WASMSEPA: Effectively Protecting WebAssembly Through Privilege Separation

Zhuochen Jiang, and 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
*Corresponding author

Abstract—WebAssembly (Wasm) is an emerging binary instruction set architecture designed for secure binary program execution and is increasingly deployed across various security-critical domains including edge computing and smart contracts. However, despite its security-oriented design, Wasm remains vulnerable due to its support for compiling functions and APIs from unsafe languages and the lack of effective protection mechanisms during compilation. These issues undermine Wasm's security guarantees.

In this paper, we present WASMSEPA, the first approach for effectively securing Wasm through privilege separation, by isolating Wasm binaries from functions written in unsafe languages. Our key insight is that, since Wasm incorporates several security features, we should execute Wasm binaries and libraries from unsafe languages in separate processes to enforce strong privilege separation. Specifically, we first identify potential unsafe function calls during Wasm binary generation. We then wrap these unsafe functions by utilizing remote procedure calls (RPC). We finally enable interprocess communication between safe Wasm binaries and unsafe functions via RPC. We implement a software prototype of WASMSEPA and conduct extensive experiments to evaluate its effectiveness, usefulness, and overhead using micro and real-world benchmarks. Experimental results demonstrate that WASMSEPA effectively protects Wasm linear memory from unsafe code while incurring an average code size increase of 48% and an execution time overhead of 7.63×.

Keywords–WebAssembly Security; Remote Procedure Call; Isolation

1. INTRODUCTION

WebAssembly [1] (Wasm) is an emerging binary instruction set architecture and code distribution format [2] designed with security as a core principle. It incorporates a range of secure language features, including a strong type system [2], mathematically rigorous operational semantics [3], software fault isolation [4], secure control flows [5], and a linear memory [6], to guarantee security. Owing to these properties, Wasm is increasingly adopted in security-critical domains such as edge computing [7] and smart contracts [8]. It is expected to become one of the most significant instruction set architectures for code execution and distribution in the coming decade.

Despite its security-oriented design, Wasm programs remain vulnerable and exploitable, primarily because current Wasm compilers support compiling unsafe languages such as C/C++ into Wasm. Consequently, vulnerabilities in source programs may propagate to the compiled Wasm binaries [9] [10] [11] [12] [13]. Although Wasm introduces a novel security feature—*linear memory* [14]—to isolate function return addresses and data buffers, it does not fully prevent memory corruption. Data stored in linear memory remains susceptible to overwrites caused by vulnerabilities such as buffer overflows inherited from unsafe source languages [9]. More worryingly, even Wasm compilers for memory-safe languages (e.g., TinyGo [15] for Go and Rustc [16] for Rust) may generate potential vulnerable libraries that are invoked by Wasm binaries through foreign function interfaces (FFI). These libraries, not protected by the Wasm security mechanisms, can introduce memory vulnerabilities that compromise the entire system, undermining the intended security guarantees [13]. Therefore, providing effective memory protection and strengthening the security of Wasm programs is both critical and urgent.

Recognizing the critical importance and urgency of Wasm security, researchers have conducted a broad range of studies on this topic [10] [11] [12] [13] [17] [18] [19] [20] [21] [22] [23]. While these studies provide valuable contributions, they do not fully address challenges of Wasm memory protection. First, existing studies overlook support for multilingual source languages. For example, Fuzzm [18] only supports C/C++ languages using the clang compiler [24], making it unclear how to extend its protection mechanisms to other source languages and their Wasm compilers (e.g., Rustc [16] for Rust). Even if such adaptations are feasible, they are often labor-intensive and cost-ineffective, due to the substantial size of compiler codebases (e.g., Clang [25] recently surpassed 6 million lines of code and continues to grow rapidly) and considerable engineering efforts required. Second, existing studies compromise generality by relying on hardware-specific security mechanisms. For example, PKUWA leverages Intel MPK [23] to mitigate memory vulnerabilities. This reliance on architecture-specific hardware limits its applicability to systems that lack similar hardware mechanisms. Finally, from a vulnerability coverage standpoint, existing studies lack scalability by only focusing on specific vulnerabilities. For example, metaSafer [22] focuses on heap metadata corruption by shadowing the entire linear memory to shadow memory, but fails to account for vulnerabilities arising from unsafe APIs. Consequently, Wasm developers struggle to deploy comprehensive protection for Wasm programs. As a result, many vulnerabilities remain undetected even after existing protections are applied.

In this paper, we present the first approach for effective

Wasm memory protection by leveraging privilege separation, to isolate Wasm binaries from unsafe libraries. Our key insight is that, since Wasm inherently provides a wide range of security features, yet various vulnerabilities still originate from unsafe source code, it is both feasible and cost-effective to separate the privileges of Wasm binaries and APIs in unsafe libraries when accessing linear memory. Based on this insight, we enforce privilege separation using a two-phase strategy that combines compile-time separation and runtime sanity checking. First, during Wasm binary generation, we identify potentially unsafe function calls and conservatively assume that any such calls might introduce security risks and thus should be separated, when source-level information is absent. We then partition the target Wasm program into a client component comprising unsafe libraries and a server component with safe code, by utilizing a compiler rewriting pass. During this partitioning, we instrument the target Wasm program with remote procedure calls (RPC) to replace the original direct procedural calls. Finally, at runtime, we employ inter-process communication (IPC) between Wasm binaries and unsafe functions, effectively isolating the privilege of unsafe APIs and preventing them from directly access the original linear memory.

In realizing our approach, we address three technical challenges. **C1:** existing software-based privilege separation techniques (e.g., SFI and sandboxing) often require substantial source code rewriting or modifications to the virtual machines, making them labor-intensive. One the other hand, hardware-based approaches (e.g., Intel MPK) lack portability due to their dependency on specific architecture [23]. To address this issue, we adopt a remote procedure call-based strategy that isolates unsafe and safe code segments, enforcing memory protection via inter-process communication. **C2:** the diversity of programming languages and toolchains poses challenges in designing a universal protection mechanism. To tackle this, we propose a binary-level solution that operates directly on Wasm binaries, avoiding reliance on specific languages and compilers. **C3:** unlike languages such as Rust, Wasm does not natively distinguish unsafe and safe code. To address this, we employ a conservative heuristic that treats all foreign function interface calls as unsafe when source-level information is unavailable.

We implement a prototype of our approach, called WASMSEPA, and conduct extensive experiments to evaluate its effectiveness, practicality, and overhead. Our evaluation includes a micro-benchmark comprising 16 vulnerable Wasm binaries, as well as a real-world benchmark featuring 25 CWEs and 10 real-world Rust/C projects. The evaluation results show that WASMSEPA effectively mitigates five types of memory vulnerabilities and outperforms existing solutions. Furthermore, WASMSEPA demonstrates practical utility by successfully securing 30 out of 35 (85.7%) Wasm programs. Finally, WASMSEPA introduces acceptable overhead, with an average code size increase of 48% and a runtime increase of 7.63×, which is consistent with prior studies [26].

In summary, this paper makes the following contributions:

- We propose the first approach to effectively protect Wasm memory, by leveraging a privilege separation method.
- We design and implement a software prototype WASMSEPA to validate our approach.
- We conduct extensive experiments to show that WASMSEPA 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 2 introduces the background. Section 3 outlines the motivations and the threat model. Section 4 and 5 presents our approach, and the evaluation results, respectively. Section 6 discusses limitations and future directions. Section 7 reviews related work, and Section 8 concludes.

2. BACKGROUND

To be self-contained, this section provides the necessary background of Wasm (§ 2.1) and privilege separation (§ 2.2).

