer`, may overwrite other local variables in the same and even in other stack frames upwards in the stack, e.g., `parent_frame`. Because overflows can also write to data in the parent function (as we show above) and even to other memory sections (as we show later), the primitive is more powerful and the use of stack canaries more important than previously realized [20, 45].

```
1  void parent() {
2    char parent_frame[8] = "BBBBBBBB"; // Also overwritten
3    vulnerable(readline());
4    // Dangerous if parent_frame is passed, e.g., to exec
5  }
6  void vulnerable(char* input) {
7    char same_frame[8] = "AAAAAAAA"; // Can be overwritten
8    char buffer[8];
9    strcpy(buffer, input); // Buffer overflow on the stack
10 }
```

(a) Vulnerable C program, overflowing `buffer` on the stack.



(b) Stack layout on x86-64 with canaries and reordering.

(c) Unmanaged stack and VM state in WebAssembly.

Figure 5: Example of a stack-based buffer overflow and its exploitability in WebAssembly.

4.1.2 Stack Overflow

Another write primitive are stack overflows, which occur due to excessive or infinite recursion or when a local buffer of variable size is allocated on the stack, e.g., using `alloca`. If an attacker controls the size of stack allocations, or provides corrupted input data that violates internal assumptions of recursive functions, she may trigger a stack overflow. For example, recursive implementations of functions operating on trees or lists often assume acyclicity; a cyclic data structure passed to such a function can then lead to infinite recursion.

On most native platforms, stack overflows will cause the program to crash as the stack grows into a special *guard page* that separates the stack from other areas of memory. In WebAssembly, such protections do not exist for the unmanaged stack, so an attacker-controlled stack overflow can be used to overwrite potentially sensitive data following the stack (Section 3.2).

4.1.3 Heap Metadata Corruption

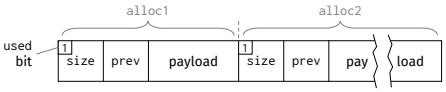
Another primitive an attacker may use to write memory in WebAssembly programs is to corrupt heap metadata of the memory allocator shipped with a WebAssembly binary. Because in WebAssembly no default allocator is provided by the host environment, compilers include a memory allocator as part of the compiled program (Section 2). Since the WebAssembly module typically has to be downloaded from the internet right before execution, the code size of the allocator is an important consideration. The Emscripten compiler therefore lets developers choose between the default allocator, based on `dlmalloc`, and the simplified allocator `emmalloc` that reduces the final code size. Similarly, Rust programs can choose a more lightweight allocator when compiling to WebAssembly, called `wee_alloc`.[5]

While standard allocators, such as `dlmalloc`, have been hardened against a variety of metadata corruption attacks, simplified and lightweight allocators are often vulnerable to classic attacks. We find both `emmalloc` and `wee_alloc` to be vulnerable to metadata corruption attacks, which we illustrate for a version of `emmalloc` in the following.[6]

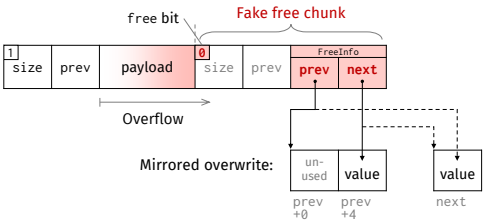
When deallocating a chunk of memory by calling `free`, allocators try to merge as many adjacent free chunks as possible into a single larger one to avoid fragmentation. This gives rise to the classical *unlink exploit* [18, 38] shown in Figure 6. Since `emmalloc` is a *first-fit* allocator, it will return the first chunk in the free list large enough to satisfy an allocation request. Thus, two directly following allocation requests yield two chunks adjacent to each other in memory, such as `alloc1` and `alloc2` in Figure 6a. Lines 1 to 9 of `emmalloc`'s source code in Figure 6c show that the metadata of each chunk starts with a bit indicating whether the current chunk is free or not, the chunk's size, a pointer to the preceding chunk, and finally either the payload (raw bytes) or a `FreeInfo` struct, which in a benign allocation makes that chunk part of a doubly linked list of free chunks.

[5] https://github.com/rustwasm/wee_alloc

[6] Recently, `emmalloc`'s implementation was slightly changed, but it is still vulnerable against this type of attack. We provide an exploit against the newer version as well in our supplementary material.



(a) Heap layout before the overflow: two adjacent chunks.



(b) Heap layout after an overflow of `alloc1`: manipulated metadata causes mirrored write to a chosen location on `free`.

```
1  struct FreeInfo { FreeInfo* prev; FreeInfo* next; };
2  struct Chunk {
3    size_t used : 1; size_t size : 31;
4    Chunk* prev;
5    union {  // Depending on whether the chunk is free or not.
6      char payload[];
7      FreeInfo freeInfo;
8    };
9  };
10 // Called on alloc2, before merging it into alloc1.
11 void removeFromFreeList(Chunk* chunk) {
12   FreeInfo* freeInfo = chunk->freeInfo;
13   freeInfo->prev->next = freeInfo->next; // mirrored
14   freeInfo->next->prev = freeInfo->prev; // write
15 }
```

(c) Excerpt from the `emmalloc` allocator (edited for clarity).

Figure 6: Example of a heap metadata corruption in `emmalloc` after an overflow on the heap.

Given an overflow of data in `alloc1` (e.g., due to a `memcpy` with the wrong length), an attacker can write to the directly adjacent metadata of `alloc2` to clear the `used` bit and set up a "fake" `FreeInfo` struct (Figure 6b). Finally, when `alloc1` is freed, the allocator checks whether there is an opportunity to merge the newly freed chunk with an adjacent free chunk. Because the manipulated metadata identifies the following chunk as free, the allocator calls `removeFromFreeList` to unlink it in preparation for merging the two. In line 13 of Figure 6c, the unlinking code of `emmalloc` then writes the attacker-controlled value of the `next` field into the `next` field of another `FreeInfo` struct (i.e., to an offset of 4 bytes) at the attacker-controlled address in `prev`. This allows the attacker to write an arbitrary value to an arbitrary address. Due to line 14, there additionally is a mirrored write into the location pointed to by `next`. Thus, to avoid a runtime error terminating execution, both `prev` and `next` must be valid pointers. Since Emscripten allocates a default stack size of 5MiB, values below 5×2^{20} can in all likelihood be safely written. This is more than sufficient for overwriting function table indices (see Section 4.3.1), which are at most in the range of thousands.

The above methods for obtaining write primitives are by no means exhaustive, but the most direct methods from the traditional exploit arsenal that currently do not have mitigations in WebAssembly. Other possible attacks may exploit format string vulnerabilities, use-after free and double-free vulnerabilities, single-byte buffer overflows, or perform more sophisticated attacks on memory management.

4.2 Overwriting Data

The second dimension of attack primitives corresponds to the data that can be overwritten with a given write primitive to gain additional control over the execution.

