PowerPoly: Analyzing Multilingual Programs with the Aid of WebAssembly

Zhuochen Jiang, and Baojian Hua(✉)

School of Software Engineering, Suzhou Institute for Advanced Research
University of Science and Technology of China, Suzhou 215123, China
jzc666@mail.ustc.edu.cn, bjhua@ustc.edu.cn

Abstract. Despite the ubiquity and importance of multilingual programming in modern software systems, it often introduces significant security vulnerabilities, particularly at language boundaries. Current approaches for analyzing multilingual systems are limited, typically focusing on specific language combinations like Java/C or Python/C, and lack generalizability. As a result, there is no clear framework for effectively analyzing multilingual programs in a unified manner.

In this paper, to fill this gap, we present PowerPoly, the first approach for analyzing multilingual programs that generalizes to diverse language combinations. Our key idea is to utilize WebAssembly, an emerging low-level code format originally designed for *execution*, as an intermediate representation for *analysis*. We first develop a unified intermediate representation utilizing WebAssembly to eliminate language boundaries by translating multilingual programs into this unified intermediate representation. We then showcase PowerPoly's capability for multilingual program analysis by first designing static analysis, then by designing a set of dynamic program analysis algorithms with user-supplied security plugins. To evaluate our approach, we design and implement a prototype for Rust/C and Go/C multilingual programs and conduct extensive experiments. Our results show that PowerPoly is effective in analyzing multilingual programs by detecting vulnerabilities. And the average Wasm binary code size increase of 10.2% and an average execution time penalty of 26.4%.

Keywords: Multilingual Programming · Vulnerability · Program Analysis · WebAssembly.

1 Introduction

Multilingual programming becomes increasingly pervasive and essential in modern software systems, allowing developers to effectively leverage complementary features from different languages. For example, NumPy [16] and PyTorch [38] comprise about 50% C/C++ code for backends, and 40% Python code for programming interfaces. As another example, Firefox contains 40% of C/C++ and 11.7% of Rust, among other languages [11]. Given the importance of multilingual

programming in modern cyberspace, guaranteeing its security and reliability is both critical and urgent.

Despite its criticality and urgency, secure multilingual programming remains a difficult task [43] [39][10], due to two main reasons. First, discrepancies between different and heterogeneous languages make multilingual programming challenging. Consequently, developers always struggle with subtle low-level syntactic and semantic differences, such as memory management [33], type systems [12], and exception handling [26], to avoid potential traps and pitfalls. Any overlook of these discrepancies leads to vulnerabilities or bugs that are difficult to detect and rectify, even for small-sized programs [22]. Second, even if each component within single-language is correct, vulnerabilities in multilingual programs can still arise at and across language boundaries [34], undermining the whole system's security guarantees. Therefore, providing an effective and holistic analysis for multilingual programs is imperative.

Recognizing this need, researchers have conducted a large amount of studies for analyzing multilingual programs [18] [27] [29] [19] [28] [44]. Generally, to effectively analyze multilingual programs, a general idea is to propose universal intermediate representations (IRs) to represent heterogeneous languages uniformly. Subsequently, static or dynamic program analysis are conducted on such IRs to obtain precise program information that are leveraged to reason about program properties. More importantly, to effectively analyze language discrepancies and boundaries that are unique to multilingual programming, existing efforts propose diverse approaches to represent and analyze foreign function interfaces (FFIs). This general idea has shown promising potentials for diverse multilingual programming paradigms. For example, ILEA [44] for Java/C proposes an IR by extending JVML for static program analysis, and represents language boundary information as pseudo-instructions in JVML. As another example, FFIChecker [28] for Rust/C utilizes the LLVM IR for lattice-based static analysis, and represents language boundary information through entry point and foreign function collection. Another powerful tool PolyCruise [27] for Python/C proposes an IR dubbed language-independent symbolic representation (LISR) to perform dynamic information flow analysis, and represents language boundary information as a dynamic information flow graph (DIFG).