2.1 Wasm

Wasm is an emerging secure and portable instruction set architecture, initially released in 2017 for Web. In 2019, the introduction of the Wasm System Interface (WASI) [27] marked its transition into an official Web standard, enabling its evolution into a general-purpose language deployed across various domains beyond the Web.

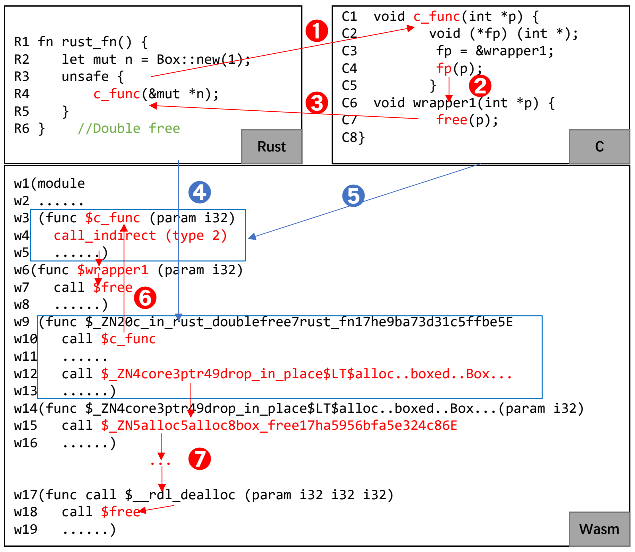
Wasm was designed with three key goals of security, efficiency, and portability. First, it incorporates multiple security features including strong typing [2], rigorous operational semantics [3], software fault isolation [4], secure control flow [5], and linear memory [6], to enhance program security. Second, Wasm VMs enable Wasm programs to efficiently utilize hardware capabilities across diverse platforms. Finally, the WASI interface facilitates its portability.

Owing to its security advantages, Wasm is rapidly being adopted across both Web and non-Web domains. In Web development, Wasm has become one of the four official languages, receiving full support from major browsers [28] [29]. Beyond the Web, Wasm is widely utilized in various scenarios such as cloud computing [30] [31] [32] [33], edge computing [34] [35] [36], and server-side computing [37].

2.2 Privilege Separation

Privilege separation [38] [39] [40] is a security architecture designed to minimize the impact of vulnerabilities by partitioning a program into components with different privilege levels. This approach reduces the attack surface by isolating sensitive operations into distinct components while restricting their interactions. Common implementations of privilege separation include Software Fault Isolation (SFI) [41] and Remote Procedure Call (RPC) [42] [43], among others. These techniques establish strict boundaries between components, ensuring that a compromised or malicious module cannot compromise the integrity of the entire system.

SFI is a mechanism designed to sandbox untrusted code by confining it within a restricted memory space. This technique modifies code during compilation to ensure it accesses only authorized memory regions and operates within predefined boundaries. By enforcing strict isolation, SFI is particularly useful in scenarios where external, potentially malicious code needs to execute within a host environment. A notable application of SFI is its implementation in the Native Client (NaCl) [44] framework. NaCl applies SFI to securely run native code in web applications, enabling performance comparable to native execution while maintaining strong security guarantees. RPC enables communication between components operating in separate address spaces, whether on the same machine or distributed across a network. RPC abstracts the complexities of inter-process communication by allowing a program to invoke procedures on a remote system as if they were local functions. This approach is widely employed in systems requiring privilege separation, facilitating secure and structured communication between isolated components. A practical example of RPC is its implementation in Rust's sandboxing framework, Sandcrust [26]. In such systems, safe code and unsafe APIs execute as separate processes, communicating via RPC to enforce strict privilege boundaries.



```
R1 fn rust_fn() {
R2     let mut n = Box::new(1);
R3     unsafe {
R4         c_func(&mut *n);
R5     }
R6 }    //Double free
```

```
C1  void c_func(int *p) {
C2          void (*fp) (int *);
C3           fp = &wrapper1;
C4           fp(p);
C5          }
C6  void wrapper1(int *p) {
C7           free(p);
C8}
```

```
w1(module
w2 ......
w3 (func $c_func (param i32)
w4   call_indirect (type 2)
w5   ......)
w6(func $wrapper1 (param i32)
w7   call $free
w8   ......)
w9 (func $_ZN20c_in_rust_doublefree7rust_fn17he9ba73d31c5ffbe5E
w10   call $c_func
w11   ......
w12   call $_ZN4core3ptr49drop_in_place$LT$alloc..boxed..Box...
w13   ......)
w14(func $_ZN4core3ptr49drop_in_place$LT$alloc..boxed..Box...(param i32)
w15   call $_ZN5alloc5alloc8box_free17ha5956bfa5e324c86E
w16   ......)
        ...
w17(func call $__rdl_dealloc (param i32 i32 i32)
w18   call $free
w19   ......)
```

Figure 1: A DF bug in Wasm compiled from Rust/C program.

3. Motivations, Challenges, and Threat Model

In this section, we present the motivation (§ 3.1) through a running example, followed by the security challenges (§ 3.2), and the threat model (§ 3.3) for this work.

3.1 Motivations

Despite the security assurances of Wasm, recent evidences [9] [10] have revealed that many memory security vulnerabilities present in native binaries may also be exploitable in Wasm binaries. For instance, while Wasm's separation of unmanaged memory protects the return address from corruption, it does not safeguard sensitive data stored in linear memory. As a result, memory vulnerabilities caused by unsafe functions such as `malloc` and `free` can lead to metadata corruption or heap overflows within linear memory, as these regions are contiguous. These findings highlight the urgent need for novel security techniques to protect Wasm's linear memory and ensure more comprehensive safeguards.

To better illustrate the motivation behind our research, we present a running example in Fig. 1 to demonstrate how memory vulnerabilities arise in Wasm when interacting with unsafe APIs. As shown in Fig. 1, a Rust function calls a foreign function `c_func` defined in C program (❶), which in turn calls the function `wrapper1` via a function pointer `fp` to reclaim an object (❷). However, the Rust program is unaware that `n` has been deallocated by the C function, and thus attempts to release it automatically when `n` goes out of its lexical scope (line R6), leading to a double-free (DF) bug. Meanwhile, in the Wasm binary that is generated from both the Rust and C functions (❹ and ❺), the object `p` is first reclaimed by the C function (❻), and is reclaimed again by the `drop_in_place` function (❼), propagating the DF bug from the C function into the Wasm binary.

Unfortunately, existing protections for Rust/C programs are limited in handling such issues. For example, FFIChecker, a Rust/C multilingual program analyzer [45], fails to analyze such issue s because it cannot analyze function pointers like the `wrapper1` function above. Meanwhile, indirect function calls are ubiquitous in Wasm binaries to invoke unsafe APIs in libraries, posing additional security challenges [46].

In Section 5.6, we will demonstrate how our approach can safeguard the vulnerabilities as the one in this example.

3.2 Security Challenges

Despite the security criticality and urgency [9] [10] [11], to the best of our knowledge, memory protections and enhancements for Wasm have not been thoroughly studied. And we must tackle several technical challenges to develop a privilege separation approach for Wasm protection.

C1: lacking of architecture support. Privilege separations are normally supported by leveraging specific hardware security features such as Intel MPK [23]. However, since Wasm binaries are executed by Wasm VMs that do not have such architectural security features, enforcing privilege separation for Wasm programs is challenging.