4.2.1 Overwriting Stack Data

The unmanaged stack in linear memory contains function-scoped data, such as arrays, structs or any value that has its address taken. With a given fully-flexible write primitive, an attacker can overwrite any potentially critical local data including function pointers represented as function table indices or arguments to security-critical functions.

In contrast to native code, there are no return addresses on the unmanaged stack. Hence, a purely linear stack-based buffer overflow cannot easily take control of the execution. However, the overflow can reach all currently active call frames if the stack is growing downwards, as it does in most configurations, see Section 3.2. Because there are no return addresses or stack canaries, the overflow can overwrite local data of all calling functions without risking early termination.

4.2.2 Overwriting Heap Data

The heap commonly contains data with longer lifetime and will store complex data structures across different functions. Targeted writes to heap data are straightforward in WebAssembly due to the fully deterministic memory allocation (Section 3.3). To make matters worse, even a linear stack-based buffer overflow of sufficient length can corrupt heap data. The reasons are that the heap comes after the stack in any compiler configuration (Section 3.2) and that no mechanism, such as guard pages, mitigates such attempts.

Note that with a single linear memory, there is no way to avoid the fundamental risk of either stack overflows or stack-based buffer overflows. If the stack grows upwards, a stack overflow can silently corrupt heap data. If the stack grows downwards, stack-based buffer overflows are the culprit.

4.2.3 Overwriting "Constant" Data

The following it the perhaps most surprising target of a data overwrite, as it is impossible in modern native platforms.

Many programming languages allow to protect data from being overwritten by declaring it constant. This is enforced not just by the type system, but also at runtime by placement in read-only memory. As WebAssembly has no way of making data immutable in linear memory, an arbitrary write primitive can change the value of any non-scalar constant in the program, including, e.g., all string literals. Even more restricted write primitives allow modification of constant data: a stack overflow with the memory layout of Figure 4b can write into constant data; similarly, a stack-based buffer overflow can reach constant data in the memory layout of Figure 4c. As a result, an attacker with either of those capabilities can overwrite any supposedly constant data, compromising the guarantees intended by the programming language. We will show two examples of exploits caused by this surprising aspect of WebAssembly linear memory in the next section.

4.3 Triggering Unexpected Behavior

Given a write primitive (Section 4.1) and a choice of data to overwrite (Section 4.2), there are several ways for an attacker to trigger unexpected behavior.

4.3.1 Redirecting Indirect Calls

The closest equivalent of native control-flow attacks in WebAssembly is the redirection of indirect function calls. This type of attack allows for executing code that normally would not be executed in a given context.

In Section 2, we have illustrated indirect function calls in WebAssembly. An attacker may redirect an indirect call by overwriting an integer in linear memory that eventually serves as an index into the `table` section. As described in Section 4.2, this integer value may be a local variable on the unmanaged stack, part of a heap object, in a vtable, or even a supposedly constant value.

WebAssembly has two mechanisms that limit an attacker's ability to redirect indirect calls. First, not all functions defined in or exported into a WebAssembly binary appear in the table for indirect calls, but only those that may be subject to an indirect call. Second, all calls, both direct and indirect, are type checked. As a result, an attacker can redirect calls only within the equivalence class of functions of the same type, similar to type-based control-flow integrity [15]. In Section 6 we measure to what extent these mechanisms reduce the available call targets an attacker can choose from.

4.3.2 Code Injection into Host Environment

WebAssembly modules can interact with their host environment in various ways to cause externally visible effects. One such way is to invoke the notorious `eval` function of a JavaScript host environment, which interprets a given string as code. To access `eval`, WebAssembly modules compiled via Emscripten can use, e.g., `emscripten_run_script`, which executes JavaScript code in the host environment, both in browsers and in Node.js-based server-side code [7]. In browsers, any function that allows to add code to the document (e.g., `document.write`) can serve as an `eval`-equivalent for constructing exploits. In Node.js, the low-level nature of the API gives even more options for code injection, e.g., the `exec` function of the `child_process` module.

Using the primitives described in Section 4.1 and Section 4.2, an attacker may inject malicious code by overwriting the argument passed to an `eval`-like function. For example, suppose a WebAssembly usually invokes `eval` with a "constant" string of code stored in linear memory, then an attacker could overwrite that constant with malicious code.

4.3.3 Application-specific Data Overwrite

Depending on the application, there can be other sensitive targets for data overwrites. For example, a WebAssembly module issuing web requests through an imported function could be made to contact a different host by overwriting the destination string, to initiate cookie stealing. As a further example, several interpreters and runtimes have been compiled to WebAssembly, e.g., to execute CIL/.NET code directly in the browser [5]. These kinds of environments contain many opportunities for significantly altering program behavior, e.g., by overwriting bytecode then interpreted by the runtime.

5 End-to-End Attacks

We now demonstrate several end-to-end attacks that represent different points in the design space of attacks defined by the primitives of Section 4. These attacks substantiate our claim that the current lack of mitigations in the WebAssembly ecosystem enables realistic attack scenarios. We make all described attacks publicly available, providing a benchmark to guide and evaluate future work on hardening WebAssembly.

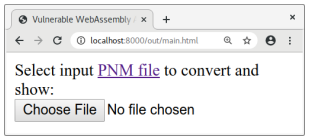
Table 1 gives an overview of the end-to-end attacks. The attacks cover several platforms that support WebAssembly: the browser, where we demonstrate a cross-site scripting attack; Node.js, where we show a remote code execution attack; and stand-alone WebAssembly VMs, such as wasmtime [10], where we show an arbitrary file write attack.

5.1 Cross-Site Scripting in Browsers

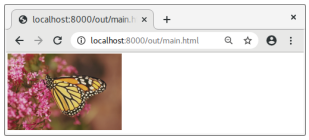
This attack shows that including vulnerable code compiled to WebAssembly into a client-side web application can enable attacks known from JavaScript-based applications, such as cross-site scripting (XSS). As an example, consider an image sharing service where users upload and view images. The service provides a web application that converts images between different formats on the client side, using a version of the *libpng* image codec library compiled to WebAssembly (Figure 7). Given a file to be converted to PNG, the application

§	Host environment	Write primitive	Overwritten data	Location of data	Malicious behavior
5.1	Browsers (client-side)	Stack-based buffer overflow (CVE-2018-14550)	Image tag in DOM string	Heap	Cross-site scripting in JavaScript via `document.write()`
5.2	Node.js (server-side)	Heap metadata corruption	Function index	Stack	Inject arbitrary shell command into `exec()`
5.3	Wasmtime (stand-alone runtime)	Stack-based buffer overflow	String literals	"Constant" data	Write arbitrary content to chosen file using `fprintf()`

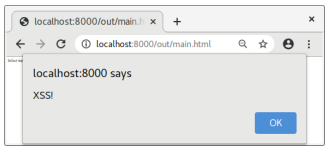
Table 1: Overview of our end-to-end attacks, using different combinations of attack primitives on three host environments.



