WASHADOW: Effectively Protecting WebAssembly Memory Through Virtual Machine-Aware Shadow Memory

Zhuochen Jiang Baojian Hua
School of Software Engineering, University of Science and Technology of China
Suzhou Institute for Advanced Research, University of Science and Technology of China
jzc666@mail.ustc.edu.cn bjhua@ustc.edu.cn

***Abstract*—WebAssembly (Wasm) is an emerging binary instruction set architecture designed for secure binary program execution and is rapidly being deployed across various security-critical domains, such as edge computing and smart contracts. However, despite its security-oriented design, Wasm remains susceptible to vulnerabilities, including integer overflows and memory corruption, due to the lack of effective protection mechanisms, which undermines its security guarantees.**

In this paper, we present WASHADOW, the first approach for effective Wasm protection using virtual machine-aware shadow memory. Our key insight is that, since Wasm is a virtual instruction set, we can leverage memory layout information in the underlying Wasm VMs to enforce the protection. Specifically, we first extend the Wasm VMs with shadow memory to record memory information and track the status of linear memory, by introducing two new pseudo Wasm instructions for inserting and performing sanity checks on canaries in the linear memory. We then design a static binary instrumentation method to instrument Wasm binaries with canary instructions. Finally, we implement these canary pseudo-instructions through virtual machine extensions as well as a set of vulnerability detection algorithms as security plugins. We implemented a software prototype for WASHADOW and conducted extensive experiments to evaluate its effectiveness, usability, and overhead on micro and real-world benchmarks. Experimental results demonstrated that WASHADOW is effective in protecting Wasm linear memory against various memory vulnerabilities, with an average code size increases of 26.5% and an execution time penalty of 108.5%.

***Index Terms*—WebAssembly Security, Canary, Instrumentation, Shadow Memory**

I. INTRODUCTION

WebAssembly [1] (Wasm) is an emerging binary instruction set architecture and code distribution format [2] with a core design goal of security, incorporating a wide range of secure language features such as strong type systems [2], mathematically rigorous operational semantics [3], software fault isolation [4], secure control flow [5], and linear memory [6]. Given its security potentials and guarantees, Wasm is rapidly being deployed in various security-critical domains such as edge computing [7] and smart contracts [8], and is poised to become one of most important instruction set architectures for code execution and distribution in the coming decade.

Yet, despite Wasm's security design goals and promises, recent studies [9] [10] [11] [12] [13] have revealed that Wasm programs remain vulnerable and exploitable due to defects in its memory model design and the lack of effective memory protections. Specifically, Wasm introduced a novel security design called *linear memory* [14] to mitigate notorious memory attacks, such as buffer overflows [15], by leveraging the key concept of shadow stacks [16] [17] [18] [19]. Unfortunately, while this security design effectively protects the return address from being attacked, data in the linear memory can still be overwritten due to unmanaged stack buffer overflows [9], corrupting both the constant data area and heap metadata. Even more concerning, current Wasm compilers, such as Emscripten [20], support generating Wasm binaries from unsafe languages like C/C++. Consequently, memory vulnerabilities in C/C++ source programs can propagate into Wasm binaries [13]. Therefore, providing effective memory protection and security enhancements for Wasm is both critical and urgent.

Recognizing the criticality and urgency of Wasm security, a board range of studies have been conducted on this topic [10] [11] [12] [13] [24] [21] [25] [26] [27]. To put the discussion into perspective, we list, in Table I, existing studies and state-of-the-art systems focused on Wasm memory protection, including Fuzzm [21], metaSafer [22], and PKUWA [23].

TABLE I: An overview of existing Wasm memory protection frameworks and WASHADOW.

	Fuzzm [21]	metaSafer [22]	PKUWA [23]	WASHADOW
Compiler	Clang	Emscripten	LLVM	Emscripten/ LLVM/Clang
CDO	✘	✘	✔	✔
DF	✘	✘	✘	✔
HBO	✔	✔	✔	✔
ML	✘	✘	✘	✔
NPD	✘	✘	✘	✔
SBO	✔	✘	✔	✔
UAF	✘	✘	✘	✔
Technique	canary/ fuzzing	shadow memory	memory isolation	canary/ shadow memory

CDO: Constant Data Overwrite; DF: Double Free;
HBO: Heap-based Buffer Overflow; ML: Memory Leak;
NPD: Null Pointer Dereference; SBO: Stack-based Buffer Overflow;
UAF: Use After Free

While each of these studies offers valuable contributions, they do not fully address the Wasm memory protection problem. First, existing studies lack versatility because they rely on specific compilation toolchains. For example, Fuzzm [21] instruments Wasm programs compiled from the Clang compiler [28], making it unclear how to adapt Fuzzm's protection to other Wasm compilers, such as Emscripten [20] utilized in metaSafer [22], due to the complexity of different compiler design. Even if the adaption is possible, it remains labor-intensive due to the substantial codebase of compilers (e.g., the Clang compiler [29] recently surpassed 6 million lines of code and continues to grow rapidly). Second, existing studies sacrifice generality by utilizing unique security mechanisms tied to specific architectures. For example, PKUWA [23] leverages Intel MPK [23] to mitigate memory vulnerabilities. This makes it challenging, if not impossible, to deploy such protection on architectures lacking similar hardware mechanisms. Finally, from the vulnerability perspective, existing studies struggle with scalability. They typically address only a few specific types of memory vulnerabilities. For example, metaSafer [22] focuses on heap metadata corruption but overlooks other vulnerabilities, such as stack-based buffer overflows. As a result, many vulnerabilities remain undetected and unmitigated, even when the proposed protections are applied.

In this paper, we present the *first* approach for effective Wasm memory protection by utilizing shadow memory in Wasm virtual machines (VMs) to address existing limitations. Our key insight is that, since Wasm is a virtual instruction set executed by underlying Wasm VMs, we can and *should* leverage the capabilities of these Wasm VMs to obtain essential memory layout information for protection. Based on this key insight, we extend Wasm VMs with shadow memory to record memory information and track the status of linear memory, aiding in the detection of various memory vulnerabilities. First, we design a representation of layout information for shadow memory, encompassing the heap, the unmanaged stack, the static data areas, and the canaries which are inserted for detecting buffer overflows. Furthermore, to effectively recognize inserted canaries in linear memory, we utilize Wasm VM-aware canaries [30].

We then design a static instrumentation technique to instrument Wasm binary programs with the aforementioned canaries. The instrumentation process involves statically rewriting the Wasm binaries to insert necessary instructions at function entry and exit blocks, as well as at other locations that may harbor potential vulnerabilities.

Finally, we develop a set of vulnerability detection algorithms to effectively and promptly identify and mitigate potential memory vulnerabilities in Wasm programs.

In implementing the whole process, we have addressed three technical challenges. **C1:** language and toolchain diversity complicates the development of comprehensive protection. To address this challenge, we proposed a holistic binary-level protection that leverages binary pseudo-instructions, static binary instrumentation, and dynamic binary hooking. As shown in Table I, our approach supports a diverse set of compiler toolchains, outperforming existing methods. **C2:** the scalability limitations of existing tools restrict the types of vulnerabilities that can be detected. To address this issue, we designed a set of vulnerability detection algorithms, including the insertion of canaries to guard against buffer overflow. Moreover, to address the issue of canary semantics being transparent to the Wasm VMs, we utilized and modified two novel Wasm instructions with VM-aware semantics, extending them to support canary insertion in the heap, in accordance with the Wasm design [14]. **C3:** Wasm VM does not natively support pseudo-instructions. To address this, we developed an implementation strategy for VM extension that requires minimal modification to the VM.

