VMCanary: Effective Memory Protection for WebAssembly via Virtual Machine-assisted Approach

Ziyao Zhang 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
zhangziyao21@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—WebAssembly is an emerging secure programming language and portable instruction set architecture, and has been deployed in diverse security-critical scenarios due to its safety advantages. However, WebAssembly's linear memory is still vulnerable to buffer overflows due to the lack of effective protection mechanism, defeating its security guarantees.**

In this paper, we present VMCanary, the first framework for effective WebAssembly memory protection, by leveraging a canary approach but with the aid from WebAssembly virtual machines (VMs). Our key idea is that, due to the fact that WebAssembly is a managed programming language to be executed by underlying WebAssembly VMs, the VMs must understand any protection mechanisms already enforced in programs. With this key idea, we first propose the concept of canary in code, which is like a traditional canary in data but whose semantics is understandable by underlying WebAssembly VMs. To realize this kind of canary, we introduced two novel WebAssembly instructions by defining their semantics. Furthermore, we designed an instrumentation for WebAssembly binaries to instrument these two instructions automatically, hence no sources and compiler toolchain modifications are required. We have implemented a software prototype for VMCanary, and have conducted extensive experiment to evaluate it on micro benchmarks and 59 real-world CWEs. Experimental results demonstrated that VMCanary is effective in protecting Wasm memory with negligible overhead (3% on average).

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

I. Introduction

WebAssembly [1] (Wasm) is a novel binary instruction set architecture and code distribution format [2], designed with the goals of security, efficiency, and portability. Specifically, to achieve its security design goal, Wasm incorporates a large spectrum of secure language designs (*e.g.*, strong type systems [2], rigorous operational semantics [3], software fault isolation [4], secure control flow [5], and linear memory [6]). In light of Wasm's security promise, recent years have witnessed the successful deployments of Wasm in diverse security-critical domains such as edge computing [7], smart contracts [8], among others [9]. Hence, given its security design goal and wide adoptions, Wasm programs should be reliable and trustworthy.

Despite the urgent need for security and reliability, recent studies [10] [11] [12] [13] [14] have demonstrated that Wasm programs are still vulnerable and exploitable, due to the defects in Wasm's memory model design. Specifically, to protect function call stacks against buffer overflow attacks [15] [16] [17], Wasm introduced a novel design of linear memory containing a *data stack* to store aggregated local variables (*e.g.*, buffers) in a function. In the meanwhile, Wasm utilized a separate *control stack* residing in the Wasm VM owned managed memory to store function return addresses, by leveraging the key idea of shadow stacks [18] [19] [20] [21]. Unfortunately, while Wasm's separation of data and control stacks effectively protected return address from being compromised, overflows on the data stack may still corrupt data on data stack frames or heaps [10], leading to the compromise of the whole system. Worse yet, while Wasm compiler toolchain supports C/C++ programs well, vulnerabilities (*e.g.*, buffer flows) in C/C++ source programs may be propagated from sources to Wasm by the toolchain without being detected. Hence, developing an effective memory protection for Wasm is essential.

Stack canary has been proposed as an effective technique to protect function return address [22] [23] [24] [25] [26] [27]. By placing a canary just below the function return address on the calling stack, the protection can detect potential buffer overflows by sanity checking the canary right before a function returns. Furthermore, as the canary is often pseudo random numbers, it is difficult for an adversary to guess or forge its value. Due to its technical advantages, stack canary has been a standard protection in production compilers such as GCC or clang.

Unfortunately, while stack canary is a promising and general-purpose protection technology, such an effective protection for Wasm does not yet exist (to the best of our knowledge). Yet developing a canary-based protection for Wasm faces three technical challenges: **C1:** *lacking of representation*, which makes it difficult to represent or encode the protection directly in the Wasm program; **C2:** *canary semantics transparency*, which defeats the effectiveness of the canary-based approach on a VM execution scenarios such as Wasm VMs; and **C3:** *toolchain diversity*, which results in the lack of universality for any protection targeting a specific toolchain.

Our work. In this paper, to fill the gap, our goal is to propose the the *first* framework for effective Wasm memory protection, with the assistant of Wasm VMs. With this design goal, we present VMCANARY, the first memory protection infrastructure. We first designed *VM-Canary*, a special form of canary but with a Wasm VM-aware semantics. Next, we designed instrumentation for Wasm programs to instrument these VM-Canary in an automated manner. Finally, we designed a tailored Wasm VM, which can detect and prevent any potential buffer overflows timely and effectively, by checking the value of a VM-Canary during program execution.

To realize the whole process, we have addressed the three aforementioned challenges. **C1:** *lacking of representation*: to address the challenge **C1**, we have designed two novel Wasm instructions `canary.insert` and `canary.check` to encode the semantics of a canary, where the former places a canary onto the stack when entering a function, and the latter sanity checks the canary before exiting a function. Furthermore, to define the two instructions' semantics rigorously, we have defined their typing rules as well as operational semantics, following the specification of Wasm [2]. **C2:** *canary semantics transparency*: to address the challenge **C2**, we have built a tailored Wasm VM by porting and modifying an open-source Wasmtime [28] VM (its Go implementation) to add support for the newly added Wasm instructions by extending the type checker as well as the execution engine. Nevertheless, while we have showcased our VM-assisted approach using the Wasmtime VM, our approach is general and can be generalized to other Wasm VM as well, as our design does not depend on the architecture of any specific VM. **C3:** *toolchain diversity*, to address the challenge **C3**, we propose a binary instrumentation approach to instrument Wasm binaries via a compiler rewriting pass on abstract syntax trees. Our approach outperforms existing canary-based protections in that it neither requires the program sources nor depends on any compiler toolchains.

To validate our design, we have implemented a software prototype for VMCANARY and have conducted extensive experiments to evaluate it in terms of effectiveness, efficiency, and overhead. To conduct the evaluation, we first created a microbenchmark with ground truths, and a macro benchmark with 59 real-world CWEs. Experimental results demonstrated that VMCANARY is effective in achieving a success rates of 94.9% in protecting stack buffer overflow vulnerabilities. Furthermore, VMCANARY is efficient in inserting the canary for each instruction in less than 1.5 milliseconds. Finally, VMCANARY brings negligible overhead, by increasing file sizes by less than 3%, and execution time by less than 3.5%.