Solution: We utilize the idea of RPC to achieve privilege separation for Wasm programs. Specifically, we leverage the wRPC standard specification [48], a latest RPC protocol and framework for Wasm based on WebAssembly Interface Types (WIT) [49]. To implement our approach, we modify Wasm code to instrument unsafe APIs with RPC wrappers adhering to the WIT specification, and enable inter-process communication through wRPC.

C2: language and toolchain diversity. Wasm has a diverse ecosystem with various toolchains (e.g., Emscripten [50] and Rustc [16]) that support a wide range of source languages (e.g., C/C++ and Rust). Moreover, the Wasm RPC relies on a specific *wasm-wasip2* version [48], which differs from general Wasm binaries. However, to the best of our knowledge, transformation tools for converting between these versions

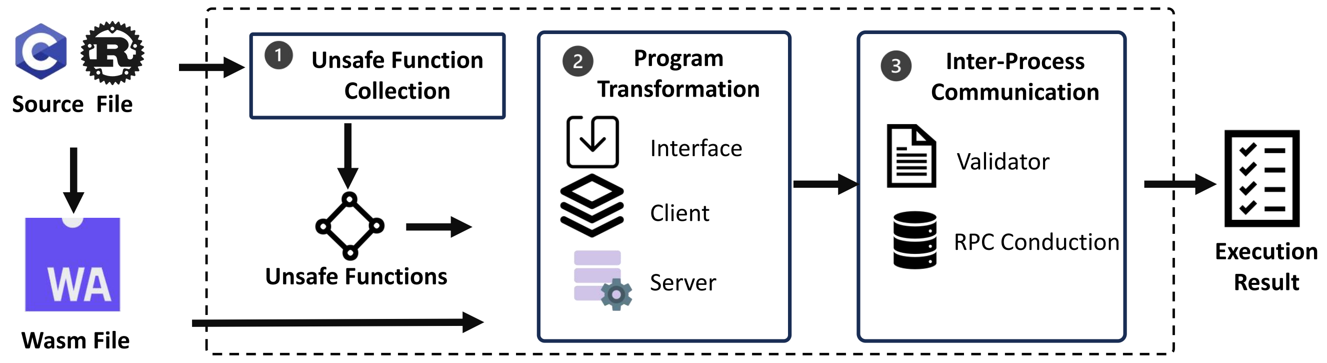


Figure 2: An overview of WASMSEPA's workflow.

have not yet emerged. Consequently, developing a holistic protection across diverse source languages is challenging.

Solution: We propose a binary-level protection strategy that operates directly on Wasm binaries while incorporating source-language information, leveraging the standard Wasm RPC specification [48]. To implement this strategy, we design an unsafe API wrapping mechanism that utilizes unsafe function information collected from source programs. Based on this, we generate client Wasm binaries from safe code and server Wasm binaries from unsafe libraries.

C3: lacking of unsafe API information. The standard Wasm does not inherently distinguish between unsafe and safe functions, which poses challenges for determining unsafe functions.

Solution: To address this challenge, we analyze unsafe function information at the source level and propagating such information to the generated Wasm binaries. Specifically, we assume that all imported functions from unsafe languages and external libraries are potentially unsafe. To mitigate risks, we wrap and isolate these functions in a separate process which are invoked through RPC, thereby protecting the memory of safe binaries from potential attacks originating from unsafe functions.

3.3 Threat Model

Wasm has a rich ecosystem comprising high-level language support, compilation toolchains, external libraries, and Wasm VMs, among others. This work focuses on Wasm memory protection through privilege separation. Accordingly, we define the threat model with the following assumptions.

We assume that Wasm VMs and RPC mechanisms used to execute Wasm programs are secure and trustworthy. On the one hand, extensive security studies have been conducted to Wasm security [51]. On the other hand, our work is orthogonal to Wasm security studies, indicating that our approach complements and benefits from advancements in those fields.

We assume that both source code and Wasm binaries may be vulnerable and thus unreliable. On the one hand, vulnerabilities in insecure source code can propagate into Wasm binaries [9]. On the other hand, design flaws or implementation bugs in Wasm compilers may introduce vulnerabilities into generated Wasm binaries.

4. APPROACH

In this section, we present our approach for effective Wasm memory protection through privilege separation called WASMSEPA. We begin by introducing the design goals (§ 4.1), followed by an overview of its workflow (§ 4.2). We then detail the design and implementation of each component (in § 4.3 through § 4.5).

4.1 Design Goals

The design of WASMSEPA is guided by three primary goals. First, WASMSEPA should effectively safeguard Wasm memory from a wide range of vulnerabilities, including but not limited to buffer and heap overflows. Second, WASMSEPA should provide holistic memory protection across *all* platforms that support Wasm binaries and Wasm VMs. Third, WASMSEPA should be an automatic, end-to-end solution with minimal user intervention, while providing a user-friendly interface to assist users in identifying and diagnosing issues.

4.2 Overview

With these design goals in mind, we present an overview of WASMSEPA's workflow in Fig. 2, comprising three key components: unsafe function collection, program transformation, and inter-process communication. First, the unsafe function collection (❶) processes source programs by scanning and identifying all unsafe function calls, producing a list of these functions for subsequent processing. Second, the program transformation (❷) takes the Wasm binary along with the identified unsafe functions as input and applies unsafe function wrapping. Third, the inter-process communication (❸) takes the wrapped program as input to generate corresponding Wasm binaries to facilitate RPC execution.

4.3 Unsafe Function Collection

The unsafe function collection processes a source program and produces a list of unsafe functions for subsequent analysis.

Vulnerabilities in Wasm binaries often originate from unsafe code, such as C/C++ functions. Therefore, distinguishing between safe and unsafe code is crucial for privilege separation. To be practical, the approach to separate those two parts of code must be non-invasive. As a result, a logical boundary for this distinction is the foreign functions called from unsafe library APIs, so our approach identifies and collects all unsafe APIs as unsafe code.

Foreign functions are usually called through foreign function interfaces (FFIs) (e.g., Python/C API [52] or Java Native Interface (JNI) [53]), which enable interaction between different languages but often introduce vulnerabilities. To guarantee security, we apply program transformation to the safe-language wrappers using the WIT [49] specification rather than modifying the original unsafe functions. Our approach ensures that all unsafe function calls remain confined to the sandboxed portion of the program, thereby preserving the security guarantees of the main Wasm binaries.

However, during Wasm binary generation, FFI information is lost, and foreign functions are treated as ordinary function calls. To tackle this issue, we directly collect all FFIs at the source program level to construct a list of unsafe functions. As a result, if FFI information is present in source programs, we extract it for subsequent analysis without modifying the original code. Otherwise, we just assume that a predefined set of C functions from standard C libraries is unsafe, or more conservatively, all functions in the external libraries are unsafe. Based on our finding, we argue that a long term solution might be an addition of unsafe function information into the Wasm binaries, which in turn requires an extension of the Wasm standard.

4.4 Program Transformation

We utilize a program transformation to partition the Wasm binary into client and server components that comprises safe and unsafe code, respectively. Specifically, we first identify the interfaces of all unsafe APIs that require RPC according to the WIT specification, then we wrap all local FFIs into corresponding RPCs as the client, and isolate all imported foreign functions into a separate program as the server. Next, we detail the design of these three components, respectively.