We implemented a prototype for our approach, debubbed WASHADOW, and conducted extensive experiments to evaluate its effectiveness, usability, and overhead. These evaluations were performed using a micro benchmark consisting of 15 CWEs that we collected and created as well as a real-world benchmark comprising four large-scale Wasm projects. The experimental results demonstrated that WASHADOW effectively detects seven kinds of memory vulnerabilities, as shown in Table I, outperforming existing approaches. Furthermore, WASHADOW proved useful in protecting real-world projects, successfully detecting 76 of 85 (89.41%) vulnerabilities across the four projects. Finally, WASHADOW introduces acceptable overhead, with an average file size increase of 26.5% and an execution time increase of 108.5%, which is consistent with prior studies [24].

In summary, this paper makes the following contributions:

- We propose the first approach to effectively protect Wasm memory, by leveraging a VM-aware shadow memory method.
- We design and implement a software prototype WASHADOW to validate our approach.
- We conduct extensive experiments to show that WASHADOW is effective in Wasm memory protection with acceptable overhead, outperforming state-of-the-art studies.

The remainder of this paper is organized as follows. Section II introduces the background. Section III outlines the motivations and the threat model. Section IV presents our approach. Section V presents the experimental evaluation of WASHADOW. Section VI discusses limitations and future directions. Section VII reviews related work, and Section VIII concludes.

II. BACKGROUND

To be self-contained, in this section, we present the necessary background knowledge on Wasm (§ II-A) and its memory layout (§ II-B).

A. Wasm

Wasm is an emerging secure and portable instruction set architecture, initially released in 2017 for Web. In 2019, with the introduction of the Wasm System Interface (WASI) [31], Wasm becomes an official Web standard and evolved into

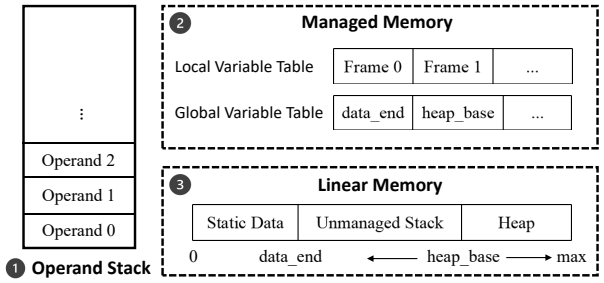


Fig. 1: The typical memory layout of Wasm VM.

```
void get_token(FILE *pnm_file, char *token){
  do{
    ret = fgetc(pnm_file);
    if (ret == EOF) break;
    i++;
    // if (i > sizeof(token)), triggers an BO
    token[i] = (unsigned char)ret;
  }while((token[i] != '\n')&&(token[i] != '\r')
        &&(token[i] != ' '));
}
```

Fig. 2: A sample program snippet we adapted from the CVE-2018-14550 [44], comprising a buffer overflow vulnerability.

```
int main(int argc, char **argv){
  std::string img_tag =
    "<img src='data:image/png:base64,";
  pnm2png("input.pnm", "output.png");
  img_tag += file_to_base64("output.png") + "'>";
  emcc::global("document").call("write",img_tag);
  return 0;
}
void pnm2png(char *from, char *to){
  // "token" resides in the linear memory
  char token[...];
  get_token(from, token);
  ...;
}
```

Fig. 3: The proof-of-concept code to exploit the vulnerability.

a general-purpose language deployed across various domains beyond the Web.

Wasm was designed with the goals of security, efficiency, and portability. First, it introduced various security features such as strong typing [2], rigorous operational semantics [3], software fault isolation [4], secure control flow [5], and linear memory [6], to ensure program security. Second, Wasm VMs allow Wasm programs to efficiently leverage hardware capabilities across various platforms. Finally, the design of WASI interface facilitates portability.

Due to its security advantages, Wasm is rapidly being deployed in both Web and non-Web domains. In the Web domain, Wasm has become one of the four official languages for Web development, with full support from major browsers [32] [33]. In non-Web domains, Wasm is widely used in various scenarios such as cloud computing [34] [35] [36] [37], edge computing [38] [39] [40], and server-side computing [41]. In the future, a desire to deploy portable binaries without sacrificing security will make WASHADOW more attractive.

B. Memory Layout

To provide strong security guarantees, Wasm have introduced a specialized memory layout based on the concept of linear memory [14]. Fig. 1 presents a typical memory layout for Wasm VMs, consisting of three key components: the operand stack (❶), the managed memory (❷), and the linear memory (unmanaged memory) (❸).

First, Wasm VMs operate as stack-based abstract machines, where operands and results are stored in the operand stack. For example, when executing an addition instruction `i32.add`, the Wasm VM pops two operands, `operand2` and `operand1`, off the operand stack and then pushes the sum, `operand1+operand2`, back onto the operand stack. Second, the managed memory stores global variables and call stack frames. The call stack frames contain only scalar data (i.e., `int` and `float`) and function return addresses. The managed memory is directly managed by Wasm VM, preventing its contents from being accessed by user programs. Third, linear memory is a continguous storage space used by Wasm user programs, consisting of a static data area, an unmanaged stack, and a heap for dynamic memory allocation. Specifically, frames in the unmanaged stack store aggregated local variables (*e.g.*, buffers) within a function. The separation of unmanaged stack from managed stack ensures that buffer overflows in the unmanaged stack do not overwrite return addresses in the managed stack, effectively preventing ROP attacks [42] and control-flow hijacking [43].

III. Motivations, Challenges, and Threat Model

In this section, we present the motivation (§ III-A) through a running example, followed by the security challenges (§ III-B), and the threat model (§ III-C) for this work.

A. Motivations

Despite the security assurances provided by Wasm, recent studies [10] [9] have revealed that that numerous memory security vulnerabilities present in native binaries—previously mitigated by existing techniques—may now be exploitable in Wasm binaries. For example, while Wasm's separation of unmanaged memory protects the return address from corruption, it does not safeguard sensitive data stored in linear memory. Consequently, memory vulnerabilities in linear memory can lead to metadata corruption or heap overflows, because these regions are contiguous. These findings underscore the urgent need to explore novel security techniques to protect Wasm linear memory, thereby providing effective and comprehensive safeguards.

To better illustrate the motivation behind our research, we provide a running example in Figs. 2 and 3 to demonstrate how memory vulnerabilities manifest in Wasm and why existing memory protection mechanisms fail. We created this example by adapting the CVE-2018-14550 [44]. The function

`get_token` in Fig. 2 contains a buffer overflow (BO) vulnerability (line 8), as the array index `i` is not validated against the length of the array `token`. The proof-of-concept code in Fig. 3 exploits this vulnerability by passing an array `token` to the vulnerable function `get_token` (line 12). Since the array `token` resides in the linear memory, overflows in `token` will corrupt adjacent data.

While well-established stack canaries [45] can effectively protect stack buffers in native binaries such as x64 and AArch64, they cannot protect Wasm memory due to Wasm's explicit separation of managed memory and linear memory—a key security feature of Wasm. As illustrated in Fig. 1, the `token` array is allocated in the unmanaged stack residing in the linear memory, whereas the canary is located in the frame residing in the managed memory (recall that all scalar values, including canaries, reside in the managed memory). This physical separation violates a fundamental semantic requirement for canaries: they must be *adjacent* to the buffer they protect. Consequently, even if a buffer overflow occurs, the canary will not be overwritten and thus the overflow remains undetected.