Contributions. To the best of our knowledge, VMCANARY is the *first* framework for protecting Wasm memory via VM-assisted approach. To summarize, this paper makes the following contributions:

- **Problem analysis.** We conducted the first investigation into why traditional canary technology may fail to protect Wasm programs.
- **Infrastructure design.** We presented the design of VMCANARY, the first framework to protect Wasm memory, by leveraging a canary-based approach with the assistant from Wasm VMs.
- **Prototype implementation.** We implemented a software prototype for VMCANARY to validate our design.
- **Extensive evaluation.** We conducted extensive experiments to evaluate VMCANARY in terms of the effectiveness, efficiency, and overhead on both microbenchmarks and real-world CWEs.

Outline. The rest of this paper is organized as follows. Section II introduces the background for this work. Section III presents the challenges and the threat model. Section IV and V introduce the design and implementation of VMCANARY, respectively. Section VI presents the experiments to evaluate VMCANARY. Section VIII discusses the related work, and Section IX concludes.

II. BACKGROUND

To be self-contained, int this section, we present the background knowledge for this work by introducing Wasm (§ II-A), and its virtual machines (§ II-B).

A. Wasm Overview

Wasm is a next-generation safe and portable abstract instruction set architecture, which was initially released in March 2017 for the Web domain. In 2019, with the standardization work of Wasm System Interface (WASI) [29] defining a secure official standard for system support, Wasm starts to evolve into a general purpose language showing promising potentials diverse domains.

Wasm was designed with the goals of efficiency, portability, and security. First, Wasm has a compact binary format leading to its high efficiency in both Web and non-Web domains. Second, Wasm has a platform-independent instruction set which is executed by underlying Wasm VMs, making its programs portable. Third, Wasm introduced diverse secure language designs (*e.g.*, strong type systems [2], rigorous operational semantics [3], software fault isolation [4], secure control flow [5], and linear memory [6]), to guarantee its security.

Due its technical advantages, Wasm has been widely deployed in diverse domains: in the Web domain, Wasm has gained widespread support from all major browsers [30] [31]; and in non-Web domains, Wasm is used in standalone Wasm runtimes [32] [33], serverless cloud computing [34] [35] [36] [37], edge computing [38] [39] [40], and server-side computing [41].

B. Wasm Virtual Machines

A Wasm Virtual Machine (Wasm VM), as a core component of Wasm ecosystem, provides the execution environment and runtime support for Wasm programs. It consists of three main components: 1) a runtime engine, which executes the instructions sequence in Wasm programs, consisting of an operand stack for expression calculation, a managed memory

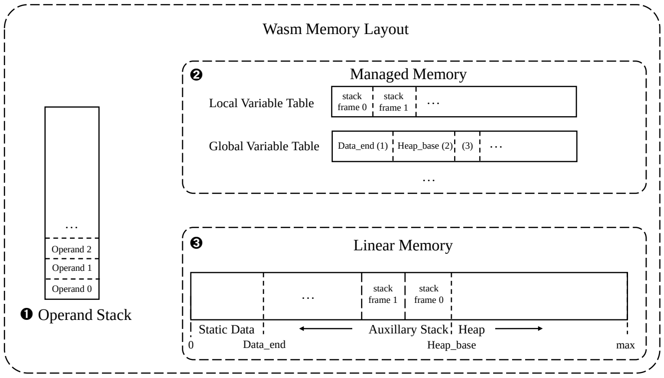

Fig. 1: The typical memory layout in a Wasm VM, consisting of three main components: the operand stack, the managed memory, and the linear memory.

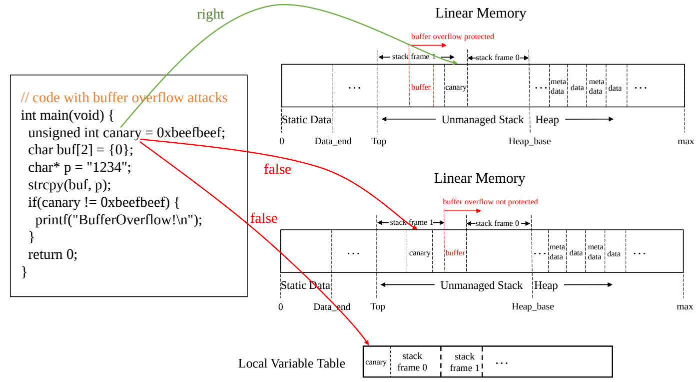

Fig. 2: A motivating sample Wasm program illustrating how buffer overflow occurs and why existing protections of canary are ineffective.

to store VM-managed data such as the control stack, and a linear memory to store user-managed data such as global or local variables; 2) supporting tools, which are a collection of utilities that assist in the development, debugging, and optimization of Wasm applications; and 3) WASI, which is a Wasm standard providing a consistent interface between Wasm programs and the operating system or the host environment, enabling a portable cross-platform execution.

As the focus of this paper is on the Wasm memory vulnerabilities and their mitigations, we thus present, in Fig. 1, a representative Wasm VM memory layout, consisting of three key components: 1) the operand stack (❶); 2) the managed memory (❷); and 3) the linear memory (❸). First, the operand stack is responsible for executing instructions, storing their operands and results. For example, when executing a binary addition Wasm instruction `i32.add`, the Wasm VM pops two operands `operand2` and `operand1` off the operand stack, then pushes the sum `operand1+operand2` onto the operand stack. As a result, the stack height decreases by one after executing this instruction. Second, the managed stack are managed directly by Wasm VM, storing the local variable table, and global variable table, among others. Specifically, the local variable table in the managed memory stores a control stack containing function returning addresses on the stack frames. Third, linear memory is a continuous storage space allocated specifically to be used by Wasm user programs, consisting of an auxiliary stack, and a heap. During program execution, the auxiliary stack stores aggregated local variables in a function (*e.g.*, buffers).

To summarize, each Wasm function utilized two separate stacks during execution: a control stack and a data stack, in which the former one resides in the managed memory owned by the underlying Wasm VM to store function return addresses, while the latter one resides in the linear memory to store aggregated local variables.

III. Security Challenges and Threat Model

In this section, we present motivations (§ III-A), security challenges and our solutions (§ III-B), and the threat model (§ III-C), for this work.

A. Motivations

Security is one of the most important design goals of Wasm. While Wasm's clear separation of the control stack and data stack guarantees memory safety by preventing the return address on the control stack from being corrupted by buffer overflows on the data stack, it fails to protect data on the data stack. Specifically, any overflows on the data stack may corrupt not only frames on the data stack but also the heap. Worse yet, traditional protection mechanisms such as canaries are ineffective to protect Wasm programs against such attacks.