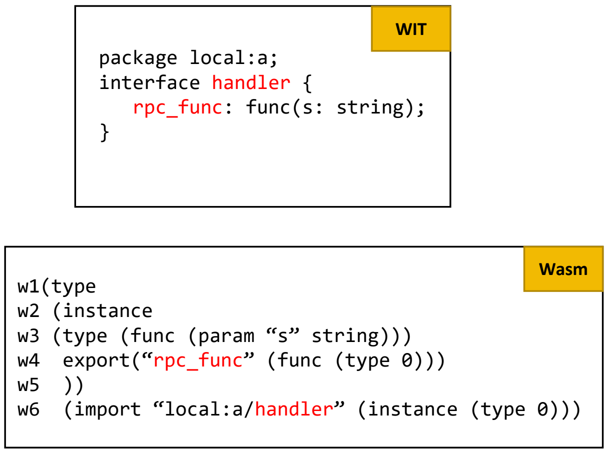
Transforming function declarations to interfaces. We first utilize the collected unsafe function information that includes all unsafe APIs called through FFIs, to define the corresponding interfaces according to the WIT specification.

Specifically, we use the `interface` keyword in WIT to define a named set of types and functions that are called through RPC between two processes [54] [55]. We treat different Wasm binaries as different Wasm Component Model [56]. While communicating through RPC, a component must import a set of functions from the host or other components, thus we must explicitly define and record all functions and types with interfaces.

To tackle the issue of source language diversity and discrepancies, we convert type definitions in source languages into the corresponding WIT format. As an example, Table I presents common type representations in Rust and WIT. Based on these representations, we transform each unsafe function declaration into the appropriate WIT format. For instance, a function declaration in Rust `foo(a: i32, b: Vec<u8>)->String` is transformed to the target of `foo: func(a: s32, b: list<u8>)->string`, so that the function signatures can be correctly recognized by subsequent components.

We next generate WIT files that declare FFIs for both the client and server, respectively. We first collect a list of function names, parameter types, and return types for all unsafe functions. Then, we convert this information into WIT format using the WIT specification for type representation. For instance, for a C function with the declared interface `void c_func(string s)`, Fig. 3 shows the corresponding generated WIT file. This file is then compiled into a Wasm binary and embedded into both the client and server binaries.

Transforming unsafe functions to servers. After transforming declarations of all unsafe functions, we transform these unsafe functions in Wasm binary into a server component, which encapsulates the unsafe libraries and exposes their functionality through RPC endpoints defined by the interface. The server executes the unsafe functions to serve RPC requests from the client, by providing the backend logic while ensuring that operations remain confined to the sandboxed environment. We wrap all local function calls in original binary into corresponding RPC functions, without altering the logic of foreign



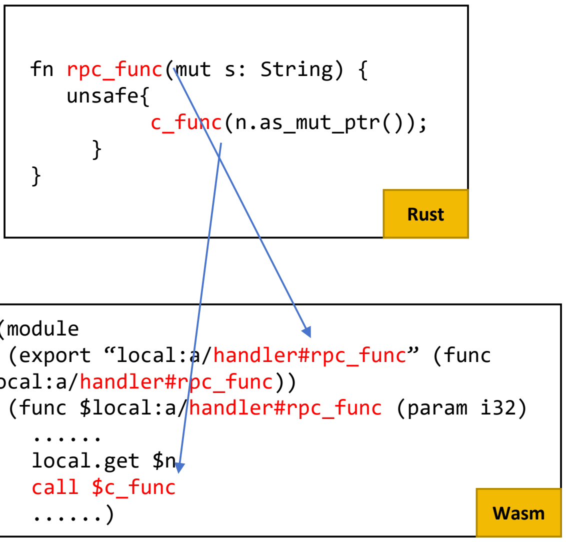
```
package local:a;
interface handler {
   rpc_func: func(s: string);
}
```

```
(type
 (instance
 (type (func (param "s" string)))
  export("rpc_func" (func (type 0)))
  ))
  (import "local:a/handler" (instance (type 0)))
```

Figure 3: Sample interface definitions in a WIT file.

TABLE I: Typical type representation of WIT and Rust.

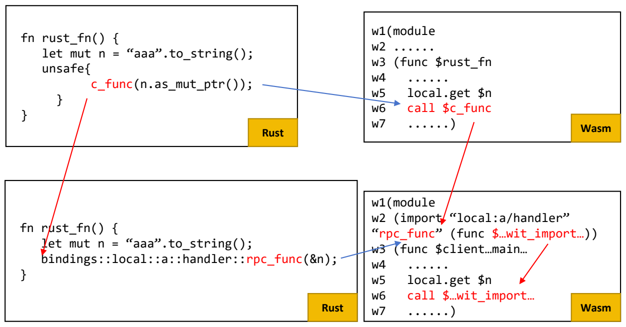
WIT	Rust	Description
u8	u8	Unsigned 8-bit integer
s32	i32	Signed 32-bit integer
f32	f32	32-bit floating-point number
string	String	UTF-8 encoded string
bool	bool	Boolean value
list<T>	Vec<T>	Dynamic array
option<T>	Option<T>	Optional value
record	struct	Named fields in a record



```
fn rpc_func(mut s: String) {
   unsafe{
          c_func(n.as_mut_ptr());
     }
}
```

```
w1(module
w2 (export "local:a/handler#rpc_func" (func
$local:a/handler#rpc_func))
w3 (func $local:a/handler#rpc_func (param i32)
w4   ......
w5   local.get $n
w6   call $c_func
w7   ......)
```

Figure 4: Server transformation example.



```
fn rust_fn() {
   let mut n = "aaa".to_string();
   unsafe{
          c_func(n.as_mut_ptr());
     }
}
```

```
w1(module
w2 ......
w3 (func $rust_fn
w4   ......
w5   local.get $n
w6   call $c_func
w7   ......)
```

```
fn rust_fn() {
   let mut n = "aaa".to_string();
   bindings::local::a::handler::rpc_func(&n);
}
```

```
w1(module
w2 (import "local:a/handler"
"rpc_func" (func $…wit_import…))
w3 (func $client…main…
w4   ......
w5   local.get $n
w6   call $…wit_import…
w7   ......)
```

Figure 5: Client transformation example.

functions, regardless of whether the source code for those functions is available. In this case, our approach ensures that privilege separation remains effective, even when the unsafe code itself is inaccessible.

To illustrate the definition of RPC functions, we revisit the Rust program shown in Fig. 1 and present the corresponding server implementation in Fig. 4. In this case, the function `rpc_func` has a parameter of the type `String`, while in the Wasm binary, the parameter is represented as an `i32` (line w3) and is exported for client access (line w2). We directly wrap the original local unsafe function into the corresponding RPC function (line w6) that defined previously in the WIT file. While the client calls the RPC function as declared in the WIT file, the server executes the target function, thereby enforcing privilege separation.

Transforming safe programs to clients. Finally, we transform safe components in the original Wasm binary into a client, which calls unsafe functions through RPC defined in other components.

The client acts as the front end for calling functions defined in the interface, enabling safe interaction with the unsafe library through privilege separation. It translates local function calls into RPC requests, replacing direct calls to unsafe functions with client-side bindings generated from the WIT schema. Since parameter and return types may be pointers, we define the original data types in WIT and retrieve the corresponding object's pointer on the server side.

To generate the client-side Wasm binary, we first create a Wasm binary template comprising only the imports of all RPC functions and other necessary functions. Next, we integrate these function imports with the original Wasm binary and replace all local function calls to unsafe functions with corresponding RPC calls.