In Section V-H, we will demonstrate how our approach and system WASHADOW can safeguard overflows as in this running example.

B. Security Challenges

Despite the security criticality and urgency [9] [10] [11], to the best of our knowledge, memory protection and enhancement for Wasm has not been thoroughly studied. Developing an effective approach for Wasm protection faces several technical challenges.

C1: language and toolchain diversity. Wasm has a rich ecosystem with various toolchains (e.g., Emscripten [20] and LLVM [29]) supporting a wide range of source languages (e.g., C/C++, JavaScript, and Rust). Consequently, developing a holistic protection for all source languages is challenging due to language discrepancies. Moreover, even porting a mature protection in one compiler toolchain (e.g., canaries in Clang) to another compiler (e.g., `rustc` for Rust) is difficult [13], due to the substantial code size of compilers.

Solution: We propose employing a binary-level protection stategy directly on Wasm binaries, leveraging the standard and mathematically rigorous Wasm specification [14]. To implement this strategy, we design explicit binary instructions and instrumentation to avoid the reliance on compilers. Evaluation results demonstrate that our approach supports all three different compilers used in experiments (see Table I), and is compatible with future compilers due to its Wasm binary-oriented design.

C2: lacking of scalability. Wasm VMs do not understand the semantics of canaries. Instead, they treat canaries as 32-bit or 64-bit random integers and store them in the managed memory, thereby undermining their effectiveness as our running example in Figs. 2 and 3 shows. Consequently, even though we record inserted canaries with corresponding shadow bytes in shadow memory, the Wasm VM does not recognize their security semantics.

Solution: We utilize the idea of canary instructions [30] to propagate canary semantics from Wasm binaries to VMs. Specifically, we leverage an instruction `canary.insert` for canary insertion and an instruction `canary.check` for sanity checking, respectively. To address the limitation in prior study [30] of only supporting the unmanaged stack, we extend the semantics of canary instructions to support both the VM heap and the shadow memory.

C3: lacking of VM support. The standard Wasm VMs do not understand the semantics of the newly introduced canary instructions.

Solution: To address this challenge, we extend Wasm VMs to materialize these canary instructions. This extension supports shadow memory to record the canary information for both static instrumentation and dynamic vulnerability detection algorithms. Our experiments demonstrate that the modification required is minimal and thus acceptable.

C. Threat Model

Wasm has a rich ecosystem consisting of high-level language support, compilation toolchains, external libraries, and Wasm VMs. The focus of this work is on Wasm memory protection via a Wasm VM-aware approach. Therefore, we make the following assumptions in the threat model for this work.

We assume that the Wasm VMs executing Wasm programs are secure and thus trustworthy. On one hand, extensive security studies have been conducted to secure Wasm VMs [46]. On the other hand, our work is orthogonal to Wasm VM security studies, and thus our approach also benefits that field.

We assume that both source code and Wasm compiler toolchains are unreliable and vulnerable. On the one hand, vulnerabilities in insecure source code may propagate to Wasm binaries [9]. On the other hand, the design and implementation defects of Wasm compiler toolchains may introduce bugs to the generated Wasm binaries.

IV. APPROACH

In this section, we present our approach for effective Wasm memory protection through VM-aware shadow memory. We begin by introducing the design goals (§ IV-A), followed by an overview of its workflow (§ IV-B). We then detail the design and implementation of each component (in § IV-C through § IV-D).

A. Design Goals

We have three primary goals guiding the design of WASHADOW. First, WASHADOW should provide comprehensive protection for Wasm memory against various vulnerabilities, including but not limited to stack and heap overflows. Second, WASHADOW should provide a holistic memory protection for *all* Wasm binary programs, regardless of the source languages or toolchains used to generate those binaries. Third, WASHADOW should be an automatic, end-to-end solution requiring minimal user interventions, while providing a user-friendly interface to assist users in identifying and diagnosing issues.

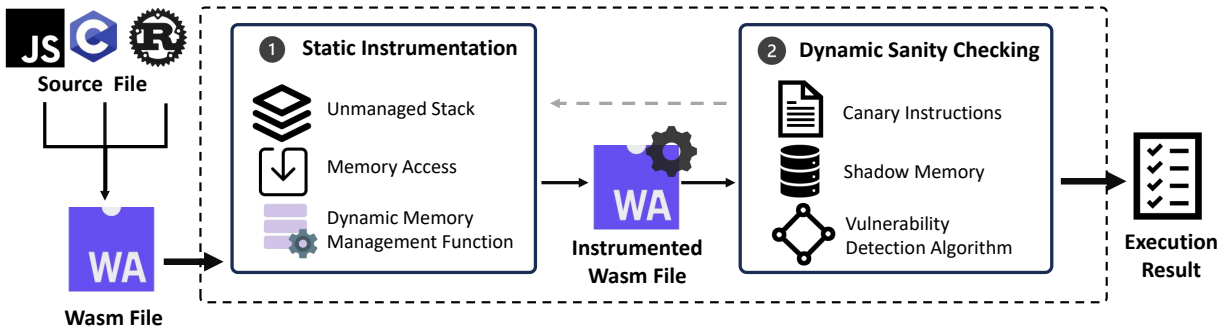


Fig. 4: An overview of WASHADOW's workflow.

B. Overview

With these design goals in mind, we present an overview of WASHADOW's workflow in Fig. 4, which comprises two key components: static instrumentation and dynamic sanity checking. First, the static binary instrumentation (❶) reads a Wasm binary program as input and scans it to insert appropriate instructions at relevant locations, then outputs an instrumented Wasm file as output for subsequent processing. Second, the dynamic sanity checking (❷) takes an instrumented Wasm program as input and performs vulnerability detection during program execution, leveraging linear memory layout information collected during execution and custom-designed detection algorithms.

C. Static Binary Instrumentation

The static instrumentation takes a Wasm binary as input, inserts extended canary instructions at appropriate positions, and outputs an instrumented Wasm binary file for subsequent processing. Specifically, the static instrumentation must handle the unmanaged stack, memory access, and dynamic memory management functions.

Instrumenting the unmanaged stack. The unmanaged stack stores information about local aggregate variables such as buffers for each function. Hence, a buffer overflow vulnerability in the unmanaged stack can overwrite continguous data. If the affected data is security-sensitive, it may present a significant threat to the Wasm program. Therefore, we should instrument the unmanaged stack to prevent potential overflows.

Instrumenting the unmanaged stack involves three key steps, as decipted in Algorithm 1. First, we scan the input Wasm program and construct a control-flow graph (CFG) representation $\mathbb{C}$ for each function $\mathbb{F}$. Second, we normalize each function by adjusting its CFG, to ensure that the resulting CFG $\mathbb{C}'$ has a unique exit block. Finally, for each function, we insert instructions "PreI" in the CFG's entry block and insert instructions "PostI" into CFG's exit block. We present code templates for PreI and PostI in Fig. 5. The two templates are essentillay short code snippets to place and sanitiy check the canaries, respectively, by adjusting the call stack top pointer `sp`.

Algorithm 1: Unmanaged Stack Instrumentation.