(a) In the benign case: Select a PNM image and...



(b) ...convert it to PNG with a C library, fully on the client side.



(c) A malicious input can overflow a buffer on the stack, then corrupt a string on the heap, which is later used in DOM manipulation.

```
1 void main() {
2   std::string img_tag = "<img src='";
3   pnm2png("input.pnm", "output.png");  // CVE-2018-14550
4   img_tag += file_to_base64("output.png") + "'>";
5   emcc::global("document").call("write", img_tag);
6 }
```

(d) Excerpt of C++ code (to be compiled with Emscripten) that uses the vulnerable C library.

Figure 7: Example of cross-site scripting caused by using the vulnerable `libpng` library (CVE-2018-14550).

calls libpng and then displays the image by calling a DOM manipulation function, such as `document.write`, provided by the JavaScript host environment.

Version 1.6.35 of libpng suffers from a known buffer overflow vulnerability (CVE-2018-14550 [3]), which can be exploited when converting a PNM file to a PNG file. When the library is compiled to native code with modern compilers on standard settings, stack canaries prevent this vulnerability from being exploited. In WebAssembly, the vulnerability can be exploited unhindered by any mitigations.

To exploit the vulnerability for cross-site scripting, an attacker provides a malicious image to another user who then displays it using the web application. Figure 7d shows a minimal version of such an application. During normal execution, the application converts the image (line 3), encodes it with base64 in a data URL, copies it into an `img` tag (line 4), and then adds the tag into the document (line 5). Since the image is embedded into the DOM as a base64-encoded string, it normally cannot lead to XSS. However, exploiting the stack-based buffer overflow in libpng allows the attacker to overwrite higher addresses, including the heap, which holds the C++ string with the `img` tag (line 2). The attacker can then replace the `img` tag with arbitrary other content, e.g., a `script` tag that displays an alert, which will then get passed to `document.write`.

Depending on how the input data is uploaded, the above scenario can lead to both non-persistent and persistent XSS attacks. In the non-persistent variant, the attacker tricks the user into uploading a malicious image, which then triggers the attack immediately in the user's browser. In the persistent variant, the attacker uploads the malicious input image himself and then shares it with others, who will be attacked once they download the input, and convert it in their browser with the vulnerable WebAssembly application.

5.2 Remote Code Execution in Node.js

In the next attack, we demonstrate that including vulnerable WebAssembly in a Node.js-based application can enable remote code execution. As an example, consider a server that accepts requests to log the ids of customers that have been happy or unhappy about some product. Figure 8b shows an excerpt of the code running in the server application. The `handle_request` function receives three attacker-controlled parameters: `input1`, which describes whether the customer was happy; `input2`, which is supposed to be the length of the string in `input1`; and `input3`, which contains the id of the customer. Depending on the customer's happiness, the code calls `log_happy` or `log_unhappy`, which is selected by assigning the respective function to the function pointer `func`.

The code contains a heap overflow vulnerability at line 9. In the absence of safe unlinking and other mitigations (we use the `emmalloc` allocator for our proof of concept) an attacker can use the overflow to obtain an arbitrary write primitive through the classic heap metadata corruption attack (see Section 4.1.3). If the function pointer `func` is compiled into a variable in linear memory (which is the case, e.g., for all function pointers in vtables), the attacker can use the write primitive to manipulate it and redirect the call (Section 4.3.1). The absence of ASLR simplifies such an attack further, as the address to overwrite is deterministic.

```c
// Functions supposed to be triggered by requests
void log_happy(int customer_id) { /* ... */ }
void log_unhappy(int customer_id) { /* ... */ }

void handle_request(char *input1, int input2, char *input3) {
  void (*func)(int) = NULL;
  char *happiness = malloc(16);
  char *other_allocation = malloc(16);
  memcpy(happiness, input1, input2); // Heap overflow
  if      (happiness[0] == 'h') func = &log_happy;
  else if (happiness[0] == 'u') func = &log_unhappy;
  free(happiness); // Unlink exploit overwrites func
  func(atoi(input3)); // 3rd input is passed as argument
}

// Somewhere else in the binary:
void exec(const char *cmd) { /* ... */ }
```

(a) Sample application that calls one of two logging functions depending on its input. It suffers from a heap overflow, which causes an arbitrary write on `free`, allowing to redirect `func` to `&exec`. Then `input3` can be chosen as the address of an injected string.

```
(func $log_happy   (param i32) (result) ...)
(func $log_unhappy (param i32) (result) ...)
(func $exec        (param i32) (result) ...)
```

(b) Excerpt of the function section for the binary compiled from Figure 8a, showing that `exec`, `log_happy`, and `log_unhappy` all have the same WebAssembly type [`i32`] → [ ].

Figure 8: Example of remote code execution.

One possible target for redirecting the call is the `exec` function that can also be found in the binary (line 17). While `exec` and the `log_*` functions have different C++ types, all three functions have identical types on the WebAssembly level (Figure 8b). The reason is that both integers and pointers are represented as `i32` types in WebAssembly, i.e., the redirected call passes WebAssembly's type check. The final challenge is to pass an arbitrary command into `exec`, which is similar to the injection of shellcode in native exploitation. One option is to inject a suitable command string into the heap when overwriting the function index, and to then pass a decimal string with the address of the command string as `input3`.

5.3 Arbitrary File Write in Stand-alone VM

WebAssembly is starting to establish itself as a universal bytecode beyond web applications. To this end, applications require access to the underlying operating system, e.g., for manipulating files. This interface is currently undergoing standardization as the WebAssembly System Interface (WASI) [9]. There are multiple virtual machines for running stand-alone WebAssembly applications, including wasmtime [10] and WAVM [11], and Clang can compile for them.

This attack demonstrates that, despite stand-alone WebAssembly VMs being advertised as a secure platform for executing C/C++ code, WebAssembly currently enables attacks impossible in modern native execution platforms. Figure 9a

```c
// Write "constant" string into "constant" file
FILE *f = fopen("file.txt", "a");
fprintf(f, "Append constant text.");
fclose(f);

// Somewhere else in the binary:
char buf[32];
scanf("%[^\n]", buf); // Stack-based buffer overflow
```

(a) C program with stack-based buffer overflow that overflows into 'constant' section, causing an arbitrary file write.

```
(data (i32.const 65536) "%[^\0a]\00
                         file.txt\00a\00
                         Append constant text.\00...")
```

(b) Excerpt of the data section for the binary compiled from Figure 9a, showing that the filename literal and contents to be written are located in regular (writable) linear memory.