To put the discussion in perspective, we present, in Fig. 2, a sample program illustrating how memory corruption might occur and why traditional protection mechanisms such as canaries might fail. To ease the understanding of the symptom and failure root causes, we use C program instead of Wasm to present the discussion without sacrificing generality.

The canary is an effective mechanism for memory protection with two phases: 1) a static; and 2) a dynamic phase. First, during the static phase, the compiler generates a canary-enabled binaries with extra instructions to process a canary. In Fig. 2, the compiler generated a `canary` with a random value `0xbeefbeef`. Second, during the dynamic phase, the canary-enabled binaries place the canary adjacent to the return address on the call stack when entering a function, and sanity check its value before exiting a function. In Fig. 2, the `canary` is checked right before exiting the function, and report an overflow if it manifested.

Unfortunately, while the canary is an effective protection against buffer overflows, it is ineffective to protect Wasm programs, for two key reasons: 1) *incorrect placement*; and 2) *wrong spacial ordering*. First, due to the special memory layout of Wasm, primitive data are placed on the control stack. Hence, if the compiler generates a canary of primitive integer types (*i.e.*, `i32`, or `i64`), the canary will be placed on the control stack, instead of on the data stack. As a result, data stack is not protected. Second, even if the canary is placed on the data stack, it may not be adjacent to the buffer it intends to protect. Worse yet, the Wasm VM is free to reorder the buffer and the canary, as it does not have the necessary knowledge of the canary semantics. In this case, the canary failed to detect the potential overflows, due to the wrong spacial ordering. Therefore, it is essential to propose an effective canary protection for Wasm, by tackling these problems.

B. Security Challenges and Our Solutions

Despite this security criticality and urgency [10] [11] [12], canary protection for Wasm has not been thoroughly studied (to the best of our knowledge). Yet, developing an effective framework for Wasm faces several technical challenges, which we present next along with our solutions.

C1: lacking of representation. Existing studies make use of an integer (32- or 64-bits), to represent a canary. While this representation is appropriate for execution of canary-enable native binaries, it does not apply to the the VM execution scenarios of Wasm programs. The key challenge lies in the fact that canaries have *no* representations in Wasm. Worse yet, existing integer representation of canaries may mislead the underlying Wasm VM, as canary does not carry any security information.

Solution. To address this challenge, our key idea is to treat the canary as *code*, instead of *data*. Specifically, we introduced two novel instructions into Wasm: `canary.insert` and `canary.check`, to represent canaries with straightforward and desired meanings: the former one places a canary onto the corresponding stack frame on the data stack when entering a function, while the latter one sanity checks the canary before exiting a function. One difficulty we must tackle is we must design both typing rules and operational semantics for these new instructions, following the the Wasm specification [2].

C2: canary semantics transparency. The underlying Wasm VM have understand the semantics of canaries, hence, it may perform arbitrary transformation on canaries, defeating canary's promise of effectiveness. Even with our aforementioned solution of treating canary as code, the Wasm VM does not understand the semantics of these newly introduced instructions.

Solution. To address this challenge, we have utilized an approach of tailored Wasm VMs. Specifically, to support the newly introduced canary-oriented Wasm instructions and to offer maximum flexibilities to VM implementation, we have designed and implemented a customized Wasm VM. To showcase our approach, we have ported and extended an open-source Wasm VM: Wasmtime [28] (its Go implementation), to add support for the newly added Wasm instructions by extending its type checker as well as the execution engine. Our VM-assisted approach has one more technical benefit: it offers maximum flexibility to VM implementation in supporting the canary, due to the fact canary generation are postponed to VM execution phase. For example, different VMs can select most appropriate pseudo random number generation algorithm to generate a canary, without relying on compiler support. Furthermore, while we have demonstrated our VM-assisted approach by using Wasmtime VM as a showcase, our approach is general and can be generalized to other Wasm VM as well, due to the fact that the typing rules and operational semantics of Wasm is rigorously specified hence is neutral to specific VM implementations [2].

C3: toolchain diversity. Current Wasm ecosystem has rich support for compiler toolchains (*e.g.*, Emscripten [42] for C/C++, Rustc/Wasm-bindgen [43] [44] for Rust, AssemblyScript [45] for TypeScript, and Tiny-Go/LLVM [46] for Golang [47]). Furthermore, as LLVM [48] has supported Wasm as one of its backends, in the future, more languages can target Wasm with the aid of LLVM. While these toolchains enable developers to leverage the technical advantages of Wasm by compiling into Wasm, the diversity of these toolchains and source languages brings a challenge: protection proposed for one language or toolchain may not be applicable to other ones. Worse yet, even if one can port the dedicated protection, the efforts in performing such a porting may be considerable.

Solution. To address this challenge, we proposed a *binary* instrumentation approach. Specifically, we designed and implemented a standalone binary instrumentation which takes as input raw Wasm binary file and rewrite it into a canary-enabled one by instrumenting the aforementioned canary instructions `canary.insert` and `canary.check`. Our static instrumentation approach has three key technical advantages: first, our approach is neutral to any specific compiler toolchain, as it operates directly on Wasm binaries. Second, our approach does not require that the compile toolchain has generated canaries or has placed them correctly. Third, our approach can be applied to any source languages, and does not need the source code.

C. Threat Model

The focus of this work is to present a comprehensive and effective framework for Wasm memory protection via a Wasm VM assisted approach. Therefore, we make the following assumptions in the threat model for this work.

We assume that the host environments for Wasm VM is safe, including but not limited to hardware, operating system, compiler, and linker. A great deal of studies have been conducted in these fields and a series of security protections and enhancements have been proposed with wide deployments [49] [50] [51]. Furthermore, compiler-level and OS-level security studies are independent and orthogonal to the study in this paper. In the meanwhile, our work will also benefit studies in these fields.

We assume that Wasm virtual machine is secure and trustworthy. We assume that the design and implementation of Wasm VM adhere to best security practices by properly isolating and executing Wasm modules. While the Wasm VM itself may have security vulnerabilities and attack vectors [52], our work is orthogonal to that VM security research direction, and our research also contributes to that field.