Input: $\mathbb{M}$: a WebAssembly module
Output: $\mathbb{M}$: the instrumented WebAssembly module

```
Function UnmanagedStkInstrument(𝕄):
    for each function 𝔽 in 𝕄 do
        ℂ = makeCfg(𝔽);
        ℂ′ = adjustCfg(ℂ);
        instrument(ℂ′, PreI, PostI);
    return 𝕄;
```

```
// template for PreI
  global.get sp // get stack pointer
  i32.const 16
  i32.sub
  global.tee sp // subtract sp with 16
  canary.insert // insert canary
// template for PostI
  global.get sp
  canary.check // check canary at sp
  global.get sp
  i32.const 16
  i32.add
  global.set sp // add sp with 16
  return
```

Fig. 5: Code templates used to instrument the unmanaged stack.

Instrumenting memory accesses. The static data and heap are two important memory regions in the linear memory (see Fig. 1) that must be protected, because buffer overflows can manifest in these regions. On the one hand, the static data stores global buffers that can be overflowed. On the other hand, the heap comprises dynamically allocated memory which can lead to heap overflows. We protect these memory regions by instrumenting memory accesses, as these memory are accesses by the the same group of memory accessing instruction in Wasm .

We instrument memory accesses in three steps, as decipted by the function `MemoryAccesssInstrumentation` in Algorithm 2. First, we traverse all the instructions $\mathbb{I}$ in the Wasm code to process each memory access instruction i

Algorithm 2: Memory and Function Instrumentation.

Input: $\mathbb{I}$: instructions in a WebAssembly module
Output: $\mathbb{I}$: the instrumented WebAssembly instructions

```
1  Function MemoryAccessInstrument (𝕀):
2      for each instruction i ∈ 𝕀 do
3          if isMemoryAccessInstr(i) then
4              o = i.offset;
5              b = i32.const o;
6              if isLoadInstr(i) then
7                  a = "CheckMemoryLoad(o, b)";
8              else if isStoreInstr(i) then
9                  a = "CheckMemoryStore(o, b)";
10             append ([a], i);
11     return 𝕀;
12 Function MemFunctionInstrument (𝕀):
13     𝕋 = [f1 ↦ (b1, a1), ..., fn ↦ (bn, an)];
14     for each instruction i ∈ 𝕀 do
15         if i is a function call f() and f ∈ 𝕋 then
16             (b, a) = 𝕋[f];
17             i = append(b, i);
18             i = append(i, a);
19         return 𝕀;
```

(line 3). We then instrument each memory instruction with sanity checks for memory accesses. Specifically, we generate a fresh sanity checking invocation (line 7 or 9), based on the the memory base address b and the offset o from the instruction (line 4 and 5). Finally, we insert the generated santity checking instruction before the original instruction i, ensuring the expected memory property are enforced before i's execution.

Instrumenting dynamic memory management functions. Dynamic memory management functions, such as `malloc` and `free`, manage heap memory for allocation, adjustment, and reclaimation. Imporper usages of such functions can lead to subtle memory vulnerabilities such as double-free and use-after-free. We instrument dynamic memory management functions to detect and mitigate such vulnerabilities.

Our instrumentation encomprises three primary steps, as decipted by the function `MemFunctionInstrument` in Algorithm 2. A key challenge in instrumenting dynamic memory management functions is to deal their diversity. To address this challenge, we first develop a table $\mathbb{T}$ mapping each memory management function f_i to its corresponding instrumentation code (b_i, a_i), for $1 \leq i \leq n$. The tuple (b_i, a_i) specifies code snippets to be instrumented before and after the corresponding function call $f_i()$, respectively. Take the allocation function `malloc` as an example, we put the mapping `malloc` $\mapsto$ (`PreMalloc`, `PostMalloc`) into the table $\mathbb{T}$, where the code snippets for (`PreMalloc`, `PostMalloc`) are decipted in the Fig. 6. Similarly, we put other dynamic memory management functions, such as `free` and `calloc`,

```
// template for PreMalloc
  i32.const 32
  i32.add
  local.tee size // add size with 32
  local.get size
// template for PostMalloc
  call CheckMalloc // sanity check "malloc"
  local.tee addr
  canary.insert // insert canary
  local.get addr
  i32.add // add address and size
  i32.const 16
  i32.sub
  canary.insert
  local.get addr
  i32.const 16
  i32.add // return address+16
```

Fig. 6: Code template used to instrument the alloacation function `malloc`.

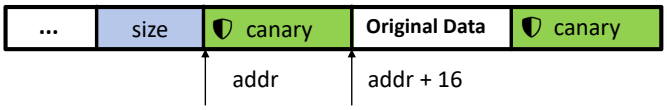


Fig. 7: Heap chunks after instrumentation.

into the table $\mathbb{T}$. An essential property of the table $\mathbb{T}$ is its extensibility to add new functions, hence facilitating the addition of user-customized memory management functions.

With this extensible table $\mathbb{T}$, we next traverse each instruction i in the instruction sequence $\mathbb{I}$. When the candidate instruction is a function call $f()$ and the callee f belongs to the table $\mathbb{T}$, we retrieve its protection code snippets (b, a) from the table $\mathbb{T}$, then instrument them before and after the instruction i, respectively.

The instrumented code snippets are essential to enforce the desired memory protection and thus deserve further discussion. Consider the code snippets for the allocation function `malloc` in the Fig. 6. The code `PreMalloc` first increases the allocation `size` by 32 bytes to reserve memory space for the canary to be inserted subsequently. After the memory allocation function `malloc` returns a memeory address `addr`, we first sanity check its validity by invoking *CheckMemoryAlloc* to validate the allocation. Next, we insert two canaries before and after the allocated memory, by invoking two canary-related instructions `canary.insert`.

To illustrate the problem, Fig. 7 shows the heap layout in linear memory after instrumentation. The algorithm increases allocation size by 32 bytes, and inserts canaries before and after original data, enabling us to detect both overflows and underflows.

D. Dynamic Sanity Checking

Dynamic sanity checking enhances the underlying Wasm VM to monitor and protect the memory during program execution. We first extend the Wasm memory with a shadow memory, then we introduce support for the new canary instruc-

TABLE II: Detailed representation of shadow memory.

Shadow Bytes (High 4 Bits)	Shadow Bytes (Low 4 Bits)	Region and Privileges
1 x x x	x x x x	constant data, read only
0 0 1 x	x x x x	canary on stack, read only
0 0 0 x	x x x x	accessible address on stack
0 1 1 1	x x x x	canary on heap, read only
0 1 0 1	0 0 0 0	accessible address on heap
0 1 0 1	k(0<k<16)	previous k byte accessible on heap
0 1 0 0	x x x x	unallocated space on heap, inaccessible

tions `canary.insert` and `canary.check`, along with a set of custimized vulnerability detection algorithms. Below, we detail the design of these three components.

Shadow memory. We utilize a shadow memory to record the memory layout of linear memory to precisely track the memory status, facilitating the detection of various memory vulnerabilities.

Since shadow memory incurs overhead by taking extra memory space, thus a challenge we face is how to design shadow memory layout to effectively represent linear memory information while saving spaces. To address this challenge, we utilize a scale mapping to create a bidirectional map between linear memory and shadow memory, reducing the space overhead of shadow memory. Specifically, we map each 16 bytes in linear memory to 1 byte in the shadow memory as decipted in Table II, since the memory address returned by the underlying memory allocator is aligned in a granularity of 16 bytes. We utilize the high four bits in the shadow byte to represent the memory regions along with the access privileges. For example, a highest bit of 1 in the shadow byte represents the read-only constant data region (the first row), whereas a three bit sequence `001` represents a read-only canary in the stack region (the second row).