As a case study, we investigate a Wasm binary generated from a Rust program that calls a C function with an argument of string pointer, as shown in Fig. 5. Before program transformation, the unsafe function `c_func` is called in Rust program through FFI, taking a pointer as a parameter. However, in the compiled Wasm binary, the FFI call is compiled into a standard function invocation (line w6). Our transformation converts the function signature into WIT format and wraps the foreign function into an RPC-callable function that accepts a string parameter according to the WIT specification, as shown in Fig. 3. The WIT file in this case defines an interface named `handler` (line w6) within the package `local:a` field, and declare the RPC function (line w4). As a result of the function redeclaration, the direct local call to `c_func` is converted into an RPC call to the `rpc_func` function defined in the interface `handler` defined in WIT file as a package `local::a` [57], also following the WIT specification [49]. During compilation to Wasm binary, the RPC function is imported from the server side (line w2), and be called in place of the original local function call (line w6).

4.5 Inter-Process Communication

We utilize two isolated processes to run the safe Wasm binary program as the client and Wasm binary of unsafe libraries as the server. And we leverage inter-process communications to implement the function calls. To guarantee security, we first utilize a Wasm validator that supports *wasm32-wasip2* as the target to validate Wasm binaries of both the client and server, respectively. Once validation is complete, inter-process communication is performed through RPC. Below, we detail the design of these two components, respectively.

Validator. The validator validates the Wasm binaries of both the client and server. To enable RPC-based communication between the two Wasm binaries running in separate processes, the compilation target must be set to a specifical format *wasm32-wasip2* according to the official standard [48] [58]. This standard specifies rigorous type system and semantics, making it modular and accessible to various source languages. Several Wasm compilers, including Cargo [59] for Rust and TinyGo [15] for Go, support compiling to the wasm32-wasip2 target. Since the transformed program incorporates WIT information, and the original Wasm binaries are typically wasm32-wasi version, validation of the client and server Wasm binaries can be performed independently without modifying the compiler.

RPC communication. We utilize the wRPC [48] specification to perform the RPC communication between the generated Wasm, thereby enforcing privilege separation between the safe and unsafe functions.

We select the wRPC because it is a component-native, transport-agnostic RPC protocol and framework, which enables execution of arbitrary functionality defined in WIT. Furthermore, it also supports both dynamic and static scenarios such as generic Wasm components that should be executed within a Wasm runtime. Communications between the main program and the unsafe library via wRPC incorporate the following key steps: 1) arguments serialization: function arguments are serialized into a format compatible with the WIT specification; 2) remote invocation: the client sends the serialized request to the server running in a separate process, invoking the appropriate function. And the details for the underlying network communication are same as the normal RPC process, so that we can reuse the off-the-shelf underlying network infrastructure; and 3) response handling: the server executes the function and returns serialized results, which the client deserializes back into data and types of the source language.

To achieve portability, we perform the RPC communications through any existing Wasm runtime that supports wRPC, treating both the client and server Wasm binaries as generic Wasm components.

4.6 Prototype Implementation

To validate our approach, we design and implement a software prototype for WASMSEPA, specifically for Wasm binaries that compiled from Rust/C multilingual programs as source languages. However, our approach also applies to any other language combinations such as Go/C, because our approach does not rely on any specific source language features. Next, we highlight some key implementation details.

Unsafe function collection. We implement the unsafe function collection by adapting and extending the FFI collection component from FFIChecker [23], a state-of-the-art tool for Rust program analysis capable of extracting all FFIs from Rust code. We also utilize the Clang compiler [24] to collect function type information for all FFIs.

Program transformation. We implement the program transformation using Python scripts with Python version 3.12.3 and a set of Shell scripts. Specifically, our Python scripts handle WIT file generation for interface definitions, function wrapping on the server side, and function call replacement on the client side. Moreover, we utilize the Shell scripts to perform Wasm binary generation, by leveraging Cargo [59] and Clang from wasi-sdk [61] to compile Rust and C programs, respectively.

Inter-process communication. We implement the validator in the inter-process communication by utilizing wasm-tools [62], an official toolset for low-level Wasm module manipulation, to validate client and server Wasm binaries targeting the *wasm32-wasip2* format. We implement the RPC component, in the inter-process communication by leveraging wrpc-wasmtime [48], a Wasm runtime that supports RPC execution for *wasm32-wasip2* binaries using TCP transport.

5. EVALUATION

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

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

RQ2: Usefulness. As a tool designed to enhance the security of Wasm programs, is WASMSEPA capable to protect memory for real-world applications?

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

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.

5.1 Datasets

We conduct the evaluation using two datasets: 1) a set of micro-benchmarks, consisting of 16 vulnerable programs we selected and created; and 2) a set of real-world benchmarks, containing a total of 25 vulnerable programs from real-world CWEs and 10 real Rust/C projects.

Micro-benchmarks. We manually construct a set of micro-benchmarks comprising 16 test cases including common vulnerabilities in Rust/C programs. Each C program was wrapped as a library and invoked by a Rust program. To better emphasize the importance of privilege separation, we have simplified some of the original code by removing irrelevant fragments. As shown in the second column of Table II, these micro-benchmarks comprises various vulnerabilities, including stack-based buffer overflow, null-pointer dereference, and use-after-free, among others. We create these micro-benchmarks from Rust/C source code because manually constructing Wasm test cases containing FFIs by directly composing Wasm binary instructions is both labor-intensive and error-prone. Furthermore, manually injecting memory vulnerabilities and FFIs into Wasm binaries is challenging, as Wasm binaries must conform to Wasm's strict semantic specification [63].

Real-world applications. CWE [64] is a collection of vulnerable C programs that contain various vulnerabilities such as buffer overflows and integer overflows. Evaluating WASMSEPA on well-established vulnerability sets provides an effective means to validate its usefulness. We add a Rust wrapper to each C code to transform it into a Rust/C multilingual program, and then compile each resulting program into its corresponding Wasm binary.

Additionally, applying WASMSEPA to off-the-shelf real-world Rust/C projects further demonstrates the usefulness of our

TABLE II: Experimental results on micro-benchmarks.

Test Case	Vulnerability Type	LoC BT / LoC AT	LoC Overhead	TT (s)	EXE time BT (s) / EXE Time AT (s)	EXE Time Overhead	WasmSepa	Fuzzm	FFIChecker
1	IO_1	24938 / 33226	33.2%	6.93	0.013 / 0.093	7.15×	✗	✗	✗
2	IO_2	24941 / 33252	33.3%	7.23	0.015 / 0.101	6.73×	✗	✗	✗
3	DF_1	34040 / 52756	55.0%	6.91	0.017 / 0.122	7.18×	✓	✗	✓
4	DF_2	34071 / 53098	55.8%	7.03	0.019 / 0.127	6.68×	✓	✗	✗
5	HBO_1	25310 / 33631	32.9%	6.98	0.011 / 0.088	8×	✓	✓	✗
6	HBO_2	29597 / 48500	63.9%	8.36	0.015 / 0.102	6.8×	✓	✓	✗
7	HBO_3	25011 / 33647	34.5%	6.93	0.018 / 0.106	5.89×	✓	✓	✗
8	SBO_1	20410 / 28905	41.6%	8.89	0.009 / 0.082	9.11×	✓	✓	✗
9	SBO_2	23911 / 37289	56.0%	6.93	0.015 / 0.087	5.8×	✓	✗	✗
10	SBO_3	23892 / 37275	56.0%	6.88	0.010 / 0.093	9.3×	✓	✓	✗
11	ML_1	29397 / 48113	63.7%	7.04	0.012 / 0.101	8.42×	✗	✗	✓
12	ML_2	33978 / 53014	56.0%	7.52	0.014 / 0.112	8×	✗	✗	✓
13	NPD_1	25021 / 46167	84.5%	6.92	0.011 / 0.086	7.82×	✓	✗	✗
14	NPD_2	24936 / 33530	34.5%	7.28	0.010 / 0.098	9.8×	✓	✗	✗
15	UAF_1	24939 / 33225	33.2%	6.84	0.010 / 0.077	7.77×	✓	✗	✓
16	UAF_2	24936 / 33533	34.5%	7.35	0.014 / 0.108	7.71×	✓	✗	✓

LoC: Line of Code; BT: Before Transformation; AT: After Transformation; TT: Transformation Time.

approach. We select real programs based on three principles: 1) the projects should be open source, so that we can obtain the source to compile them to Wasm; 2) the projects could be compiled to Wasm easily so as to employ our approach; and 3) the projects must either include known memory vulnerabilities or be written in memory-unsafe languages. As a result, we select 10 programs from FFIChecker [45] as real-world Rust/C benchmark.