Figure 9: Example of arbitrary file write.

shows an excerpt of an apparently harmless application that appends a constant string to a statically known file. Somewhere else in the program, the code suffers from a textbook buffer overflow, which enables an attacker to overwrite data on the stack. Compiled to a native target, exploiting the buffer overflow cannot influence the file I/O, which is entirely based on string literals stored in the read-only pages loaded from the `.rodata` section.

When running on a stand-alone WebAssembly VM, this vulnerability can be exploited for an arbitrary file write. The strings for filename and contents are stored in the unmanaged linear memory, as shown in Figure 9b. They can be overwritten by a stack-based buffer overflow of sufficient length if data lies above the stack (see Section 3.2). As a result, the attacker can write arbitrary data into an arbitrary file by overwriting the filename and contents strings. In our exploit, even the file open mode `"a"` (append) is changed to `"w"` by simply overwriting the corresponding string in the data section.

6 Quantitative Evaluation

To better understand how realistic the attacks described so far are in practice, we now present a quantitative evaluation on real-world WebAssembly binaries. We address the following research questions:

RQ1 *How much data is stored on the unmanaged stack?* This question is relevant because the unmanaged stack serves both as an entry point to obtain a write primitive, e.g., via a stack-based buffer overflow, and as a target for overwrites, e.g., to manipulate sensitive data. (Section 6.2)

RQ2 *How common are indirect calls and how many functions can be reached from indirect calls?* These questions are relevant to understand the risk for control-flow divergence by redirecting indirect calls. (Section 6.3)

RQ3 *How does WebAssembly's type checking of indirect call targets compare to current control-flow integrity (CFI) defenses for native binaries?* Since the runtime validation of indirect call targets performed by the WebAssembly VM resembles CFI defenses, we compare both in terms of CFI equivalence classes and class sizes. (Section 6.4)

We make our full dataset and the tools we developed to obtain them available for download (see Section 1).

6.1 Experimental Setup and Analysis Process

Program Corpus The binaries we analyze in our quantitative evaluation are split into two groups. First, we collect a set of 9 binaries from real-world, deployed WebAssembly applications: Adobe's Document Cloud View SDK[7] renders and annotates PDFs in the browser; Figma[8] is a collaborate user-interface design web application; the 1Password X 1.17 browser extension[9] contains a WebAssembly component for password generation; Doom 3 as an example of a large game engine ported to WebAssembly[10]; and finally a set of codecs (webp, mozjpeg, optipng, hqx) for in-browser image conversion[11]. The binaries span different application domains (document editing, games, codecs), deployment scenarios (web application, browser extension), and source languages (C, C++, Rust), so we believe they are a good first approximation of realistic WebAssembly binaries. We collect their most recent versions as of March 2020. Since our tool is open source, we welcome others to replicate our results and extend them by analyzing other WebAssembly binaries.

The second group of binaries in our corpus are 17 C and C++ programs from the SPEC CPU 2017 benchmark suite, compiled to WebAssembly. SPEC CPU has been used before to study the performance of WebAssembly [37]. It has also been used to evaluate the security of CFI techniques for native code [21, 69], enabling us to address RQ3. Those programs are from compute-heavy domains (programming language implementations, simulations, video codecs, compression), matching the original use cases for WebAssembly [13].

Our combined program corpus consists of 26 WebAssembly binaries, which contain 19.2M instructions across 98,924 functions in total. Table 2 gives a more detailed overview.

Toolchain and Configuration We compiled the SPEC CPU programs with Emscripten 1.39.7, i.e., the most recent version at the time of writing, with its *upstream* backend. Since this backend is shared by all LLVM-based WebAssembly compilers (Clang, Rust), our results should translate also to them. For completeness, we also compiled all SPEC CPU programs with the now deprecated *fastcomp* backend of Emscripten. Since fastcomp was the default backend of Emscripten until October 2019, its results are relevant for large amounts of code previously compiled to WebAssembly. The results for fastcomp and upstream are very similar, so for brevity we only present the upstream results in the following.

To obtain optimized binaries without symbols or debug information, we compile with `-O3`. GCC, x264, Blender, and Xalan-C++ required several preprocessor flags for compatability, e.g., to set correct integer bit-widths and platforms. Some programs also had to be manually linked because Emscripten's libc (based on `musl`) causes errors due to duplicate symbol definitions.

Static Analysis To address our research questions, we develop a lightweight static analysis tool. To the best of our knowledge, it is the first security analysis tool for WebAssembly binaries. The analysis is written in Rust and does:

- Extract general information about the program, e.g., instruction counts, number of functions, and their types.
- Analyze the unmanaged stack by inferring which `global` is the stack pointer, which functions access it, and how the stack pointer is incremented and decremented.
- Analyze the `table` section and its static initialization, to find out which functions are present in it, as well as the function type for each initialized table index.
- Analyze indirect call edges to extract the statically encoded type of allowed `call_indirect` targets, how many functions match that type, additional restrictions on the call targets, and CFI equivalence classes and their sizes.

We explain the analyses in more detail in the following.

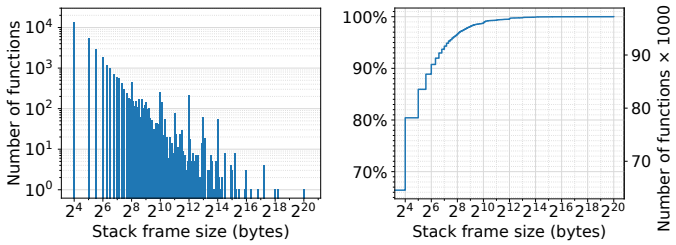
6.2 Measuring Unmanaged Stack Usage

Measuring how much data a program stores on the unmanaged stack (RQ1) is important for two reasons. First, such data could potentially suffer from a stack-based overflow. Second, such data may become subject to overwrites once an attacker has a write primitive. So how much data ends up on the unmanaged and, as we saw earlier, unprotected stack?

Static Analysis Our static analysis measures the size of the stack frame on the unmanaged stack for each non-imported function. The analysis operates on optimized, stripped binaries without debug information, as a realistic attacker would, and thus has to infer the unmanaged stack usage directly from the bytecode.

First, the analysis needs to identify the stack pointer. Unlike in native binaries, there is no convention to use a fixed register (such as `rsp` on x86, which does not exist in WebAssembly) or global variable for the stack pointer. Instead, the analysis extracts all instructions that modify globals and selects the one that is most frequently read and written. A manual analysis confirms that this heuristic reliably finds the stack pointer. From the identified global's static initialization, we also know the base address of the unmanaged stack in linear memory.

[7] https://www.adobe.io/apis/documentcloud/dcsdk/viewsdk.html
[8] https://www.figma.com/
[9] https://1password.com/
[10] http://www.continuation-labs.com/projects/d3wasm/
[11] https://squoosh.app/



(a) Histogram (double logarith.). (b) Cumulative distribution.

Figure 10: Two representations of the distribution of frame sizes on the unmanaged stack for all functions in our program corpus.