Canary instructions. To address the limitations of traditional canary protection to Wasm, we leverage two canary-oriented instructions [30] to provide secure and effective canary protection on Wasm, *i.e.*, `canary.insert` and `canary.check`. However, our work differs from existing study in that we design the the operational semantics of `canary.insert` and `canary.check` to support protecting the whole linear memory including heap and shadow memory, whereas existing work is limited in protecting only the unmanged stack.

To formulate our design rigorously, we present the operational semantics of these two instructions in Fig. 8. Specifically, the `canary.insert` instruction takes an $i32$ value as the address, and generates a 16-byte random number as the canary to be inserted into linear memory. This instruction also records the addresses and values of the inserted canaries for subsequent verification, before updating the shadow memory. The `canary.check` instruction accepts an $i32$ value as the address, then retrieves the canary from memory and compares it against the saved value to determine whether a buffer overflow occurred, before eliminating canary in shadow memory of this address.

Vulnerability detection algorithms. We design customized vulnerability detection algorithms to detect memory vulnerabilities in Wasm programs, by leveraging instrumented instructions along with information in shadow memory.

Algorithm 3: Vulnerability Detection.

Input: b: base address for memory access; o: offset for memory access; s: required size of memory allocation; a: heap address of memory allocation and free

```
Function CheckMemoryLoad(b, o):
    a = b + o;
    s_a = CalShadowAddr(a);
    s_byte = ShadowMemory[s_a];
    CheckLoadValidity(a, s_byte);
Function CheckMemoryStore(b, o):
    a = b + o;
    s_a = CalShadowAddr(a);
    s_byte = ShadowMemory[s_a];
    CheckStoreValidity(a, s_byte);
Function CheckMemoryAlloc(a, s):
    UpdateShadowMemory(a + 16, s - 32);
Function CheckMemoryFree(a, s):
    s_a = CalShadowAddr(a);
    s_byte = ShadowMemory[s_a];
    CheckFreeValidity(a, s, s_byte);
    UpdateShadowMemory(a, s);
```

A fundamental challenge in designing vulnerability detection algorithms lies in the diversity of security requirement for different operations. For example, a memory load requires its address to be valid, whereas a memory reclaimation operation necessitates that its address has not been reclaimed. To address this challenge, we design algorithms following a heuristics-based approach as presented in Algorithm 3. Specifically, the function `CheckMemoryLoad` validate the memory access through a base address b and an offset o. This function first calculates a target memory address a along with shadow memory address `s_a`, then retrieves its shadow memory content `s_byte` that encodes its memory region along with privileges. Next, the function validates the memory load operation by invoking an auxilary function `CheckLoadValidity` with the the memory address `a` and shadow byte `s_byte` as arguments. Similarly, the functions `CheckMemoryStore` `CheckMemoryAlloc` and `CheckMemoryFree` are used to check the store, allocation and deallocation, respectively. During the process of validity checking, the status of the shadow memory are updated to reflect the effects of memory operation.

Below, we highlight some technical details of how we detect specific vulnerabilities decipted in Table I.

Constant data overwriting. Constant data resides in linear memory addresses ranging from 0 to *data_end*, which can be overwritten by store instructions. WASHADOW detects a constant data overflow by comparing the highest bit of the

$$\frac{\Sigma(canary) = v1}{\Sigma, \Gamma, R \vdash M(addr) = v1 \quad R(addr) = v1 \quad SM(addr/16) = canaried}[\text{canary.insert (i32.const addr)}]$$

$$\frac{R(addr) = v1 \quad M(addr) = v}{\Sigma, \Gamma, R \vdash SM(addr/16) = uncanaried \quad isequal(v1, v)}[\text{canary.check (i32.const addr)}]$$

Fig. 8: Operational semantics of `canary.insert` and `canary.check`.

shadow byte of starting address or ending address against 1.

Stack-based buffer overflow. WASHADOW detects stack-based buffer overflows by first comparing the high 3 bits of the shadow byte of starting address with `001`, followed by comparing the high 3 bits of the shadow byte of ending address with `001`.

Heap overflow. WASHADOW utilizes the following three steps to detect a heap overflow. First, it compares the high 4 bits of the shadow byte of starting address with `0111`. Then, it compares the high 4 bits of the shadow byte of accessed memory with 0101 and the low 4 bits are less than the size to be accessed. Finally, it compares the high 4 bits of the shadow byte of starting address with 0101 and the high 4 bits of the shadow byte of ending address with 0100.

Invalid heap memory access. WASHADOW detects use-after-free (UAF) and double-free (DF) by comparing the high 4 bits of the shadow byte of starting address with `0100`.

Null-pointer dereference. WASHADOW detects null-pointer dereference by comparing the address and offset against 0.

Memory leak. WASHADOW detects memory leaks by traversing the shadow memory of heap upon program termination. Specifically, it compares the high 4 bits of shadow bytes against 0111 or 0101.

E. Prototype Implementation

To validate our approach, we designed and implemented a software prototype for WASHADOW, consisting of three components. First, we implemented the static instrumentation by adapting and extending the frontend of Fuzzm [21], a state-of-the-art tool for Wasm fuzzing rather than for holistic memory protection as this work does. Second, we implemented the VM extension by modifying and extending the Wasm3 [47], a widely used Wasm VM. We choose Wasm3 for two reasons: 1) Wasm3 uses interpretive execution without Ahead-Of-Time (AOT) or Just-In-Time (JIT) compilation techniques, providing greater versatility and avoiding the unnecessary impacts introduced by these methods; and 2) Wasm3 is a lightweight VM that can be easily modified or integrated as a library into other projects, simplifying implementation and evaluation. Third, we implemented vulnerability detection algorithms in C, comprisings 1,132 lines of code.

V. EVALUATION

To understand the effectiveness of WASHADOW, we evaluate it on micro-benchmarks and real-world Wasm programs. Specifically, our evaluation aims to answer the following questions:

TABLE III: Real-world benchmarks to evaluate WASHADOW.

Project	Domain	Stars (k)	Vulnerabilities
libpng	Image processing	1.1	25
libcurl	Network transfer	33.8	26
libtiff	TIFF image	-	21
flac	Audio encode	1.5	13

RQ1: Effectiveness. Given that WASHADOW is designed to provide memory protection for Wasm, is WASHADOW effective in achieving this goal?

RQ2: Usefulness. As a tool designed to enhance the security of Wasm programs, is WASHADOW capable to detect memory vulnerabilities in real-world applications?

RQ3: Overhead. WASHADOW's utilization of static instrumentation will inevitably increase the code size and execution time of the Wasm programs. Therefore, what overhead does WASHADOW introduce?

RQ4: Compare with existing studies. Does WASHADOW outperform existing Wasm protection methods?

A. Experimental Setup

All experiments and measurements are performed on a server with one 8 physical Intel i7 core CPU and 16 GB of RAM running Ubuntu 20.04.

B. Datasets

We conducted the evaluation using two datasets: 1) a set of micro-benchmarks, consisting of 15 vulnerable programs we derived from real-world CWEs [48]; and 2) a set of real-world benchmarks, comprising 4 real-world Wasm applications.

Micro-benchmarks. We constructed a set of micro-benchmarks consisting of 15 test cases adapted and compiled from CWE-658 [49], which contains normal vulnerabilities in C programs as well as those vulnerabilities WASHADOW could detect. To better hightlight the significance of vulnerability detection, we have simplified some of the original buggy code by removing irrelevant fragments. As shown in the second column of Table IV, these micro-benchmarks consist of various vulnerabilities, including stack-based buffer overflow, double-frees, and use-after-frees, among others. We created these micro-benchmarks from C/C++ sources because manually constructing Wasm test cases by directly composing Wasm binary instructions is both labor-intensive and error-prone. Furthermore, introducing memory vulnerabilities into Wasm binaries manually remains challenging, as Wasm binaries must adhere to Wasm's rigorous semantic specification [50].