We assume that both source code of high-level languages to generate Wasm binaries and compiler toolchains are unreliable and thus vulnerable. On the one hand, the bugs in source code may be introduced by the insufficient range checking, which

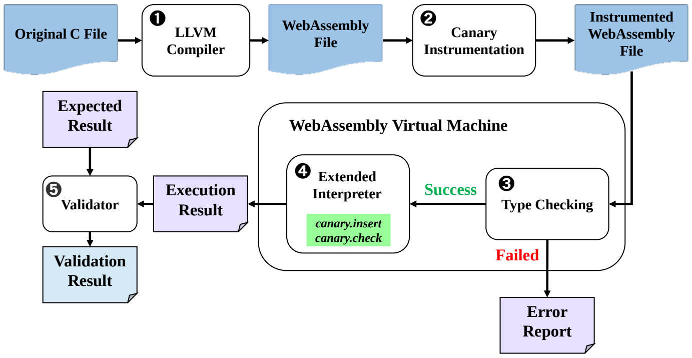

Fig. 3: The architecture of VMCanary.

are further propagated to the resulting Wasm binaries [10]. On the other hand, Wasm compiler toolchains are large and complex software, hence compiler bugs are inevitable, which may introduce vulnerabilities to safe source programs.

IV. VMCanary Design

In this section, we present the design of VMCanary, by first introducing the design goals (§ IV-A), its architecture (§ IV-B). Next, we present the design of each component (§ IV-C to § IV-G) of VMCanary, respectively.

A. Design Goals

We have three goals guiding the design of VMCanary: 1) completeness; 2) low overhead; and 3) full automation. First, VMCanary should provide complete protection for Wasm data stack in the linear memory without relying on other protection technologies; otherwise, there is no guarantee that the Wasm binary we are trying to protect will be free of buffer overflow vulnerabilities. Second, VMCanary should generate the ideal security-enhanced code, with minimal additional overhead and without changing the functionalities of the original programs. Third, VMCanary should be fully automated to enforce the protections, while manual interventions should only be required to investigate potential failures or to perform root cause analysis.

B. Architecture

With these design goals, we present, in Fig. 3, the architecture of VMCanary. Specifically, VMCanary consists of several key components: 1) the canary-oriented instruction design, in which we present the syntax, operational semantics, and typing rules for the newly introduced canary instructions `canary.insert` and `canary.check`, which will guide the underlying Wasm VM; 2) a binary instrumentation (❷), which analyzes the target Wasm binaries and instruments canary instructions into the program by placing them into appropriate positions; 3) a type checker (❸), which type checks the instrumented Wasm program to enforce the typing rules; 4) an execution engine (❹), which executes the instrumented Wasm program to implement canaries; and 5) an automated validator (❺), which validates the effectiveness of VMCanary leveraging differential testings and guarantees the normal functionalities by utilizing regression testings.

Next, we present the design of each component in detail, respectively.

C. Canary-oriented Instruction Design

To address the limitations of traditional canary protection when applied to Wasm, we have designed and extended two canary-oriented instructions to provide secure and effective canary protection on Wasm, *i.e.*, `canary.insert` and `canary.check`. We provide detailed syntax, operational semantics, and type rules for these two instructions.

First, like other instruction that conform to Wasm Standard Specification, `canary.insert` and `canary.check` are bytecode instructions, which mean that their opcode length is one byte. We used unused opcode values from the Wasm Standard Specification for them as their respective opcodes. Furthermore, these two instructions do not have any static operands.

Second, we define the operational semantics of `canary.insert` and `canary.check`. Fig. 4 provides the operational semantics of these two instructions. For `canary.insert`, it generates an 8-byte random number as the canary and inserts it into memory based on the position of the top of the data stack. The pointer to the top of the stack is also adjusted to ensure that the addresses of other memory operations in the code are appropriately adjusted for correctness. We also record the addresses and values of the inserted canaries for each stack frame, which will be used for subsequent verification operations. For `canary.check`, it is executed as the last instruction before `return` instruction. It retrieves the value of the canary from memory and compares it with the saved value to determine whether a buffer overflow occurred. For other instructions, we do not make any modifications, ensuring that the semantics of the Wasm program remain unchanged after the security enhancements.

Finally, the typing rules for `canary.insert` and `canary.check` is easy. the $canary$ value generated by `canary.insert` is has an type of $i64$, thus we just ensure the $canary$ placed at data stack is $i64$ type.

D. Binary Instrumentation

The binary instrumentation takes a binary Wasm file as input, and outputs an instrumented binary file by inserting canary instructions to appropriate positions. After binary instrumentation, the binary Wasm file is protected with effective canary protection, making the Wasm file free of buffer overflow on data stack.

This module works in two steps: first, it traverse each function in the binary Wasm file and adjusts the structured control flow of each function, ensuring that each function has a unique function exit point. Second, it inserts `canary.insert` before the first instruction of each function body, and `canary.check` before the $return$ instruction of the function.

$$[\text{canary.insert}]\frac{\Sigma(canary) = v1 \qquad \Sigma(fn) = f \qquad \Sigma(SP) = v2}{\Sigma, \Gamma, R \vdash \Sigma(SP) = v2 - 8 \qquad M(SP) = v1 \qquad R(f) = \{v2, v1\}}$$

$$[\text{canary.check}]\frac{R(f) = \{v2, v1\} \qquad Sigma(v2) = v}{\Sigma, \Gamma, R \vdash isequal(v1, v)}$$

Fig. 4: Operational semantics of `canary.insert` and `canary.check`.

While the workflow looks straightforward from a conceptual point of view, its design faces three unique challenges: 1) too many exit points; 2) too much overhead; and 3) functionality unchanged. First, due to the structured control flow of Wasm, a function can have many exit points. Therefore, it is very costly to insert `canary.check` before each exit point. As a key design goals of VMCanary, we need to ensure that the code remains nearly identical in terms of execution speed and has a minimal increase in file size after binary instrumentation. This extra overhead can be problematic when dealing with functions that have numerous exit points. To address this issue, we have designed Algorithm 1 to adjust the control flow of each function, ensuring it has a unique exit point. We wrap the entire function body with a block (line 2, 3, 12, 13). In addition, we traverse each instruction, keeping track of the current depth of the instruction (line 6 to 9). When encountering a `return` instruction, we replace it with a $br(depth - 1)$ instruction to break out of the outermost block (line 11). Such adjustment of the control flow effectively minimizes the overhead introduced by binary instrumentation. Second, we aim to minimize the time required for binary instrumentation of the Wasm file. We expect the overall enhancement process to be completed quickly. Algorithm 2 is designed to accomplish this task in a short amount of time. The algorithm only requires a sequential traversal of each function. It first adjusts the control flow of the functions and then inserts the `canary.insert` and `canary.check` instructions. Therefore, our algorithm demonstrates excellent efficiency. Third, we need to ensure that the functional and security characteristics of our operations on Wasm files remain unchanged after binary instrumentation. Therefore, we cannot destroy other data stored in memory after inserting canary protection, *i.e.*, we need to rearrange the storage addresses of the other data on data stack accordingly according to the inserted data. Meanwhile, for memory load instructions, we also need to adjust the load address of these instructions so that the security-enhanced code can still load the correct data. In addition, in order not to affect the function of the code, we cannot change the logic of the code, but only enhance the security of the entire Wasm file, and the original security features in the file also need to be preserved.