Second, for each function, the analysis infers the size of the stack frame on the unmanaged stack. In all analyzed binaries, the previously identified stack pointer is modified in a protocol similar to function prologues and epilogues in native binaries. Specifically, our analysis pattern matches against the following sequence of instructions and extracts the *delta* value, which gives us the stack frame size:

```
1 global.get $i
2 i32.const <delta>
3 i32.add or i32.sub
4 local.tee $j (optional)
5 global.set $i
```

This sequence first reads the current stack pointer from global `$i` (identified earlier), then increments or decrements it (depending on whether the stack grows upwards or downwards, see Section 3.2), optionally saves it to a `local` (akin to a base pointer), and finally writes the modified value back.

Results Figure 10 shows the distribution of stack frame sizes across all analyzed binaries, both as a histogram (Figure 10a) and the cumulative distribution (Figure 10b). One third (32,651) of all functions in the program corpus store some data on the unmanaged stack. The smallest frame size of 16 (2^4) bytes is allocated by 13,620 functions (14% of all functions). Stack frame sizes span the whole range from 16 bytes to 1MiB, which is the largest static stack allocation. The distribution has a long tail towards large stack frames. From the cumulative distribution in Figure 10b, we see that 6% (6,127) of all functions allocate 128 (2^7) bytes or more on the unmanaged stack, and 1.3% (1,232) of all functions allocate at least 1KiB.

Overall, we see that many functions use the unmanaged stack, which is susceptible not only to arbitrary memory writes but also to inter-frame buffer overflows (see Section 4). This implies that with increasing call depth, the chance for an attacker to find at least some data to overwrite increases quickly. For example, with ten nested calls (assuming a uniform distribution of functions), there would be some data on the unmanaged stack with $1-((1-0.33)^{10}) \approx 98.2\%$ probability. We conclude that (1) a lot of stack data is prone to being overwritten by buffer overflows and arbitrary write primitives, and (2) it is important to isolate stack frames on the unmanaged stack, e.g., using stack canaries.

6.3 Measuring Indirect Calls and Targets

To better understand the risk for control-flow attacks (RQ2), we analyze indirect calls and their call targets in the binaries.

Indirect Calls First, we want to know how many indirect calls are present in a binary, since each such call could be a source of an unintended control-flow edge. Our analysis disassembles all binaries in Table 2 and counts the number of `call_indirect` instructions (column "Indirect calls: Count"). The percentage of indirect calls relative to all calls varies considerably between programs (column "of All"), from 0.6% up to 31.3%. We also observe that the proportion of indirect calls is independent of whether the source language is C or C++. Averaged over all 26 programs, 9.8% of all call instructions are indirect, i.e., almost every tenth call can be potentially diverted to other functions.

Indirectly Callable Functions To successfully redirect a control-flow edge, an attacker not only needs to find an indirect call instruction as the source, but also a compatible function as the target. Two pre-conditions must hold for a function to be a valid indirect call target (Section 2). First, the function's type must be compatible with the type statically encoded in the indirect call instruction. WebAssembly function types are very low-level, however. For example, the type `[i32]` → [] is compatible with all C functions that return `void` and take as argument a pointer (regardless of type or `const`-ness), an array, a plain `int`, or anything else that is represented as a 32-bit integer, e.g., enums.

Second, the function must be present in the `table` section of the binary, because the index passed to `call_indirect` is resolved to a function via this table. Our static analysis tool finds which functions are initialized in the table at program startup. Entries in the table cannot be manipulated by the WebAssembly program itself. In principle, the host environment, e.g., JavaScript in the browser, could add or remove entries at runtime. We manually verified that the JavaScript code generated by Emscripten does not modify the table, and thus assume our analysis precisely measures the potential targets of indirect calls.

The columns "Indirectly Callable" in Table 2 show how many functions are type-compatible with at least one `call_indirect` instruction *and* present in the `table` section. The percentage of indirectly callable functions ranges from 5% to 77.3%, with on average 49.2% of all functions in the program corpus.

Function Pointers in Memory The above results give an *upper bound* of potential targets for control-flow divergence. In practice, if the table index passed to `call_indirect` comes from a local variable, a global variable, or is the result of

Binary	Source	Instruct.	Indirect calls		Functions					CFI equivalence classes			
			Count	of All	Count	Indirectly Callable		Idx. from mem.		Count	Min	Max	Avg
Collected from deployed applications													
Adobe View SDK	C++	1.1M	2803	6.2%	12566	3076	24.5%	3054	24.3%	87	1	848	32.2
1Password X exten.	Rust	730.2k	283	1.4%	1941	596	30.7%	586	30.2%	19	1	91	14.9
Doom 3	C++	1.7M	17903	31.3%	8239	4449	54.0%	4408	53.5%	642	1	3889	27.9
Figma	C++	3.2M	10469	8.1%	13619	3657	26.9%	3635	26.7%	68	1	4519	154.0
WebP encoder	C	73.1k	87	3.6%	889	165	18.6%	69	7.8%	22	1	15	4.0
WebP decoder	C	43.4k	69	5.4%	563	160	28.4%	107	19.0%	20	1	9	3.5
mozjpeg	C	77.7k	298	22.0%	388	135	34.8%	116	29.9%	28	1	169	10.6
optipng	C	119.2k	169	5.4%	735	152	20.7%	124	16.9%	28	1	34	6.0
hqx	Rust	111.4k	34	0.6%	73	17	23.3%	15	20.5%	4	1	16	8.5
Compiled from SPEC CPU 2017													
500.perlbench	C	837.8k	425	1.6%	2128	980	46.1%	956	44.9%	31	1	93	13.7
502.gcc	C	2.9M	3642	2.5%	9541	3394	35.6%	3375	35.4%	78	1	982	46.7
505.mcf	C	27.4k	44	8.8%	136	12	8.8%	8	5.9%	7	1	28	6.3
508.namd	C++	323.0k	41	1.1%	296	124	41.9%	107	36.1%	15	1	12	2.7
510.parest	C++	1.0M	1229	2.6%	3762	2864	76.1%	2714	72.1%	97	1	199	12.7
511.povray	C++	385.4k	228	1.9%	1421	521	36.7%	510	35.9%	29	1	57	7.9
519.lbm	C	13.4k	12	6.2%	80	7	8.8%	6	7.5%	5	1	8	2.4
520.omnetpp	C++	619.3k	4536	10.6%	4615	3569	77.3%	3505	75.9%	79	1	1631	57.4
523.xalancbmk	C++	1.5M	13567	16.2%	8050	6225	77.3%	6072	75.4%	77	1	3893	176.2
525.ldecod	C	233.0k	354	8.6%	551	129	23.4%	68	12.3%	24	1	135	14.8
525.x264	C	283.6k	773	14.2%	636	253	39.8%	177	27.8%	31	1	105	24.9
526.blender	C++	3.2M	17198	14.9%	25901	17387	67.1%	17263	66.6%	128	1	5360	134.4
531.deepsjeng	C	53.0k	10	1.1%	174	10	5.7%	8	4.6%	5	1	6	2.0
538.imagick	C	517.5k	1901	9.9%	1068	91	8.5%	74	6.9%	22	1	1592	86.4
541.leela	C++	118.8k	263	5.0%	1101	600	54.5%	520	47.2%	41	1	74	6.4
544.nab	C	55.6k	17	1.7%	201	10	5.0%	8	4.0%	6	1	7	2.8
557.xz	C	53.3k	71	11.0%	250	98	39.2%	86	34.4%	19	1	11	3.7
Average per binary		738.1k	2939.5		3804.8	1872.3		1829.7		62.0	1	914.7	33.2
Total		19.2M	76426	9.8%	98924	48681	49.2%	47571	48.1%				