TABLE IV: Experimental results on micro-benchmarks.

Test Case	Vulnerability Type	LoC BI / LoC AI	LoC Overhead	IT (s)	Run time BI (s) / EXE Time AI (s)	Run Time Overhead	WASHADOW	Fuzzm
1	CDO_1	5455 / 6805	24.7%	0.028	0.046 / 0.072	56.5%	✔	✘
2	CDO_2	5481 / 6831	24.6%	0.022	0.048 / 0.089	85.4%	✔	✘
3	DF_1	9268 / 11730	26.6%	0.034	0.054 / 0.107	98.1%	✔	✘
4	DF_2	4230 / 5579	31.9%	0.019	0.050 / 0.095	90.0%	✔	✘
5	HBO_1	9284 / 11716	26.2%	0.026	0.051 / 0.112	119.6%	✔	✔
6	HBO_2	9233 / 11629	25.9%	0.041	0.055 / 0.108	96.4%	✔	✔
7	HBO_3	8301 / 10409	25.4%	0.038	0.044 / 0.070	59.1%	✘	✘
8	HBO_4	3783 / 4931	30.3%	0.020	0.043 / 0.087	102.3%	✔	✔
9	HBO_5	8301 / 10409	25.4%	0.033	0.049 / 0.073	49.0%	✘	✘
10	ML_1	9609 / 12160	26.5%	0.048	0.063 / 0.133	111.1%	✔	✘
11	ML_2	9589 / 12146	26.6%	0.044	0.047 / 0.097	106.4%	✔	✘
12	NPD	9307 / 11762	26.4%	0.051	0.059 / 0.142	140.7%	✔	✘
13	SBO_1	5717 / 7147	25.0%	0.023	0.056 / 0.126	125.0%	✔	✔
14	SBO_2	5553 / 6931	24.8%	0.015	0.051 / 0.161	215.7%	✔	✔
15	UAF	9075 / 11485	26.6%	0.046	0.057 / 0.155	171.9%	✔	✘

LoC: Line of Code; BI: Before Instrumentation; AI: After Instrumentation; IT: Instrumentation Time.

Real-world applications. Our selection of real-world Wasm applications is guided by three principles. First, the chosen projects should be widely used and open source, enabling root cause analysis using the available source code. To this end, we select open-source projects from GitHub and measure their popularity based on the number of stars. Second, the projects must be successfully compilable to Wasm without requiring special support from specific tools. Therefore, we focus on real-world projects that can be compiled by the standard Wasm compiler toolchain, Emscipten. Finally, the chosen projects should contain confirmed memory vulnerabilities, which are essential for evaluating the effectiveness of WASHADOW.

As a result, we selected four projects from different domains: libpng, libcurl, libtiff, and flac, as presented in Table III. Specifically, libpng is the official library to process PNG files. Libcurl is a widely used URL transmission library facilitating network requests. Libtiff is an open-source library for manipulating TIFF files. Flac is a lossless audio codec tool. Moreover, we collected a total of 85 previously discovered CVEs across these four projects. We compiled each project into its corresponding Wasm binary and ensured successful execution on the Wasm VM.

C. Evaluation Metrics

We use the *precision* and *recall* metrics to measure the effectiveness of WASHADOW. The definition of these two metrics is given in the equation 1.

$$precision = \frac{tp}{tp + fp} \quad recall = \frac{tp}{tp + fn} \tag{1}$$

In the equation, we use tp, fp, fn to denote true positives, false positives, and false negatives, respectively. We also compute the $F1$ score according to equation 2.

$$F1\,score = \frac{2 \times precision \times recall}{precision + recall} \tag{2}$$

F1 score can reflect the overall accuracy of analysis tools.

D. RQ1: Effectiveness

To answer RQ1, we first applied WASHADOW to micro-benchmarks to assess its effectiveness. We compiled the micro-benchmarks into their corresponding Wasm binaries using the Emscripten compiler [20], and then employed WASHADOW to detect vulnerabilities in each binary.

The experimental results are presented in the 8th column (i.e., **WASHADOW**) of Table IV. Among the 15 benchmarks, WASHADOW successfully protected 13 but failed on 2 test cases. Consequently, the recall of WASHADOW is 86.7%, and the precision is 100%, resulting in an F1 score of 92.9%. These results demonstrate that WASHADOW is effective in protecting Wasm binaries.

To further investigate the root causes for the above 2 test case that WASHADOW failed to provide protection, we conducted a manual inspection of their source code. This inspection revealed a key root cause: the Wasm compiler will insert extra padding after a buffer for alignment purposes, if the length of the protected buffer is not divisible by the size of a machine word (i.e., 4 or 8 bytes). Therefore, when a buffer overflow only overwrites the padding bytes without reaching the canary, WASHADOW does not detect it. However, such buffer overflows are considered benign, as they overwrite padding bytes that contain no sensitive data.

E. RQ2: Usefulness

To answer RQ2 by investigating its practical usefulness, we apply WASHADOW to real-world benchmarks. We first compile each benchmark to Wasm, then apply WASHADOW to the generated binaries. We record vulnerabilities detected by WASHADOW, then compare them with the ground truth for these benchmarks.

The experimental results for the real-world benchmarks are presented in Table V. Specifically, for each benchmark, we present the number of vulnerabilities WASHADOW detected (#Deteced) from the ground truth (#GT). In summary, WASHADOW successfully detected 76 out of 85 vulnerabilities across all these real-world benchmarks, missing 9. These

TABLE V: Experimental results on real-world benchmarks.

Benchmark	libpng	libcurl	libtiff	flac
#Detected / #GT	23 / 25	22 / 26	21 / 21	10 / 13

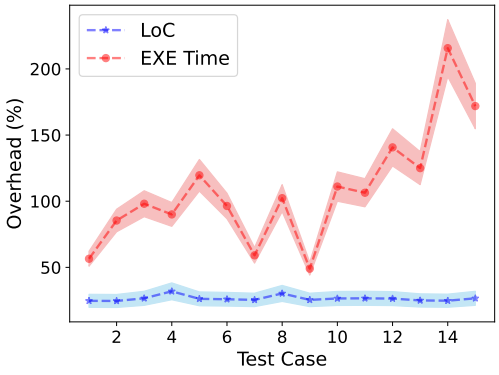


Fig. 9: The file sizes and execution time changes introduced by WASHADOW.

results yield a recall of 89.4%, a precision of 100%, and an F1 score of 94.4%, demonstrating that WASHADOW is useful to detect vulnerabilities in real-world large projects.

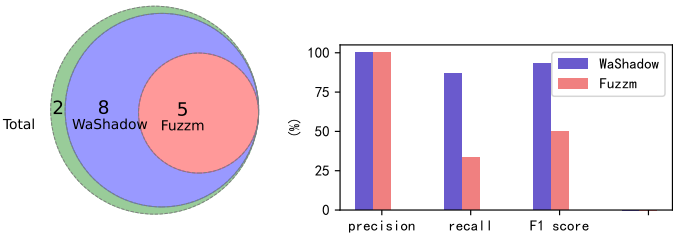
Furthermore, we investigated the root causes of the 9 failures. Through manual inspection of the relevant benchmarks, we identified that these failures were also caused by buffer padding, as outlined in Section V-D.