Algorithm 1: Control Flow Adjustment Algorithm.

Input: $\mathbb{I}$: instructions in a WebAssembly function
Output: $\mathbb{I}$: the control flow adjusted WebAssembly function

```
1  Function ControlFlowAdjust(𝕀):
2      StartInstr ← block;
3      append (StartInstr, 𝕀);
4      depth ← 0;
5      for i in 𝕀 do
6          if instr == block or instr == loop or instr
             ==if then
7              depth ← depth + 1
8          else if instr == end then
9              depth ← depth − 1
10     if instr == return then
11         instr ← br(depth-1)
12     EndInstr ← end;
13     append (𝕀, EndInstr);
14     return 𝕀;
```

Algorithm 2: Canary Instrumentation Algorithm.

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

```
Function CanaryInstrumentation(𝕀):
    for code in 𝕀 do
        adjustControlFlow (code);
        ProtectInstr ← canary.insert;
        append (ProtectInstr, code);
        VerifyInstr ← canary.check;
        append (code, VerifyInstr);
    return 𝕀;
```

E. Type Checker

The task of the type checker is to verify the correctness of the types of instructions and operands in a Wasm module. This includes checks for type consistency, stack height verification, validation of function signatures, and validation of table indices and function calls. The type checker aims to detect type errors in advance and improve the safety and reliability of the module. If the type checker determines that the types are correct, the corresponding Wasm module can be executed by the execution engine. However, if there are type errors, the type checker will provide relevant error or warning messages and terminate the subsequent execution flow.

The type checker achieves its functionality by simulating the execution process of a Wasm module. It does not actually execute the corresponding Wasm module but instead examines the types of operands and results for each instruction. It gathers information about the types of operands and results for each instruction and checks whether they are correct according to the defined types. By analyzing the types of instructions and their operands, the type checker can determine if the types are consistent and detect any type errors or inconsistencies. For the newly introduced `canary.insert` and `canary.check` instructions, we only need to check that the top operand on the operand stack is an 8-byte data. For other instructions, we do not alter their type-checking and operational semantics. After type checking, the check results will be generated. If obtaining the correct result, we pass the Wasm file to the subsequent module for subsequent interpretation, operation and testing. Otherwise, we give accurate error information for debugging and terminate the subsequent process.

F. Execution Engine

The execution engine is one of the core modules of the Wasm virtual machine. It takes a type-checked Wasm binary file as input and performs decoding, instantiation, and execution phases to interpret and execute each instruction in the Wasm file, resulting in the execution result. The canary protection is provided in a virtual machine-assisted approach, which involves extending the interpretation logic of the Wasm interpreter to support two additional instructions.

We present the detailed explanation of how the two newly introduced instructions are executed. For `canary.insert`, it generates an 8-byte random number as the canary and inserts it into memory based on the position of the top of the unmanaged stack. The pointer to the top of the stack is also adjusted to ensure that the addresses of other memory operations in the code are appropriately adjusted for correctness. We also record the addresses and values of the inserted canaries for each stack frame, which will be used for subsequent verification operations. For `canary.check`, it is executed as the last instruction before `return` instruction. It retrieves the value of the canary from memory and compares it with the saved value to determine whether a buffer overflow occurred. For other instructions, we do not make any modifications, ensuring that the semantics of the Wasm program remain unchanged after the security enhancements. The interpretation logic of the two newly introduced instructions fully adheres to their operational semantics. For `canary.insert`, we accurately insert canary at the correct position in the non-managed stack, avoiding the impact of canary semantic transparency on the location of the canary in Wasm memory. For `canary.check`, we always check whether the value of the canary has been modified before a function exits, providing real and effective canary protection for each function.

G. Automated Validator

The automated validator implements a complete process of instruction instrumentation and execution validation for protection. This module takes the original Wasm file and the expected execution result as input. It uses the binary instrumentation to insert protection instructions into the Wasm file, performs type checking, executes the instrumented file using the execution engine, and generates the execution result for differential testing. Additionally, the module performs regression testing to ensure that functionality is not affected by the inserted protection instructions.

For the differential testing, this module requires the expected results as input in order to compare the execution results after inserting the protection instructions with the expected results. If they match, it indicates that the inserted protection instructions have successfully provided the intended protection. In case of mismatched results, a detailed error report is generated to assist developers in examining the differences between the execution results and the expected results, helping them identify the cause of the error.

For regression testing, we need to verify that the return values of each function remain the same before and after inserting the protection instructions. To implement regression testing, we record the return values of each function before and after binary instrumentation for each test case. If the return values of all functions, except those affected by buffer overflows, remain the same, it indicates that our binary instrumentation ensures the preservation of the program's functional logic.

V. Implementation

To validate our design, we have implemented a software prototype for VMCanary. We have implemented the canary instrumentation using 831 lines of C code, leveraging the LLVM compiler. Specifically, we have implemented, in C language, the canary instrument algorithm in binary instrumentation for inserting protection instructions and control flow adjustment algorithm for adjusting Wasm structured control flow. To implement the newly introduced Wasm instructions, we have ported and extended a WasmVM Wasmtime-go [28], which is implemented in Golang. Specifically, we have expanded its type checker and interpretation engine to add the support for the `canary.insert` and `canary.check` instructions. Finally, we implemented the automated validator using bash and Python scripts, by leveraging test cases distributed with the benchmark.

VI. Evaluation

In this section, we conduct experiments to evaluate VMCanary. We first introduce the research questions guiding the evaluation (§ VI-A). Then we introduce the experimental setup for the evaluation (§ VI-B). Next, we introduce the datasets used for the evaluation (§ VI-C). Finally, we evaluate the effectiveness, efficiency, and overhead of VMCanary, and compare it with the existing frameworks for Wasm (§ VI-D to § VI-G).

A. Research Questions

By presenting the experimental results, we mainly investigate the following research questions:

RQ1: Effectiveness. Since VMCanary is designed to provide security protection on the Wasm unmanaged stack, is VMCanary effective in providing canary security protection?