Unfortunately, while existing efforts have made valuable contributions, they focus on specific language combinations thus lack generalizability. First, the IR design of existing efforts is intrinsically specialized for one certain language combination. For instance, MirChecker [29] utilizes MIR [41], an IR in Rust compiler, for analyzing Rust/C multilingual programs. However, it is difficult and costly to translate C to MIR because MIR comprises specific language features from Rust such as lifetime and borrow that C lacks. Second, the analysis algorithms in existing efforts are both diverse and making the migration of these algorithms from one IR to another one challenging and labor-intensive. For example, FFIChecker [28] utilizes LLVM IR [31] to implement abstract interpretation-based static analysis for Rust/C programs. However, it remains unclear how to migrate FFIChecker's analysis to other IRs (such

as LISR in PolyCruise [27]) that are designed for dynamic analysis thus lack the support for lattice required for static analysis. Even if the migration is possible, it remains labor-intensive due to the considerable volume of the analysis. Third, existing efforts' representation of language boundaries are still language-agnostic. For example, ILEA [44] represents Java/C language boundary information as pseudo-instructions in JVML. Consequently, it remains unclear how to represent Python/C or JavaScript/C combinations using this approach because JVML is designed for statically typed object-oriented languages instead of dynamically typed ones.

Insight. In this paper, we aim to answer the following unanswered questions: can we provide a holistic framework with right IR abstractions that developers can use to analyze any multilingual programs? In other words, our goal is no longer tied the analysis to a specific language combination or analysis, and to instead any potential combinations facilitating user-customizable analyses. We argue that such a holistic framework should satisfy three requirements. First, the framework should be language-neutral. It should support various high-level source language combinations. Second, the framework should be expressive. Various static and dynamic program analysis should be easily developed in this framework. Third, the framework should be cost-effective. The representation of language boundaries should be uniform and incur no extra cost to adapt to support new languages.

We present PowerPoly, the first framework for effective and holistic multilingual program analysis. Our key idea is to utilize WebAssembly (Wasm), an emerging binary instruction set architecture originally designed for secure binary *execution*, as the platform for multilingual program *analysis*. We argue that our selection of Wasm satisfies the aforementioned three requirements. First, we utilize Wasm as a language-neutral IR in PowerPoly to exploit Wasm's rich ecosystem comprising diverse high-level languages (e.g., C/C++ [9], Rust [40], Python [50], and Go [46]). In the meanwhile, with Wasm's support for multithreading and garbage collection, we benefit from its support for other languages (e.g., Java or C#). Second, we showcase Wasm's capability of program analysis by developing a set of static and dynamic analysis in PowerPoly. Specifically, we show that PowerPoly outperforms the state-of-the-art approaches through the design and implementation of a vulnerability detection algorithm to detect vulnerabilities in real-world programs. Third, we utilize the function table, a unique feature in Wasm to represent indirect function calls between different source languages, to eliminate language boundaries. Consequently, our approach is cost-effective, requiring no specialized approaches in analyzing language boundaries.

To validate our design, we implement a prototype for PowerPoly for Rust/C and Go/C program, due to three reasons. First, Rust and Go is an increasingly important secure language deployed in many security-critical scenarios. Second, Rust and Go can interact with other languages such as C/C++ through `unsafe` and cgo sub-language, thus can lead to serious vulnerabilities [28]. Third, existing efforts [28] for analyzing Rust/C programs are still limited in analyzing its

FFIs, and to the best of our knowledge, there have not been a tool used for Go/C program analysis yet. To this end, this work also represents a new step towards Rust and Go security study in its own right. However, it should be noted that PowerPoly is not tied to Rust/C and Go/C, but can process other language combinations as well (see § 6).