Table 2: Program corpus overview and static analysis results regarding indirect calls, function table, and CFI equivalence classes.

a sequence of instructions, then the indices an attacker can choose from are likely more restricted. As a lower bound of targets to choose from, we also measure how many table indices are read directly from memory. We obtain this number through a static analysis of the instructions preceding indirect calls. Columns "Idx. from mem." show the number of type-compatible and in-table functions, for which at least one indirect call exists that takes its table index *directly from linear memory*. For each such function, given an arbitrary write primitive into linear memory, an indirect call could be diverted to reach the function. Perhaps surprisingly, this lower bound of reachable functions is very close to the upper bound: On average, 48.1% of all functions can be reached by a `call_indirect` that takes its argument from linear memory.

Overall, our analysis of indirect calls and targets shows a large potential for effective control-flow divergence. Many functions are indirectly callable (49.2%, on average) and most of them could be reached by simply overwriting an index stored in linear memory (48.1%). We conclude that diverging indirect function calls poses a serious threat to the integrity of control flow in WebAssembly.

6.4 Comparing with Existing CFI Policies

WebAssembly's type checking of indirect calls can be seen as a form of control-flow integrity (CFI) for forward edges.[12] CFI has been extensively researched [15, 21, 39, 48, 49, 63, 69, 71, 72] and is deployed in open-source (GCC [62] and LLVM [6]) and commercial compilers (MSVC [2]). We now compare WebAssembly's type checking with state-of-the-art CFI defenses for native binaries (RQ3).

Equivalence Classes Following prior work on CFI [21], we measure its effectiveness by analyzing the sets of control-flow targets an indirect transfer may be diverted to according to the CFI mechanism. Each such a set is called a *CFI equivalence class*. To assess the effectiveness of a CFI defense, we use two measures: The class count, i.e., how many different classes exist, and the sizes of the classes, i.e., how many targets are in each class. A small class count means the CFI defense distinguishes little between targets, giving attackers more options for control-flow divergence. A large class size is also

[12] Backward edges, i.e., returns, are protected due to being managed by the VM and offer security conceptually similar to shadow stack solutions.

Program	Number of CFI equivalence classes			
	MCFI [48]	πCFI [49]	LLVM-CFI 3.9	Wasm
perlbench	38	30	36	31
mcf	12	8	N/A	7
omnetpp	357	321	35	79
xalancbmk	1534	1200	260	77
namd	166	150	4	15
povray	218	204	33	29

(a) Number of equivalence classes (higher means more secure).

Program	Size of largest CFI equivalence class			
	MCFI [48]	πCFI [49]	LLVM-CFI 3.9	Wasm
perlbench	348	347	350	93
mcf	29	15	N/A	28
omnetpp	275	253	170	1631
xalancbmk	1141	608	95	3893
namd	187	113	30	12
povray	187	113	81	57

(b) Sizes of equivalence classes (lower means more secure).

Figure 11: Comparing WebAssembly with native CFI solutions. Native data taken from [21] for programs in the intersection of SPEC CPU 2006 (theirs) and 2017 (ours).

insecure, as it means a large number of control-flow targets can all be reached from a single source instruction.

For WebAssembly, we measure CFI equivalence classes by analyzing the type signatures of indirectly callable functions, assigning all functions with the same type signature into an equivalence class. Additionally, we analyze the preceding instructions before an indirect call to determine whether they restrict the table index, e.g., via bitmasking, to a smaller range. The last block in Table 2 shows the results. On average, there are 62 equivalence classes per program, which each contain 33.2 functions. The largest equivalence class, in the Blender program, contains over 5,300 functions. Overall, this shows that an attacker has plenty of call targets to choose from.

Comparing with Native CFI Defenses To put the results on equivalence classes in perspective, we compare them with results reported for native CFI defenses [21]. The tables in Figure 11a and Figure 11b compare the counts and sizes of equivalence classes, respectively. For example, MCFI [48] and πCFI [49] partition the control-flow targets of xalancbmk into 1534 and 1200 classes, respectively, whereas WebAssembly's indirect call target restrictions yield only 77 such classes. Regarding the size of equivalence classes, WebAssembly has especially large classes for omnetpp and xalancbmk, and similar classes sizes as the native defenses for other programs.

Interestingly, omnetpp and xalancbmk are C++ programs that make heavy use of object-oriented programming with virtual functions. Source-level type information, e.g., about class hierarchies, can help compiler-based CFI methods to identify more precise, and thus restrictive, equivalence classes. In contrast, WebAssembly's type checking has only (combinations of) four low-level primitive types to work with, which might explain the stark difference to the native schemes.

Overall, WebAssembly's type checking is often less effective than modern CFI defenses available for native binaries. While type-checked indirect calls certainly are a step forward compared to not having any CFI defense, adapting more sophisticated CFI defenses could significantly harden the currently produced binaries. For example, Clang's CFI scheme, which uses source-level information, can also be employed by passing `-fsanitize=cfi` when compiling to WebAssembly.

7 Discussion of Mitigations

The following discusses several mitigations that could defeat the attacks presented in this paper, e.g., through amending the language specification, updates to compilers, or by application and library developers.

7.1 WebAssembly Language

Several proposals for extending the core WebAssembly language could address some of our attack primitives.

The multiple memories proposal [54] gives one module the option of having multiple linear memories. Under the proposal, memory operations statically encode which memory they operate on, e.g., an `i32.load $mem2` instruction can only load data from memory 2. Multiple memories would enable separating stack, heap, and constant data. Thus, an overflow in one memory section would no longer affect data in another memory. Also, pointers to the heap could no longer be forged to point into the stack and vice versa. Finally, if compilers emit only load instructions for a particular memory section, it becomes effectively read-only, since stores to other memories can never modify it. This would prevent overwriting of constants. A challenge with this proposal is that compiling to multiple memories is not straightforward. Since memory accesses are statically restricted to a certain memory, code that must handle pointers of different regions must either be duplicated or objects explicitly copied between memories.