RQ2: Efficiency. As a tool that provides security enhancements on Wasm files, can VMCANARY efficiently insert security protections into specified locations without consuming too much time?

RQ3: Overhead. As a tool to provide security protection, the security protection code inserted by VMCANARY will inevitably increase the size of the Wasm file and affect the execution efficiency. Therefore, is VMCANARY guaranteed to bring low overhead when inserting security?

RQ4: Compare with existing framwork. Can VMCANARY outperform existing bug detection framwork?

B. 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.

C. Datasets

We used two datasets to conduct the evaluation: 1) micro-benchmarks; and 2) real-world CWEs, containing a total of 59 vulnerable programs.

Micro-benchmark. We manually constructed a micrco-benchmark consisting of 10 test cases with diverse points the buffer overflow vulnerability may occur (as presented by the second column of Table I), including calling $strcpy$, calling $strcat$, variadic parameters, and so on. These test cases are collected from two sources: 1) public CVEs; and 2) existing literature on Wasm security studies. To better reflect the significance of buffer overflow vulnerability and to simplify the validation, we have rewritten some of the original buggy code by removing irrelevant code.

Real-world CWEs. CWE [53] is a set of vulnerabilities in C programs with a total of 59 programs which contain stack buffer overflow vulnerability. Conducting our VMCANARY on CWE is an effective way to validate the effectiveness of our framwork. In order to use CWE as our real-world dataset, we precompiled each program into the Wasm binary format and ensured that they can run correctly on the Wasm virtual machine and import the required runtime modules.

D. RQ1: Effectiveness

To evaluate the effectiveness of VMCANARY, we first apply VMCANARY to micro-benchmarks. The last two columns Table I presents the experimental result, which illustrates that VMCANARY is effective in protecting buffer overflow on unmanaged stack. The experimental results demonstrate that VMCANARY effectively mitigates the potential buffer overflow attacks in these cases, thus enhancing the security of Wasm programs.

In order to study the effectiveness of VMCANARY when on real-world programs, we apply VMCanary to the CWE. For the experimental results of this test set that Table II shows, out of the 59 CWEs, VMCANARY successfully protected 56 of them, while 3 were not adequately protected. Therefore, the effective rate of VMCANARY is 94.9%. Furthermore, we investigate the root cause of the 3 failed cases. After manually inspecting these Wasm binaries, we discovered that the reason for the protection failure was that the stack buffer overflows in these cases did not surpass the current stack frame. As a result, the canary check failed to detect them.

Summary: The canary protection instructions instrumented by our VMCANARY effectively enhanced the security of Wasm binaries, protecting Wasm code from source-level stack buffer overflow vulnerabilities.

E. RQ2: Efficiency

To answer RQ2, we apply VMCANARY to both micro-benchmark and real-world CWE. When calculating the time required for security enhancements, we recorded the time when the security enhancements were started and the time when the security enhancements were completed, at the same time we record the total number of canary security protections inserted. We then calculated the average time required to insert each canary security.

Table I presents the experimental results. The 5th column lists the time spent on security enhancement for each test case and the average time of each instruction we instrumented.

The results give interesting findings and insights. First, VMCANARY can efficiently insert security-enhancing protections, and the average time spent on inserting each instruction is less than 1.5ms. Second, different test cases have some differences in the average time it takes to insert security protection. After research, we believe that the reason for this phenomenon is that the insertion of security enhancement needs to traverse each function in the program, which causes the execution time of the algorithm to be affected by the number of instructions contained in the function, so this difference occurs.

Summary: VMCANARY can efficiently insert the canary into the head of the Wasm unmanaged stack frame to provide protection against unmanaged stack buffer overflows. The average time for the framework to insert canary instructions for each function is less than 1.5 milliseconds, so the security enhancement of Wasm can be completed in a very short time.

F. RQ3: Overhead

Table I presents the overhead that security enhancements impose on Wasm binaries. The overhead after VMCANARY security enhancement includes: 1) increase in the number of Wasm instruction; and 2) execution time when Wasm binary interprets. We compared the change in the LoC(line of code) of the Wasm binary before and after the security enhancement. The results show that the security-enhanced file size increased by 3% on average and remained at a very low level. In addition, we evaluated the increase of execution overhead after security enhancement, we run them 10 times respectively, and take the average value of the running. The experimental data shows that the execution overhead we introduced is below 3.5%, which is very low. Therefore, VMCANARY security enhancements will not only bring a small file size increase and runtime increase to the program.

TABLE I: Experimental results on micro-benchmarks.

Test Case	Vulnerability Type	LoC BI	LoC AI	Instrumentation Time (s) / per instr (ms)	EXE Overhead	EXE Res BI / EXE Res AI	VMCanary
1	BF_strcpy	799	823	0.023 / 0.971	3.1%	Overflow / NotOverflow	✓
2	BF_sprintf	1387	1437	0.071 / 1.307	3.4%	Overflow / NotOverflow	✓
3	BF_strcat	845	887	0.029 / 0.684	2.7%	Overflow / NotOverflow	✓
4	BF_fgets	2136	2179	0.069 / 1.152	1.9%	Overflow / NotOverflow	✓
5	BF_scanf	1841	1910	0.063 / 0.832	3.2%	Overflow / NotOverflow	✓
6	BF_fread	3562	3697	0.135 / 0.924	2.9%	Overflow / NotOverflow	✓
7	BF_funcall	4267	4335	0.981 / 1.088	2.4%	Overflow / NotOverflow	✓
8	BF_pointer	1982	2037	0.046 / 0.753	3.0%	Overflow / NotOverflow	✓
9	BF_localvar	2687	2803	0.142 / 1.251	2.5%	Overflow / NotOverflow	✓
10	BF_variadic	5426	5574	0.994 / 0.635	1.6%	Overflow / NotOverflow	✓

TABLE II: Experimental results on real-world-benchmarks.

Dataset	Total	VMCanary Success / Miss	Fuzzm Success / Miss	VMCanary Rate	Fuzzm Rate
CWE [53]	59	56 / 3	52 / 7	94.9%	88.1%

Summary: The overhead introduced by VMCanary is very small, including the increase in code size and the increase in code execution time. In terms of code size, the average increase per Wasm file is 3%. And in terms of code execution time, compared to before security enhancement, each Wasm file showed less than 3.5% performance loss during execution, which is negligible.