5.2 Evaluation Metrics

We use the *precision* and *recall* metrics to evaluate the effectiveness of WasmSepa. These metrics are defined in the equation 1.

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

In these equations, tp, fp, and fn represent true positives, false positives, and false negatives, respectively. Additionally, we compute the $F1$ score as shown in equation 2.

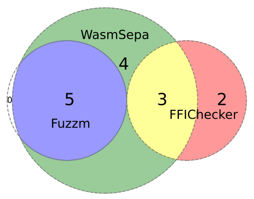
$$F1\,score = \frac{2 \times precision \times recall}{precision + recall} \tag{2}$$

The F1 score provides a balanced measure of a tool's accuracy by considering both *precision* and *recall*.

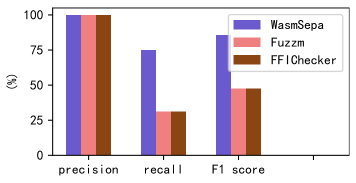
5.3 RQ1: Effectiveness

To answer RQ1, we first apply WasmSepa to the micro-benchmarks to evaluate its effectiveness. We first compile the micro-benchmarks into their corresponding Wasm binaries using the Cargo compiler [59], and then employ WasmSepa to enforce privilege separation for each binary.

The 8th column (i.e., WasmSepa) of Table II presents the experimental results. Among the 16 benchmarks, WasmSepa successfully protects 12, but fails on 4 cases. Consequently, the recall of WasmSepa is 75%, and the precision is 100%, yielding an F1 score of 85.7%. These results demonstrate that WasmSepa is effective in protecting Wasm memory.



(a) Bug detection capability.



(b) Evaluation metrics.

Figure 6: A comparison of WasmSepa and two state-of-the-art tools Fuzzm [18] and FFIChecker [45].

To further investigate the root causes leading to the 4 failed cases, we conduct a manual inspection of relevant source code. This inspection reveals two key root causes: First, privilege separation cannot prevent semantically incorrect results produced by unsafe code, as the test case 1 and 2 show, which contain integer overflows. This is a fundamental limitation of any sandboxing solution [26]. However, such incorrect results do not affect sensitive data in memory if the result is not utilized for memory management. Second, memory leaks in Wasm binaries could not be mitigated by privilege separation, as the test case 11 and 12 show, where allocated objects may not be properly released in isolated processes.

Moreover, to understand the technical advantages of WasmSepa, we compare WasmSepa with two state-of-the-art security tools: 1) Fuzzm [18], a tool for Wasm; and 2) FFIChecker [45], a Rust/C program analyzer, to evaluate their effectiveness using the same set of micro-benchmarks. We first apply WasmSepa, FFIChecker, and Fuzzm to the micro-benchmark, and then compare their execution results.

The last three columns of Table II present the experimental results. Out of the 16 vulnerabilities across 7 categories, WasmSepa successfully detects 12 vulnerabilities in 5 cate-

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

Dataset	Total	Success	Recall	Precision	F1-score
CWE [64]	25	21	84%	100%	91.3%
Real [45]	10	9	90%	100%	94.7%
Total	35	30	85.7%	100%	92.3%

gories. In contrast, Fuzzm detects only 5 vulnerabilities across 2 categories, while FFIChecker detects 5 vulnerabilities in 3 categories.

Furthermore, we analyze the protection capabilities and evaluation metrics of these tools, and show the results in Fig. 6. First, as the Venn diagram in Fig. 6a shows, WASMSEPA mitigates all 5 vulnerabilities detected by Fuzzm, and 7 additional vulnerabilities that Fuzzm missed. Similarly, WASMSEPA mitigates 3 of the vulnerabilities detected by FFIChecker, and 9 additional vulnerabilities that FFIChecker missed. However, WASMSEPA fails to mitigate memory leaks that detected by FFIChecker. Notably, all three tools were unable to detect integer overflows. These results demonstrate WASMSEPA outperforms the state-of-the-art tools regarding protection capabilities.

Second, the histogram in Fig. 6b compares the evaluation metrics. While all three tools WASMSEPA, Fuzzm, and FFIChecker achieve a precision of 100%, WASMSEPA attains a recall of 75%, significantly outperforming both Fuzzm and FFIChecker, which each achieves only 31.3%. Consequently, WASMSEPA achieves a higher F1 score of 85.7%, compared to 47.6% for both Fuzzm and FFIChecker. These results highlight the greater overall effectiveness of WASMSEPA in program protection.

5.4 RQ2: Usefulness

To answer RQ2, we apply WASMSEPA to real-world benchmarks that are compiled from sources to Wasm binaries. We record the vulnerabilities that could be mitigated by WASMSEPA in these real-world programs, and compare them with the pre-annotated vulnerabilities, We then count the number of vulnerabilities WASMSEPA successfully mitigated in real-world benchmarks.

As Table III shows, WASMSEPA successfully detects 30 out of the 35 vulnerabilities, while missing 5. These results yield a recall of 85.7%, a precision of 100%, and an F1 score of 92.3%. This result demonstrates that WASMSEPA is useful to mitigate vulnerabilities in real-world programs.

Furthermore, we investigate the root causes leading to the 5 failure cases. After manually inspecting these Wasm cases, we reveal that the root causes for failure aligns with those discussed in Section 5.3. Specifically, two cases are caused by memory leaks, and three are cause by semantically incorrect results. However, these limitations do not reflect design flaws in WASMSEPA itself in real-world projects, but the limitations of the privilege separation technology used by WASMSEPA.

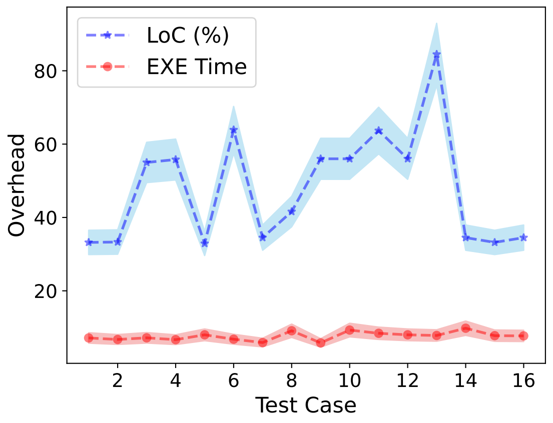


Figure 7: The file sizes and execution time changes introduced by WASMSEPA.

5.5 RQ3: Overhead

To answer RQ3, we investigate the overhead introduced by WASMSEPA, including 1) the time required for program transformation; 2) the code size increase of the Wasm binaries; and 3) the execution time penalty. To this end, we first record the code size of Wasm binaries compiled from the micro-benchmark, then run each binary 20 rounds to compute the average execution time, following prior work [17]. We then apply WASMSEPA to generate transformed Wasm binaries, then repeat the above process on each generated binaries. Finally, we calculate the average as well as changes regarding code size and execution time.