F. RQ3: Overhead

To answer RQ3, we investigate the overhead WASHADOW may introduce, including 1) time spent on binary instrumentation for each program; 2) code size increase; and 3) execution time penalty. To this end, we first compile the micro-banchmark to Wasm binaries and record the code size, then run each binary 20 rounds to calculate the average execution time, following prior work [24]. We then apply WASHADOW to generate protected Wasm binaries, then repeat the above process on each of these binaries. Finally, we calculate the changes in code size and execution time.

We present experimental results in Table IV. The third column shows the changes in code size before and after the static instrumentation in terms of line of code (LoC), respectively, while the fourth column shows the code size increases, ranging from 24.6% to 31.9%. Similarly, the sixth column presents the execution time of the relevant Wasm binaries before and after the binary instrumentation, while the seventh column shows the execution time increases, ranging from 49.0% to 215.7%.

In summary, we present the changes in file size and execution time introduced by WASHADOW in Fig. 9, which average 26.5% and 108.5%, respectively. These results align with prior work [24] [51], demonstrating that the overhead introduced by WASHADOW is acceptable.



(a) Bug detection capability. (b) Evaluation metrics.

Fig. 10: A comparison of WASHADOW and state-of-the-art tool Fuzzm.

Furthermore, the fifth column shows the time WASHADOW spent to perform instrumentation. WASHADOW takes less than 0.05 seconds to process each test case, demonstrating its efficiency in completing binary instrumentation.

G. RQ4: Compare with Existing Work

To understand WASHADOW's technical advantages, we compare WASHADOW with the state-of-the-art Wasm security tool Fuzzm [21]. We first run both WASHADOW and Fuzzm on the same set of micro-benchmarks, and then compare their execution results.

The experimental results are presented in the last two columns of Table IV. Among all the 15 vulnerabilities in 7 categories, WASHADOW successfully detects 13 vulnerabilities in 7 categories, whereas Fuzzm only detects 5 vulnerabilities in 2 categories.

Furthermore, we compare WASHADOW and Fuzzm in terms of their detection capabilities and evaluation metrics, as shown in Fig. 10. First, as the Venn diagram in Fig. 10a shows, WASHADOW detects all the 5 vulnerabilities detected by Fuzzm, plus an additional 8 vulnerabilities that Fuzzm misses. This demonstrates WASHADOW's superior detection capacity. Second, as the histogram in Fig. 10b shows, while WASHADOW and Fuzzm both achieve a precision 100%, WASHADOW's recall (86.7%) significantly surpasses Fuzzm's (33.3%). Overall, WASHADOW achieves a higher F1 score of 92.9% compared to Fuzzm's 50.0%, showcasing WASHADOW's greater effectiveness in detecting vulnerabilities.

H. Case Study

To better understand the capability of WASHADOW, we present a case study of how WASHADOW protects real-world Wasm programs. Specifically, we apply WASHADOW to our running example in Figs. 2 and 3 from Section III-A.

To detect and protect the memory vulnerabilities in this example, WASHADOW first instruments it using the canary instruction `canary.insert`. As shown in Fig. 11, the instrumentation will insert a canary at the bottom of the call stack (i.e., the canary right to the `main` frame), hence protecting the whole stack as well as the adjacent heap. In the meanwhile, WASHADOW instruments each function by

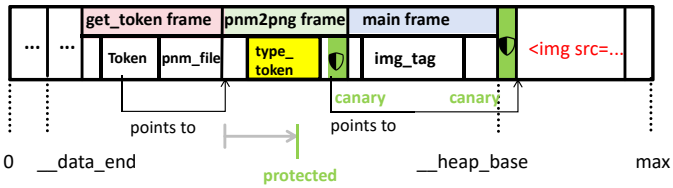


Fig. 11: Wasm linear memory after protected by WASHADOW.

inserting `PreI` and `PostI` code segments at entry and exit block (see Fig. 5), respectively. Therefore, canaries will be placed at the bottom of each call stack frame during function invocations (i.e., the canary in `pnm2png`'s frame). These canaries are used to detect buffer overflow in this function (e.g., the `token` buffer).

As a result, WASHADOW's protection will promptly stop an adversary who attempts to trigger buffer overflows by overwriting a large string into the buffer. Similarly, WASHADOW can detect vulnerabilities in other memory regions including static data area or heap, thus providing a holistic approach for Wasm memory protection.

VI. DISCUSSION

In this section, we discuss some possible enhancements to this work, along with directions for future work.

Higher accuracy. Although our approach provides an effective method for protecting Wasm linear memory, the recorded shadow information in the shadow memory is limited. Consequently, vulnerabilities requiring more fancy shadow information cannot be detected. For example, buffer overflows that do not overwrite the canary in memory remain undetectable. To address this issue, one possible approach we can adopt is to leverage a more fine-grained shadow memory model into WASHADOW. Specifically, we can record the starting and ending addresses for a buffer in the shadow memory, to aid in the overflow detection. We leave this a future direction.

Other vulnerabilities. Even though WASHADOW effectively detects various memory vulnerabilities as shown by our experimental results, it will miss certain memory vulnerabilities. Specifically, WASHADOW, by its current design, does not detect memory vulnerabilities caused by race conditions, that is, imporper memory accesses by concurrent threads. To address this issue, we can leverage recent studies [52] to extend WASHADOW's design to support concurrency semantics. Moreover, Wasm recently introduces a design of concurrency features [53], with which we plan to triage the WASHADOW extensions.

Canary implementation. WASHADOW modifies Wasm VMs to support canary instructions and shadow memory. While this approach is effective for open-source Wasm VMs, it is difficult to applied to commercial VMs with no source code. To address this issue, we can extend WASHADOW with a dynamic program analysis approach [24], to implement canary instructions as hooks and dynamic analysis requiring no VM modifications. We leave this an important direction to explore.

VII. RELATED WORK

In recent years, there has been substantial research on Wasm vulnerabilities and their associated security enhancements.

Wasm security. Despite Wasm's design goal of security, Wasm programs still susceptible to vulnerabilities due to its design defects. Lehmann et al. [9] [11] conducted research on a set of Wasm binaries, and revealed that Wasm binaries are often more exploitable than native binaries. Romano et al. [10] conducted an empirical study of bugs in Wasm compilers and VMs, based on the bug reports from GitHub repositories, and concluded the development challenges and bug causes of Wasm compilers. Quentin et al. [12] [13] studied security risks in Wasm binaries and compiler, respectively, and revealed the lacking of protection mechanisms in linear memory and compiler protection of Wasm.

However, a significant limitation of these studies is that they do not propose memory protection mechanisms like the one we present in this work.

Wasm security enhancement. There have been a lot of works on Wasm security enhancement. Narayan et al. [27] propose Swivel, a new compiler framework for hardening Wasm binaries against Spectre attacks. Jiang et al. [54] proposed WasmFuzzer for fuzzing Wasm VMs. Song et al. presented metaSafer [22] to protect heap metadata in Wasm linear memory. Lehmann et al. [21] presented Fuzzm to detect overflows in the Wasm stack and heap with the aid of canaries. Lei et al. [23] utilized MPK to protect Wasm linear memory in a function granularity. Arteaga et al. [25] proposed the CROW system which statically transforms code using code diversification technology.

However, a key difference between these studies and this work is that existing studies has not proposed a thorough and holistic protection solution. In contrast, we propose a solution that leverages the capability of the underlying Wasm VMs. By introducing canary instructions and shadow memory, our approach provides comprehensive protection for the entire Wasm memory.