G. RQ4: Compare with existing framework

We compared VMCanary with the existing binary-level canary insertion tool for Wasm, Fuzzm [54], to evaluate their effectiveness on real-world CWEs. The experimental results, presented in Table II, demonstrate that VMCanary provides more effective protection against unmanaged stack buffer overflows compared to Fuzzm. VMCanary achieved an effectiveness rate of 94.9%, while Fuzzm exhibited only 88.1% effectiveness. Through analysis of the 7 cases where Fuzzm failed, we found that an additional 4 cases failed due to Fuzzm's approach of using existing instructions for canary insertion. However, this protection method may fail due to variations in memory layout implementation across Wasm virtual machines, thus not effectively safeguarding Wasm programs with stack buffer overflow vulnerabilities.

Summary: VMCanary offers effective protection against unmanaged stack buffer overflows in Wasm. It outperforms existing tools by providing a more efficient and reliable defense mechanism.

VII. Discussion

In this section, we discuss some possible enhancements to this work, along with directions for future work. It should be noted that our work represents the first step towards improving the security of Wasm code through virtual machine-assisted techniques.

More accurate instrumentation position. Although our framework can effectively protect the security of the Wasm unmanaged stack by inserting canary, we use a conservative way to insert canary to achieve the protection effect. In this way, canary protection instructions are inserted into each function, which may cause some unnecessary instrumentation operations. Therefore, we can optimize the algorithm to achieve more accurate instrumentation positions and reduce the number of inserted instructions. In addition, we can also combine common static analysis frameworks [55] [56] or some algorithms to detect heap buffer overflows [57], to assist in identifying code segments that require protection, thereby generating higher-quality Wasm code. Optimized security enhancements result in minimal increase in the size of Wasm code and improved runtime efficiency.

Heap overflow. VMCanary enhances the security of Wasm by providing canary protection for the unmanaged stack in Wasm's linear memory, effectively mitigating buffer overflow attacks and avoiding stack smashing attacks. However, VMCanary does not provide effective protection for data on the unmanaged heap in linear memory. Data on the heap is vulnerable to attacks [10] [58], which can lead to serious security issues. Therefore, addressing security enhancements for the heap is an important future direction to further enhance the security of Wasm.

Hook implementation. Since VMCanary modifies the virtual machine to interpret the $canary.insert$ and $canary.check$ instructions, our approach may not be effective for non-open-source Wasm virtual machines. Therefore, we consider to employ an dynamic analysis approach, leveraging existing Wasm dynamic analysis framework [59], to insert hook functions at the entry and exit points of functions. This will allow us to achieve memory protection for Wasm in a non-instrusive manner that does not require modifications to Wasm virtual machine.

VIII. Related Work

In recent years, there has been a significant amount of research on Wasm security and its security enhancements. However, our work in this paper stands for a novel contribution to these fields.

Wasm security study. There has been a lot of empirical research on Wasm security [12] [13] [14] [60] [52]. Hilbig et al. [12] conducted research on 8,461 Wasm binary files and found that about 80% of the binary files were compiled through the LLVM toolchain. However, in the process of compiling and generating Wasm, there is no support for protection mechanisms such as stack canary. Quentin et al. [13] found discrepancies in the execution results of 1,088 programs in 4,469 buffer vulnerable C programs on x86 and Wasm. In addition, Quentin et al. [14] selected 17,802 C programs from the Juliet suite, compared the running results of their corresponding x86 programs and Wasm programs, and found that 4,911 C programs had different running results. One of the important reasons for this difference is the lack of security protection mechanisms such as stack canary in Wasm's linear memory. Musch et al. [60] studied the prevalence and code characteristics and assessment of Wasm modules. Wang et al. [52] studied Wasm runtime bugs in terms of root cause, symptoms, and so on. Therefore, our VMCANARY represents the first effective work to enhance the security of Wasm linear memory via a virtual machine-assisted approach.

Buffer overflow protection. Buffer overflow is a common and highly dangerous vulnerability that exists widely in various operating systems and software applications. Extensive research has been conducted to mitigate or protect against buffer overflow attacks, and many effective techniques have been developed primarily for the native x86 platform [61] [62] [63] [64] [65]. These techniques have proven effective in preventing buffer overflow attacks at the native architecture. StackArmor [61] departs from the traditional stack organization structure and relies on mechanisms such as randomization, isolation, and zero initialization to enhance the security of the stack in the x86 architecture. By introducing these measures, StackArmor aims to mitigate the risks associated with stack-based vulnerabilities and improve the overall security posture of software running on the x86 platform. SafeStack [62] provides an automated approach to detect and mitigate stack buffer overflow vulnerabilities by manipulating memory accesses. It achieves this by relocating vulnerable buffers to a protected memory area, thus offering protection against buffer overflow attacks. By leveraging memory access patterns and instrumentation, SafeStack aims to automatically identify and prevent potential buffer overflow exploits, enhancing the security of software applications. Slowinska et al. [63] introduced BodyArmor, which employs binary instrumentation to manage read and write operations on pointers within executable files. By instrumenting the binary code, BodyArmor tracks and enforces security policies related to pointer operations, aiming to prevent memory corruption vulnerabilities such as use-after-free and buffer overflow. Huang et al. [64] improved security through a more comprehensive and accurate safety analysis. Duck et al. [65] provided stack bound protection with low-fat pointers. Our VMCANARY shares a similar technical approach with these tools in terms of utilizing binary instrumentation for security protection. However, the key difference lies in the fact that Wasm lacks fine-grained protection for linear memory, making it unable to leverage traditional stack non-executable techniques. Therefore, we adopt the approach of inserting canaries as a means of providing protection in the context of Wasm.

Wasm security enhancement. As an important part of security research, security enhancement can enhance the defense function of software systems by introducing specific security mechanisms. Existing research [10] [11] has indeed shown that while Wasm incorporates numerous security protection details in its design, it still has several security vulnerabilities. There has been a lot of research on Wasm security enhancements [66] [67] [54]. Arteaga et al. [66] proposed the CROW system, which statically deforms the code through code diversification technology, so as to achieve the uniqueness goal of Wasm binary code distributed based on the same source code. But this work is based on source code implementation and cannot be directly used in Wasm binary code, and cannot be enhanced at runtime. Disselkoen et al. [67] proposed MS-WASM. By extending the standard Wasm memory model, the concept of segment memory was introduced, and combined with ARM's MTE and other hardware mechanisms, security checks were performed on the validity of pointers. However, the security enhancements made in this work can only be effective on the latest platforms with specific hardware protection mechanisms, and are not universal. Daniel et al. [54] proposed Fuzzm, which protects the heap and stack insertion canaries in the Wasm linear memory area through the rewriting technology based on Wasm binary code, so as to achieve the protection of Wasm memory. However, the protection instructions inserted by this work may be affected by the rearrangement of Wasm runtime memory layout, and the protection of the heap area is only achieved by inserting corresponding instructions for functions such as $malloc$, $calloc$, and $realloc$. Therefore, the protection of Wasm memory is not universal and not thorough enough. Instead, we implement protection and detection operations before memory allocation and after memory usage by extending two instructions dedicated to protection and detection and inserting them into the corresponding positions of the Wasm binary. Therefore, such a protective effect will not be affected by memory layout rearrangement, and has universality.