With this implementation, we conduct extensive experiments to evaluate it in terms of effectiveness, usefulness and performance, on a micro benchmark with 17 Rust/C and 17 Go/C multilingual programs as well as a real-world benchmark with 50 CWEs and 21 real projects. Our results demonstrate that PowerPoly effectively detects 7 kinds of vulnerabilities, outperforming FFIChecker and Govulncheck. Furthermore, PowerPoly is useful in protecting real-world projects, detecting 63 of 71 (88.7%) vulnerabilities in CWEs. Finally, PowerPoly brings acceptable overhead with file size increase of 10.2% and execution time increase of 26.4% on average, which is in line with prior studies [25].

Contributions. To the best of our knowledge, PowerPoly is a new step towards proposing a holistic framework for multilingual program analysis. In summary, this work makes the following contributions:

- We propose PowerPoly, the first framework for effective and holistic multilingual program analysis by leveraging Wasm.
- We design and implement a software prototype to validate our design.
- We conduct extensive experiments to evaluate the effectiveness and performance of PowerPoly on both micro benchmarks and real-world projects.

The rest of this paper is organized as follows. Section 2 introduces the background and the motivations and the threat model. Section 3 presents the design of PowerPoly. Section 5 presents the experiments to evaluate PowerPoly. Section 6 discusses limitations and directions for future work. Section 7 presents the related work, and Section 8 concludes.

2 Background and Motivation

To be self-contained, in this section, we present the background knowledge for this work (§ 2.1) and our motivation (§ 2.2), followed by challenges (§ 2.3).

2.1 Background

Multilingual programming. Multilingual programming refers to using multiple programming languages to develop program components and software systems. It makes it available for developers to combine the features and advantages of different languages, as well as reusing existing libraries. Hence, it has been widely used in many scenarios such as scientific computation[16] and deep learning[38]. In order to support seamlessly interoperation between different languages, multilingual programming introduced a mechanism called foreign function interface (FFI), to call external interfaces and connect different languages.

For example, Python supports Python/C API[17], and Java supports Java Native Interface (JNI)[37].

Wasm. Wasm is an emerging secure and portable instruction set architecture first released in 2017 for Web [54]. In 2018, the first complete formal definition of Wasm was released[56]. In 2019, Wasm was announced as the fourth official Web standard [48] and has also grown into a general-purpose language deployed in various domains, with the introduction of the Wasm System Interface (WASI) [35].

Wasm was designed with the aims of safety, efficiency, and portability. First, to guarantee program safety, Wasm incorporates diverse secure features such as strong typing [14], secure control flow [52], and linear memory [55]. Second, Wasm VMs enable Wasm programs to efficiently utilize hardware capabilities across different platforms with high efficiency. Third, Wasm has the design of WASI, making it convenient to deploy Wasm programs outside of browsers.

Due to its technical advantages, Wasm has been widely used in both web and non-web domains. In Web domain, Wasm has become the fourth official Web language development with full support by major browsers [47] [2]. In non-Web domains, Wasm is widely used in diverse scenarios such as cloud computing [15] [32] [1], IoT [30], and blockchain[7] [21] [4].

2.2 Motivation

Analyzing and vulnerability detecting for multilingual programs are difficult due to the discrepancies between languages as well as the complexity of FFI. Moreover, vulnerabilities of multilingual programs may exist at and across language boundaries, causing single-language analysis for each language fail to detect such vulnerabilities due to the lack of cross language information.

Motivating examples. To better illustrate our research motivation, we present a set of running examples to demonstrate how memory vulnerabilities manifest in multilingual programs.

As shown in Fig. 1, a Rust function calls a foreign function `c_func` defined in C program (❶). The `c_func` function calls `wrapper1` via a function pointer to free an object (❷). However, the Rust program is unaware that `n` has been deallocated in C function, and thus attempts to release it automatically after `n` goes out of scope (line R6), resulting in a double-free (DF) bug.

2.3 Challenges

Despite this security criticality and urgency [18] [29] [27] [19] [28] [44], to the best of our knowledge, unified multilingual programming analysis and vulnerability detection has not been thoroughly studied. Developing an effective and holistic framework for analyzing multilingual programs requires addressing several technical challenges.