VIII. CONCLUSION

In this work, we present an effective approach for protecting Wasm memory through VM-aware shadow memory. Our approach leverages information from the underlying Wasm VM to design shadow memory. We first design a static instrumentation to instrument Wasm binaries with two canary instructions. Next, we design dynamic sanity checking, to detect vulnerabilities using a set of detection algorithms. We implement a prototype called WASHADOW and conduct experiments to evaluate its effectiveness, usefulness, and overhead. The evaluation results demonstrate that WASHADOW provides effective protection against various memory vulnerabilities with acceptable overhead, outperforming state-of-the-art studies. Overall, this work represents a new step towards security enhancement of Wasm, making Wasm's promise as a safe language a reality.

REFERENCES

[1] "WebAssembly," https://webassembly.org/.

[2] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Barcelona Spain: ACM, Jun. 2017, pp. 185–200.

[3] C. Watt, "Mechanising and verifying the WebAssembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Los Angeles CA USA: ACM, Jan. 2018, pp. 53–65.

[4] "Security - WebAssembly," https://webassembly.org/docs/security/.

[5] "Execution — WebAssembly 2.0 (Draft 2024-04-28)," https://webassembly.github.io/spec/core/exec/index.html.

[6] "Structure — WebAssembly 2.0 (Draft 2024-04-28)," https://webassembly.github.io/spec/core/syntax/index.html.

[7] S. Shillaker and P. Pietzuch, "FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing."

[8] A. A. Monrat, O. Schelén, and K. Andersson, "A Survey of Blockchain From the Perspectives of Applications, Challenges, and Opportunities," *IEEE Access*, vol. 7, pp. 117 134–117 151, 2019.

[9] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly."

[10] A. Romano, X. Liu, Y. Kwon, and W. Wang, "An Empirical Study of Bugs in WebAssembly Compilers," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Melbourne, Australia: IEEE, Nov. 2021, pp. 42–54.

[11] A. Hilbig, D. Lehmann, and M. Pradel, "An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases," in *Proceedings of the Web Conference 2021*. Ljubljana Slovenia: ACM, Apr. 2021, pp. 2696–2708.

[12] Q. Stiévenart, C. De Roover, and M. Ghafari, "The Security Risk of Lacking Compiler Protection in WebAssembly," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2021, pp. 132–139.

[13] Q. Stievenart, C. De Roover, and M. Ghafari, "Security Risks of Porting C Programs to Webassembly," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, Apr. 2022, pp. 1713–1722.

[14] "Github-WebAssembly/design." https://github.com/WebAssembly/design.

[15] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities."

[16] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. Singapore Republic of Singapore: ACM, Apr. 2015, pp. 555–566.

[17] N. Burow, X. Zhang, and M. Payer, "Shining Light On Shadow Stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 985–999.

[18] J. Zhou, Y. Du, Z. Shen, L. Ma, J. Criswell, and R. J. Walls, "Silhouette: Efficient Protected Shadow Stacks for Embedded Systems."

[19] S. Sinnadurai, Q. Zhao, and W.-F. Wong, "Transparent Runtime Shadow Stack: Protection against malicious return address modifications."

[20] "Emscripten-core/emscripten: Emscripten: An LLVM-to-WebAssembly Compiler," https://github.com/emscripten-core/emscripten.

[21] D. Lehmann, M. T. Torp, and M. Pradel, "Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly," Oct. 2021.

[22] S. Song, S. Park, and D. Kwon, "metaSafer: A Technique to Detect Heap Metadata Corruption in WebAssembly," *IEEE Access*, vol. 11, pp. 124 887–124 898, 2023.

[23] H. Lei, Z. Zhang, S. Zhang, P. Jiang, Z. Zhong, N. He, D. Li, Y. Guo, and X. Chen, "Put Your Memory in Order: Efficient Domain-based Memory Isolation for WASM Applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. Copenhagen Denmark: ACM, Nov. 2023, pp. 904–918.

[24] D. Lehmann and M. Pradel, "Wasabi: A Framework for Dynamically Analyzing WebAssembly," Aug. 2018.

[25] J. C. Arteaga, O. Malivitsis, O. V. Pérez, B. Baudry, and M. Monperrus, "CROW: Code Diversification for WebAssembly," in *Proceedings 2021 Workshop on Measurements, Attacks, and Defenses for the Web*, 2021.

[26] J. Sun, D. Cao, X. Liu, Z. Zhao, W. Wang, X. Gong, and J. Zhang, "SELWasm: A Code Protection Mechanism for WebAssembly," in *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. Xiamen, China: IEEE, Dec. 2019, pp. 1099–1106.

[27] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," p. 19.

[28] "Clang: a C language family frontend for LLVM," https://clang.llvm.org/.

[29] "The LLVM Compiler Infrastructure Project," https://llvm.org/.

[30] Z. Zhang, W. Zheng, B. Hua, Q. Fan, and Z. Pan, "VMCanary: Effective Memory Protection for WebAssembly via Virtual Machine-assisted Approach," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. Chiang Mai, Thailand: IEEE, Oct. 2023, pp. 662–671.

[31] "WASI —," https://wasi.dev/.

[32] "V8 JavaScript engine," https://v8.dev/.

[33] "Safari," https://www.apple.com/safari/.

[34] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 225–236.

[35] "Edge programming with Rust and WebAssembly — Fastly," https://www.fastly.com/blog/edge-programming-rust-web-assembly.

[36] "Lucet Takes WebAssembly Beyond the Browser — Fastly," https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, Mar. 2019.

[37] "WebAssembly on Cloudflare Workers," http://blog.cloudflare.com/webassembly-on-cloudflare-workers/, Oct. 2018.

[38] "wasmCloud," https://wasmcloud.com/.

[39] "Fastly," https://learn.fastly.com/edgecompute-faster-simpler-secure-serverless-code.

[40] "WasmEdge," https://wasmedge.org/.

[41] "Deno — A modern runtime for JavaScript and TypeScript," https://deno.com/runtime.

[42] N. Carlini and D. Wagner, "{ROP} is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 385–399.

[43] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, vol. 13, no. 1, pp. 4:1–4:40, Nov. 2009.

[44] "NVD - CVE-2018-14550," https://nvd.nist.gov/vuln/detail/CVE-2018-14550.

[45] C. Cowan, C. Pu, D. Maier, J. Walpole, and P. Bakke, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks."

[46] Y. Wang, Z. Zhou, Z. Ren, D. Liu, and H. Jiang, "A Comprehensive Study of WebAssembly Runtime Bugs," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2023, pp. 355–366.

[47] "Wasm3," Wasm3 Labs, May 2023.

[48] "CWE - Common Weakness Enumeration," https://cwe.mitre.org.

[49] "CWE - CWE-658:Weaknesses in Software Written in C (4.14)," https://cwe.mitre.org/data/definitions/658.html.

[50] W. C. Group and A. Rossberg, "WebAssembly Specification."

[51] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318.

[52] C. Watt, A. Rossberg, and J. Pichon-Pharabod, "Weakening webassembly," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, Oct. 2019.

[53] "Github-WebAssembly/threads: Threads and Atomics in WebAssembly." https://github.com/WebAssembly/threads.

[54] B. Jiang, Z. Li, Y. Huang, Z. Zhang, and W. K. Chan, "WasmFuzzer: A Fuzzer for WasAssembly Virtual Machines," in *The 34th International Conference on Software Engineering and Knowledge Engineering*, Jul. 2022, pp. 537–542.