The reference types proposal [55] allows modules to have multiple tables for indirect calls. Our call redirection primitive is powerful only because all indirectly callable functions currently are in the same table. Multiple tables allow for more fine-grained defenses. One option is to define different protection domains, e.g., one per statically-linked library, and to keep a separate table per protection domain. Another option is to split call targets into equivalence classes, similar to existing CFI techniques for native binaries, and to keep a separate table per equivalence class.

Finally, the MS-Wasm proposal [26] explicitly targets memory safety. It proposes to add so called segments to WebAs-

sembly, memory regions with defined size and lifetime. Handles into those segments are promoted to first class types, with own operations for allocation and slicing. This requires quite some implementation effort by hosts, and unless hardware support for memory safety is provided, will likely incur a performance overhead.

A challenge with all changes to the core language is that they require updating existing virtual machines. Since WebAssembly is implemented not just by one vendor, but in at least four browsers (Chrome, Firefox, Safari, and Edge), Node.js, and several standalone VMs (Wasmtime, WAVM, Lucet), this risks a split of the still young ecosystem. However, both the multiple memories and reference types proposal are (as of June 2020) in phase 3 of the four phase standardization process.[13]

7.2 Compilers and Tooling

The perhaps most simple way of preventing many of our attack primitives is to implement and activate security features that compilers, linkers, and allocators already provide for native compilation targets. Decades of research on binary security [61] have resulted in several mitigations that could be applied to WebAssembly. Examples that would benefit WebAssembly compilers are `FORTIFY_SOURCE`-like code rewriting, stack canaries, CFI defenses, and safe unlinking in memory allocators. In particular for stack canaries and rewriting commonly exploited C string functions, we believe there are no principled hindrances to deployment. We hope they will be implemented by compilers in the future, since they offer good security benefit for relatively little change to the ecosystem, unlike, e.g., language changes.

A longer-term mitigation in compilers is to use the WebAssembly language extensions discussed above, once they become available. For example, when compiling C/C++ to WebAssembly, multiple memories could mimic some of the security features provided by page protections in native code.

7.3 Application and Library Developers

Developers of WebAssembly applications can reduce the risk by using as little code in "unsafe" languages, such as C, as possible. To reduce the attack surface, developers should also ensure to import only those APIs from the host environment that are strictly necessary. For example, calling critical host functions, such as `eval` or `exec` is impossible unless these functions are imported in the WebAssembly module.

8 Related Work

WebAssembly The language has a formally defined type system shown to be sound [32, 67]. WebAssembly performance and how it compares to native performance has been studied [37]. Wasabi [42] is a general dynamic analysis framework for WebAssembly. We discussed different proposals to extend the language [26, 54, 55] in the previous section.

Malicious WebAssembly Among the early adopters of WebAssembly have been websites that use the computing resources of visitors to mine cryptocurrencies [41, 46, 56]. Since this is often unwelcome, several approaches detect and defend against mining [40, 66], e.g., by profiling executed instructions. Taint tracking techniques can also be used to enforce security policies on untrusted WebAssembly programs [29, 60].

Malicious WebAssembly binaries are also crafted to escape browser sandboxes and gain remote code execution [16, 59]. Unlike our work, those exploits attack bugs in specific VM implementations and fall into the realm of host security, as discussed in Section 1. Others use malicious WebAssembly code to perform side-channel [30] and speculative execution attacks [43] against the host. In contrast, we do not aim to escape the sandbox, and our attacks assume nothing but a standards-compliant WebAssembly VM. For example, the exploit in Section 5.1 works in both Firefox and Chrome. Since we do not escape the VM, we depend on the available imported host functions for malicious actions. However, as we show in our end-to-end exploits, cross-site scripting, remote code execution, and file writes can still be consequences.

Vulnerable WebAssembly Two industry whitepapers show example attacks against vulnerable WebAssembly binaries [20, 45]. Their pioneering work prompted us to investigate WebAssembly binary security more thoroughly and expand this research significantly in several directions.

In Section 3, we systematically analyze how program data is mapped to linear memory by three different compilers, two backends, and two linker configurations, whereas previous work has only looked at select examples from a single compiler. From our analysis we conclude that, fundamentally due to linear memory, WebAssembly cannot separate static data, heap, and unmanaged stack, as guard pages like in native binaries are unavailable. Unlike previous work, we thus show a much larger set of attack primitives, including primitives have not been reported for WebAssembly at all. For example, we are the first to propose stack overflows (not buffer overflows) as an attack primitive (Section 4.1.2). Prior work has hypothesized that exploitation is possible, but we are the first to demonstrate it in practice. One whitepaper and a blog post [25, 45] warn that WebAssembly binaries come with their own allocator, which is potentially not hardened. Our exploits against two different versions of Emscripten's `emmalloc` substantiate their hypotheses.

We also perform the first quantitative security evaluation on a set of 26 WebAssembly binaries with more than 19 million instructions in total (Section 6). A previous blog post [28] explores that indirect calls can be redirected to unintended

[13] https://github.com/WebAssembly/proposals

functions on a single example. We make this observation quantifiable and measure that almost every second function can be reached via an indirect call that takes its argument directly from linear memory. We are also the first to estimate how much data resides on the unmanaged stack in linear memory, a relevant number for estimating the risk from previously described data overwrite primitives, and the first to compare WebAssembly's type-checking of indirect calls with native CFI schemes.

Defensive WebAssembly WebAssembly's well-designed host security properties can also serve as a basis for software-fault isolation (SFI). By compiling individual libraries to WebAssembly and embedding a runtime into the main application, memory errors in the library are isolated from the main program. This has recently been successfully employed to sandbox libraries in Firefox [47]. WebAssembly has also been used as a compilation target for formally verified cryptography [52] and extended to guarantee constant-time operations for cryptographic primitives [68].

Exploiting Native Binaries There exists ample work on binary exploitation; Sezkeres et al. [61] provide an excellent overview of techniques for exploiting memory errors. The stages used in their survey roughly corresponds to the dimensions of attack primitives we use in Section 4. We find that, although the exploit chains have to be adapted and effects depend on the runtime environment, many techniques that are effective in native binaries also transfer to WebAssembly.

Exploit Mitigations In response to attacks on native binaries, many mitigations have been developed, including data-execution prevention [17], stack canaries [24], ASLR [51], and safe unlinking. The idea of control-flow integrity [15] is the basis of several protection mechanisms. Control-flow bending [22], data-only attacks [36], and other advanced attacks [27, 57] demonstrate that even restrictive CFI policies leave sufficient freedom for an attacker. Burow et al. [21] provide a survey assessing the security of different CFI implementations. Section 6.3 empirically compares with some of their results. The restrictions imposed by WebAssembly raise the difficulty for exploitation, but do not offer complete security. We expect to see an arms race of mitigations and ever more complex attacks in the WebAssembly ecosystem, too, which will gradually increase security.