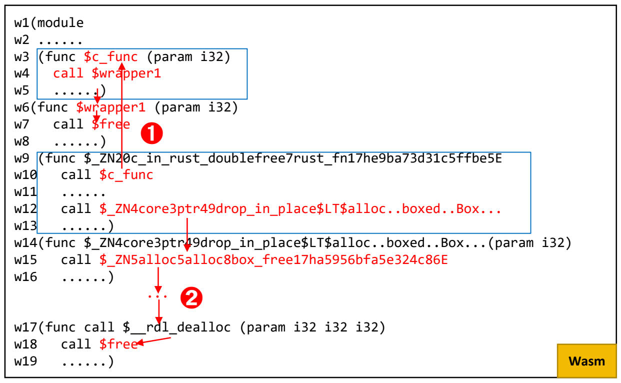
We present experimental results in Table II. The third column shows the lines of code (LoC) before and after the program transformation, respectively, while the fourth column shows the code size increases, ranging from 32.9% to 84.5%. Similarly, the sixth column presents the execution time of the relevant Wasm binaries before and after the transformation, respectively, while the seventh column shows the execution time increases, ranging from 5.8× to 9.8×.

Furthermore, Fig. 7 summarizes the average increases regarding code size and execution time introduced by WASMSEPA, which average 48.0% and 7.63×, respectively. These results align with prior work [26], since the execution time increase caused by Sandcrust ranges from 2× to 7.4×, indicating that the overhead introduced by WASMSEPA is acceptable.

Furthermore, the fifth column in Table II shows the time WASMSEPA used to perform program transformation. WASMSEPA takes an average of 7.25 seconds to process each test case, with a range from 6.84 to 8.89 seconds. The time is primarily spent on compilation step to generate the Wasm template, which requires compiling the source code into Wasm.

5.6 Case Study

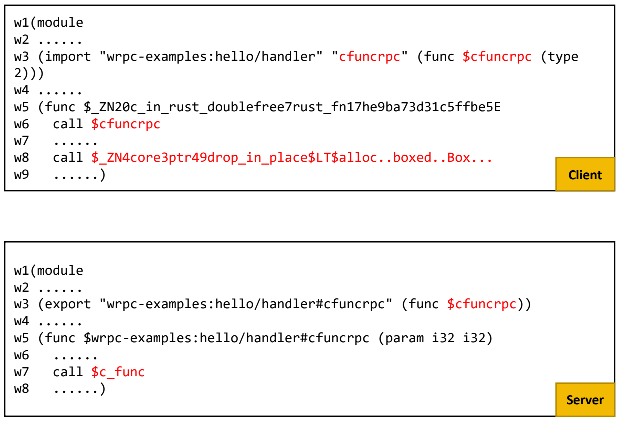
To better understand the capabilities of WASMSEPA, we conduct a case study of how WASMSEPA protects real-world Wasm programs.



```
w1(module
w2 ......
w3 (func $c_func (param i32)
w4   call $wrapper1
w5   ......)
w6(func $wrapper1 (param i32)
w7   call $free
w8   ......)
w9 (func $_ZN20c_in_rust_doublefree7rust_fn17he9ba73d31c5ffbe5E
w10   call $c_func
w11   ......
w12   call $_ZN4core3ptr49drop_in_place$LT$alloc..boxed..Box...
w13   ......)
w14(func $_ZN4core3ptr49drop_in_place$LT$alloc..boxed..Box...(param i32)
w15   call $_ZN5alloc5alloc8box_free17ha5956bfa5e324c86E
w16   ......)

w17(func call $__rdl_dealloc (param i32 i32 i32)
w18   call $free
w19   ......)
```

Figure 8: The initial Wasm binary compiled from the multi-lingual program.



```
w1(module
w2 ......
w3 (import "wrpc-examples:hello/handler" "cfuncrpc" (func $cfuncrpc (type
2)))
w4 ......
w5 (func $_ZN20c_in_rust_doublefree7rust_fn17he9ba73d31c5ffbe5E
w6   call $cfuncrpc
w7   ......
w8   call $_ZN4core3ptr49drop_in_place$LT$alloc..boxed..Box...
w9   ......)
```

```
w1(module
w2 ......
w3 (export "wrpc-examples:hello/handler#cfuncrpc" (func $cfuncrpc))
w4 ......
w5 (func $wrpc-examples:hello/handler#cfuncrpc (param i32 i32)
w6   ......
w7   call $c_func
w8   ......)
```

Figure 9: The Wasm binaries generated by WASMSEPA.

We present in Fig. 8 a sample vulnerable Wasm program and shows how WASMSEPA can protect it. In this case, the Wasm program is essentially the case in Fig. 1. Recall that function `rust_fn` calls a function `c_func` (❶), which is an unsafe function resulting in a DF bug (❷).

WASMSEPA eliminates the memory vulnerability by first transforming the original Wasm binary into two Wasm binaries of client and server, respectively, as Fig. 9 shows. In this case, the foreign function `c_func` is wrapped as a RPC in server-side Wasm binary (line w5 of the server), and the safe binary is rewritten into a client-side Wasm binary to call the relevant RPC functions instead of local functions (line w6 of client). As a result of this transformation, the initial single Wasm binary that compiled from multilingual programs are transformed to two distinct binaries comprising the server code with unsafe functions and client side with safe functions, respectively.

Furthermore, in these two resulting Wasm binaries, since the local function calls to the function `c_func` is transformed to an RPC function call (line w6 of client), the heap object is freed by the RPC function `cfuncrpc` (line w7 of server) as well as by the Wasm `drop_in_place` function (line w8 of client). Since our approach enforces privilege separation through RPC, the `c_func` reclaims the heap object in the memory of server process, while the `drop_in_place` function reclaims a distinct memory in client process. As the result, these two reclaimations of objects manifest in different memory address spaces in separate processes, thus mitigating the DF bug.

6. Discussion

In this section, we discuss some limitations of this work, along with directions for future work.

Higher accuracy. Although our approach offers an effective method for protecting Wasm linear memory, the scope of recorded unsafe function information is still limited. Consequently, vulnerabilities that require more detailed instruction information cannot be mitigated. Specifically, unsafe operations that do not involve FFIs cannot be handled by WASMSEPA. To address this limitation, one potential solution is to incorporate a more fine-grained unsafe operation collection mechanism into WASMSEPA. Specifically, we can record code that modifies pointers without memory protection mechanisms as unsafe. Additionally, we could analyze the control-flow graphs (CFGs) of each function in the Wasm binary using the Wasm static analyzer Wassail [65] to obtain more precise static information. We leave this as a direction for future work.

Other vulnerabilities. Although WASMSEPA effectively protects Wasm linear memory against various memory vulnerabilities as our experimental results demonstrate, it does not mitigate all types of vulnerabilities. Specifically, WASMSEPA, in its current design, does not detect memory vulnerabilities caused by race conditions, i.e., improper memory accesses by concurrent threads. To address this issue, we can build upon recent studies [66] to extend WASMSEPA's design and support concurrency semantics. Furthermore, Wasm has recently introduced concurrency features [67], which we plan to incorporate into the WASMSEPA extensions.

Incorrect results. WASMSEPA cannot protect Wasm programs that produce semantically incorrect results through privilege separation. This is a fundamental limitation of any sandboxing solution [26]. To address this issue, we can leverage other methods such as differential testing or taint analysis [68] to overcome these constraints.

7. Related Work

In recent years, there have been substantial studies on privilege separation and Wasm security enhancement.