IX. CONCLUSION

In this work, we present VMCANARY, a virtual machine-assisted technology stack canary protection framework for Wasm. We implemented a prototype system of VMCANARY and conducted experiments to compare it with the existing framwork. The evaluation results demonstrate that VMCANARY can provide more effective protection against buffer overflow attacks on the Wasm unmanaged stack compared to the existing framework. It enhances the security of Wasm programs while introducing negligible overhead. This work represents an important step in improving the security of Wasm unmanaged stacks through virtual machine assistance, reducing the impact of Wasm memory vulnerabilities in practical applications.

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 2023-04-24)," https://webassembly.github.io/spec/core/exec/index.html.

[6] "Structure — WebAssembly 2.0 (Draft 2023-04-24)," 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] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *International Journal of Web and Grid Services*, vol. 14, no. 4, pp. 352–375, Jan. 2018.

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

[11] 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.

[12] 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.

[13] 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.

[14] "Security risks of porting C programs to webassembly — Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing," https://dl.acm.org/doi/abs/10.1145/3477314.3507308.

[15] K.-S. Lhee and S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities," *Software: Practice and Experience*, vol. 33, no. 5, pp. 423–460, Apr. 2003.

[16] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2, Jan. 2000, pp. 119–129 vol.2.

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

[18] 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.

[19] 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.

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

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

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

[23] M. Prasad and T.-c. Chiueh, "A Binary Rewriting Defense against Stack based Buffer Overflow Attacks."

[24] "12th USENIX Security Symposium — Technical Paper," https://www.usenix.org/legacy/event/sec03/tech/full_papers/cowan/cowan_html/.

[25] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, and B. Mao, "To Detect Stack Buffer Overflow with Polymorphic Canaries," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2018, pp. 243–254.

[26] H. Marco-Gisbert and I. Ripoll, "Preventing Brute Force Attacks Against Stack Canary Protection on Networking Servers," in *2013 IEEE 12th International Symposium on Network Computing and Applications*, Aug. 2013, pp. 243–250.

[27] T. Petsios, V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "DynaGuard: Armoring Canary-based Protections against Brute-force Attacks," in *Proceedings of the 31st Annual Computer Security Applications Conference*, ser. ACSAC '15. New York, NY, USA: Association for Computing Machinery, Dec. 2015, pp. 351–360.

[28] "Bytecodealliance/wasmtime-go: Go WebAssembly runtime powered by Wasmtime," https://github.com/bytecodealliance/wasmtime-go.

[29] "WASI —," https://wasi.dev/.

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

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

[32] "WebAssembly Micro Runtime," Bytecode Alliance, May 2023.

[33] "Wasmtime," https://wasmtime.dev/.

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

[43] "Rust-lang/rust: Empowering everyone to build reliable and efficient software." https://github.com/rust-lang/rust.

[44] "Rustwasm/wasm-bindgen: Facilitating high-level interactions between Wasm modules and JavaScript," https://github.com/rustwasm/wasm-bindgen.

[45] "AssemblyScript/assemblyscript: A TypeScript-like language for WebAssembly." https://github.com/AssemblyScript/assemblyscript.

[46] "Tinygo-org/tinygo: Go compiler for small places. Microcontrollers, WebAssembly (WASM/WASI), and command-line tools. Based on LLVM." https://github.com/tinygo-org/tinygo.

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

[48] J. Bergbom, "Memory safety: Old vulnerabilities become new with WebAssembly."

[49] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee, "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, Oct. 2015, pp. 280–291.

[50] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and Deployable Continuous Code Re-Randomization."

[51] C. Zou, Y. Gao, and J. Xue, "Practical Software-Based Shadow Stacks on x86-64," *ACM Transactions on Architecture and Code Optimization*, vol. 19, no. 4, pp. 1–26, Dec. 2022.

[52] 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.

[53] "CWE - CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer (4.11)," https://cwe.mitre.org/data/definitions/119.html.

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

[55] F. Breitfelder, T. Roth, L. Baumgärtner, and M. Mezini, "WasmA: A Static WebAssembly Analysis Framework for Everyone," in *2023*

IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), Mar. 2023, pp. 753–757.

[56] Q. Stiévenart and C. D. Roover, "Compositional Information Flow Analysis for WebAssembly Programs," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2020, pp. 13–24.

[57] O. Ruwase and M. S. Lam, "A Practical Dynamic Buffer Overflow Detector."

[58] Q. Zeng, M. Zhao, and P. Liu, "HeapTherapy: An Efficient End-to-End Solution against Heap Buffer Overflows," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Rio de Janeiro, Brazil: IEEE, Jun. 2015, pp. 485–496.

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

[60] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, R. Perdisci, C. Maurice, G. Giacinto, and M. Almgren, Eds. Cham: Springer International Publishing, 2019, vol. 11543, pp. 23–42.

[61] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, *StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries*, Feb. 2015.

[62] G. Chen, H. Jin, D. Zou, B. B. Zhou, Z. Liang, W. Zheng, and X. Shi, "SafeStack: Automatically Patching Stack-Based Buffer Overflow Vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, no. 6, pp. 368–379, Nov. 2013.

[63] T. Stancescu and H. Bos, "Vrije Universiteit Amsterdam."

[64] K. Huang, Y. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "The Taming of the Stack: Isolating Stack Data from Memory Errors," in *Proceedings 2022 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2022.

[65] G. J. Duck, R. H. C. Yap, and L. Cavallaro, "Stack Bounds Protection with Low Fat Pointers," in *Proceedings 2017 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2017.

[66] 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.

[67] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, "Position Paper: Progressive Memory Safety for WebAssembly," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. Phoenix AZ USA: ACM, Jun. 2019, pp. 1–8.