9 Conclusion

WebAssembly promises a portable platform for code compiled from C, C++, and other languages that combines near-native performance with strong safety and security guarantees. This paper presents the first in-depth security analysis of WebAssembly binaries and compares the level of security provided by WebAssembly with native platforms. We find that vulnerable source programs result in binaries that enable various kinds of attacks, including attacks that have not been possible on native platforms since decades. Our findings are based on a set of attack primitives that enable an attacker to gain a write primitive, overwrite sensitive data, and trigger compromising behavior. Several end-to-end examples of attacks, which cover WebAssembly running in the browser, on Node.js, and in stand-alone VMs, demonstrate that these primitives can be combined into effective exploits. Moreover, an empirical evaluation of real-world binaries quantifies the exploitation risk, showing a large attack surface. Overall, our findings are a call to arms for further hardening the WebAssembly language, its compilers, and ecosystem, making the promise of a secure platform a reality.

Acknowledgments

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

References

[1] WebAsssembly Design – Security – Memory Safety. https://github.com/WebAssembly/design/blob/master/Security.md#memory-safety, 2016.

[2] Control Flow Guard. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, 2018.

[3] National Vulnerability Database – CVE-2018-14550 Detail. https://nvd.nist.gov/vuln/detail/CVE-2018-14550, 2018.

[4] Wasmer – The Universal WebAssembly Runtime. https://wasmer.io/, 2019.

[5] Blazor – Build client web apps with C#. https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor, 2020.

[6] Clang 11 documentation – Control Flow Integrity. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2020.

[7] Emscripten – Calling JavaScript from C/C++. https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#interacting-with-code-call-javascript-from-native, 2020.

[8] Emscripten – Release Notes. https://emscripten.org/docs/introducing_emscripten/release_notes.html, 2020.

[9] WASI – The WebAssembly System Interface. https://wasi.dev/, 2020.

[10] Wasmtime – A small and efficient runtime for WebAssembly & WASI. https://wasmtime.dev/, 2020.

[11] WAVM. https://wavm.github.io/, 2020.

[12] WebAssembly. https://webassembly.org/, 2020.

[13] WebAssembly – Use Cases. https://webassembly.org/docs/use-cases/, 2020.

[14] WebAssembly Specification. https://webassembly.github.io/spec/core/index.html, 2020.

[15] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS 05)*, 2005.

[16] Georgi Geshev Alex Plaskett, Fabian Beterke. Apple Safari – Wasm Section Exploit. https://labs.f-secure.com/assets/BlogFiles/apple-safari-wasm-section-vuln-write-up-2018-04-16.pdf, 2018.

[17] Starr Andersen and Vincent Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb457155(v=technet.10), 2004.

[18] Anonymous. Once upon a free. *Phrack*, 11(9), November 2001.

[19] J.F. Bastien. WebAssembly – Going public launch bug. https://github.com/WebAssembly/design/issues/150, 2015.

[20] 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.

[21] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.*, 50(1), 2017.

[22] Nicholas Carlini, Antonio Barresi, Mathias Payer, David A. Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security 15)*, 2015.

[23] 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.

[24] Crispan Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX Security 98)*, 1998.

[25] Frank Denis. WebAssembly doesn't make unsafe languages safe (yet). https://00f.net/2018/11/25/webassembly-doesnt-make-unsafe-languages-safe/, 2018.

[26] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position Paper: Progressive Memory Safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2019.

[27] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015.

[28] Jonathan Foote. Hijacking the control flow of a WebAssembly program. https://www.fastly.com/blog/hijacking-control-flow-webassembly-program, 2019.

[29] William Fu, Raymond Lin, and Daniel Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. http://arxiv.org/abs/1802.01050, 2018.

[30] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-By Key-Extraction Cache Attacks from Portable Code. In *Proceedings of the International Conference on Applied Cryptography and Network Security (ACNS)*, 2018.

[31] Robbert Gurdeep Singh and Christophe Scholliers. WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2019)*, 2019.

[32] 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 (PLDI 2017)*, 2017.

[33] 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)*, 2019.

[34] Pat Hickey. Edge programming with Rust and WebAssembly. https://www.fastly.com/blog/edge-programming-rust-web-assembly.

[35] Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, 2019.

[36] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, 2018.

[37] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, July 2019.

[38] Michel Kaempf. Vudo – An object superstitiously believed to embody magical powers. *Phrack*, 11(8), November 2001.

[39] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[40] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In *Proceedings of the 2019 World Wide Web Conference (WWW '19)*, 2019.

[41] 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 '18)*, 2018.

[42] Daniel Lehmann and Michael Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *Proceedings of the 24th International Conference on Architectural Support*

for Programming Languages and Operating Systems (ASPLOS '19), 2019.

[43] Giorgi Maisuradze and Christian Rossow. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*, 2018.

[44] 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.

[45] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. NCC Group Whitepaper – Security Chasms of WASM. https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf, 2018.

[46] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2019)*, 2019.

[47] 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 *Proceedings of the 29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[48] Ben Niu and Gang Tan. Modular Control-Flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*, 2014.

[49] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015.

[50] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996.

[51] PaX Team. PaX Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt, 2002.

[52] J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. Formally Verified Cryptographic Web Applications in WebAssembly. In *2019 IEEE Symposium on Security and Privacy (SP 2019)*, 2019.

[53] Andreas Rossberg. Why WebAssembly? https://medium.com/dfinity/why-webassembly-f21967076e4, 2018.

[54] Andreas Rossberg. Multiple per-module memories for Wasm. https://github.com/WebAssembly/multi-memory, 2019.

[55] Andreas Rossberg. Proposal for adding basic reference types. https://github.com/WebAssembly/reference-types, 2019.

[56] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. Digging into browser-based crypto mining. In *Proceedings of the Internet Measurement Conference 2018 (IMC '18)*, 2018.

[57] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *2015 IEEE Symposium on Security and Privacy (SP 2015)*, 2015.

[58] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, 2004.

[59] Natalie Silvanovich. The Problems and Promise of WebAssembly. https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html, 2018.

[60] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint Tracking for WebAssembly. https://arxiv.org/abs/1807.08349, 2018.

[61] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy (SP 2013)*, 2013.

[62] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, August 2014.

[63] Victor van der Veen, Dennis Andriesse, Enes Göktaundefined, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015.

[64] Kenton Varda. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/, 2018.

[65] Luke Wagner. WebAssembly consensus and end of Browser Preview. https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html, 2017.

[66] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks. In *24th European Symposium on Research in Computer Security (ESORICS 2018)*, 2018.

[67] Conrad Watt. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '18)*, 2018.

[68] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[69] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security 19)*, August 2019.

[70] Alon Zakai. Emscripten and the LLVM WebAssembly backend. https://v8.dev/blog/emscripten-llvm-wasm, 2019.

[71] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy (SP 2013)*, 2013.

[72] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013.