Privilege separation. There has been considerable research on privilege separation. Lamowski et al. [26] introduce an automated sandboxing solution based on Rust macros, which isolates unsafe C libraries in separate processes via RPC to preserve Rust's memory safety guarantees while simplifying integration. Wu et al. [69] propose a transparent library isolation framework that uses a semi-shared memory model to support tight interactions with the main program, enabling sandboxing of dynamically linked libraries without source code modifications, thus balancing security and performance.

Almohri et al. [47] introduce a method for safely incorporating unsafe code into Rust programs by isolating sensitive data in memory, achieving strong isolation without requiring compiler modifications or complex abstractions, and enabling safe interactions with unsafe library functions.

However, a significant limitation of these studies is that they do not propose generating memory protection mechanisms for other languages, such as Wasm, as we present in this work.

Wasm security enhancement. There has been a lot of works on enhancing Wasm security. Narayan et al. [21] propose Swivel, a new compiler framework for hardening Wasm binaries against Spectre attacks. Jiang et al. [70] propose WasmFuzzer for fuzzing Wasm VMs. Song et al. present metaSafer [22] to protect heap metadata in Wasm linear memory. Keno et al. [71] introduce WAFL to fuzz the Wasm binaries with the aid of fact snapshots. Lei et al. [23] utilize MPK to protect Wasm linear memory at the function granularity level. Arteaga et al. [19] proposed the CROW system which statically transforms code using code diversification technology.

However, a key difference between these studies and our work is that existing studies have not proposed a comprehensive protection mechanism through privilege separation. In contrast, we propose a solution that leverages the capability of RPC to protect Wasm binaries.

8. Conclusion

In this work, we present an approach for protecting Wasm memory through privilege separation via RPC. Our approach leverages unsafe code information from the source language to design privilege separation for isolating Wasm binaries. We first design an unsafe function collection to gather unsafe FFIs from source language information. Next, we design a program transformation that rewrites Wasm binaries into the WIT format, including interface, client, and server definitions. We finally design inter-process communication using the Wasm validator and RPC conduction based on wRPC. We implement a software prototype called WasmSepa for our approach and conduct extensive experiments to evaluate its effectiveness, usefulness, and overhead. The evaluation results demonstrate that WasmSepa provides effective protection for Wasm memory with acceptable overhead, outperforming state-of-the-art approaches. Overall, this work represents a new step towards security enhancement of Wasm, making Wasm's promise as a safe binary 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] "Tinygo - go compiler for small places," https://github.com/tinygo-org/tinygo, Jul. 2023.

[16] "What is rustc? - the rustc book," https://doc.rust-lang.org/rustc/what-is-rustc.html.

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

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

[19] 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.

[20] 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.

[21] 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.

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

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

[26] B. Lamowski, C. Weinhold, A. Lackorzynski, and H. Härtig, "Sandcrust: Automatic sandboxing of unsafe components in rust," in *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, 2017, pp. 51–57.
[27] "WASI —," https://wasi.dev/.
[28] "V8 JavaScript engine," https://v8.dev/.
[29] "Safari," https://www.apple.com/safari/.
[30] 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.
[31] "Edge programming with Rust and WebAssembly — Fastly," https://www.fastly.com/blog/edge-programming-rust-web-assembly.
[32] "Lucet Takes WebAssembly Beyond the Browser — Fastly," https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime, Mar. 2019.
[33] "WebAssembly on Cloudflare Workers," http://blog.cloudflare.com/webassembly-on-cloudflare-workers/, Oct. 2018.
[34] "wasmCloud," https://wasmcloud.com/.
[35] "Fastly," https://learn.fastly.com/edgecompute-faster-simpler-secure-serverless-code.
[36] "WasmEdge," https://wasmedge.org/.
[37] "Deno — A modern runtime for JavaScript and TypeScript," https://deno.com/runtime.
[38] D. Brumley and D. Song, "Privtrans: Automatically partitioning programs for privilege separation," in *USENIX Security Symposium*, vol. 57, no. 72, 2004.
[39] D. Akhawe, P. Saxena, and D. Song, "Privilege separation in {HTML5} applications," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012, pp. 429–444.
[40] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012, pp. 71–72.
[41] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.
[42] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.
[43] B. J. Nelson, *Remote procedure call*. Carnegie Mellon University, 1981.
[44] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, 2010.
[45] Z. Li, J. Wang, M. Sun, and J. C. S. Lui, "Detecting Cross-language Memory Management Issues in Rust," in *Computer Security – ESORICS 2022*, V. Atluri, R. Di Pietro, C. D. Jensen, and W. Meng, Eds. Cham: Springer Nature Switzerland, 2022, vol. 13556, pp. 680–700.
[46] D. Lehmann, M. Thalakottur, F. Tip, and M. Pradel, "That's a Tough Call: Studying the Challenges of Call Graph Construction for WebAssembly," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Seattle WA USA: ACM, Jul. 2023, pp. 892–903.
[47] H. M. Almohri and D. Evans, "Fidelius charm: Isolating unsafe rust code," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 248–255.
[48] "Bytecodealliance/wrpc: Component-native transport-agnostic RPC protocol and framework based on WebAssembly Interface Types (wit)," https://github.com/bytecodealliance/wrpc.
[49] "An Overview of WIT," https://component-model.bytecodealliance.org/design/wit.html.
[50] "Emscripten-core/emscripten: Emscripten: An LLVM-to-WebAssembly Compiler," https://github.com/emscripten-core/emscripten.
[51] 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.
[52] M. Hu and Y. Zhang, "The python/c api: Evolution, usage statistics, and bug patterns," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb. 2020, pp. 532–536.
[53] "Java native interface specification contents," https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.
[54] "The WebAssembly Component Model," https://component-model.bytecodealliance.org/design/wit.html#interfaces.
[55] "The wit format," https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md#wit-interfaces.
[56] "Component Model design and specification," https://github.com/webassembly/component-model.
[57] "Package Names," https://github.com/WebAssembly/component-model/blob/main/design/mvp/WIT.md#package-names.
[58] "WASI Preview 2," https://github.com/WebAssembly/WASI/tree/main/wasip2.
[59] "The Cargo Book," https://rustwiki.org/en/cargo/commands/cargo-build.html.
[60] "Connective Technology for Adaptive Edge & Distributed Systems," https://nats.io/.
[61] "WASI SDK," https://github.com/WebAssembly/wasi-sdk.
[62] "Bytecodealliance/wasm-tools: A Bytecode Alliance project CLI and Rust libraries for low-level manipulation of WebAssembly modules," https://github.com/bytecodealliance/wasm-tools.
[63] W. C. Group and A. Rossberg, "WebAssembly Specification."
[64] "CWE - CWE-658:Weaknesses in Software Written in C (4.14)," https://cwe.mitre.org/data/definitions/658.html.
[65] Q. Stievenart 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)*. Adelaide, Australia: IEEE, Sep. 2020, pp. 13–24.
[66] 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.
[67] "Github-WebAssembly/threads: Threads and Atomics in WebAssembly." https://github.com/WebAssembly/threads.
[68] H. Sun, X. Zhang, C. Su, and Q. Zeng, "Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 483–494.
[69] Y. Wu, S. Sathyanarayan, R. H. Yap, and Z. Liang, "Codejail: Application-transparent isolation of libraries with tight program interactions," in *Computer Security–ESORICS 2012: 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings 17*. Springer, 2012, pp. 859–876.
[70] 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.
[71] H. Keno and M. Dominik, "WAFL: Binary-Only WebAssembly Fuzzing with Fast Snapshots," in *Reversing and Offensive-oriented Trends Symposium*, 2021, pp. 23–30.