C1: language boundaries and FFIs. As the running examples in Section 2.2 shows, vulnerabilities of multilingual programs can exist at and across language

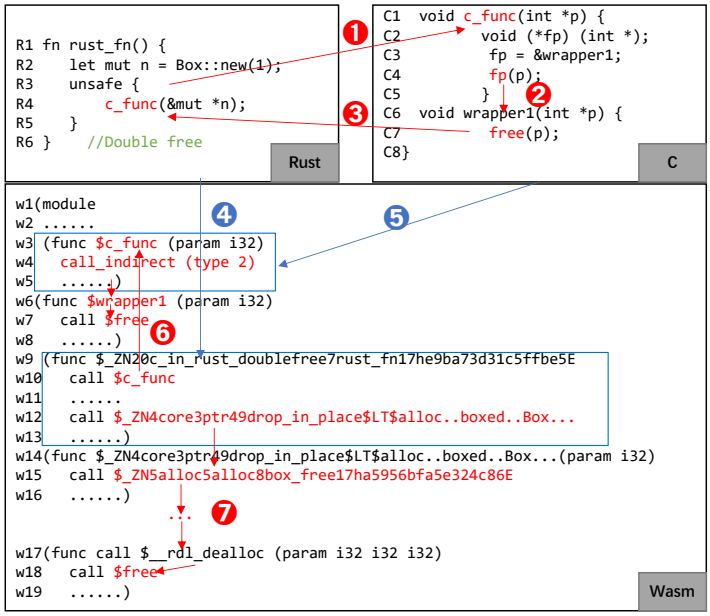


Fig. 1: Sample code illustrating a DF vulnerability across Rust and C.

boundaries, making the analysis across two languages difficult. Hence, developing a vulnerability detection for multilingual programs to detect vulnerabilities at or across language boundaries is challenging.

Solution: To address this challenge, we use inter-procedural analysis. Our approach is based on a key observation: most FFIs in sources programs are compiled to direct or in-direct function calls in Wasm. As a result, source-level FFIs are regarded as common function calls at the Wasm level, thus effectively eliminating language boundaries.

C2: lack of function call information. Wasm does not provide function call information required for the target analysis, due to two reasons. First, the function being called resides in a dynamically linked library that is absent during analysis. Second, the function being called is a function pointer, hindering precise function call analysis [24].

Solution: To address this challenge, we utilize dynamic analysis to analyze multilingual programs with instrumentation and user-customized plugins. Our selection of dynamic analysis enables us to record and analyze function information dynamically. Furthermore, to achieve more flexibility, we introduce a extensible framework into PowerPoly to allow developers design their own dynamic analysis via plugins.

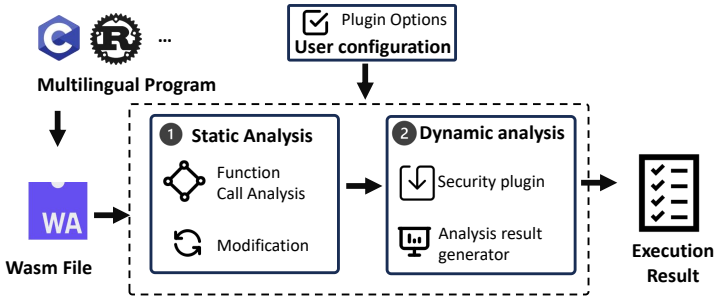


Fig. 2: An overview of PowerPoly.

3 Approach

In this section, we present the approach of PowerPoly, by first introducing the design goals and its overview (§ 3.1). Then we present the design of each component (§ 3.2 to § 3.3), respectively.

3.1 Design Goals and Overview

We have three goals guiding the design of PowerPoly. First, PowerPoly should provide complete comprehensive analysis for multilingual programs. Second, PowerPoly should be an automatic and end-to-end solution with minimal user interventions. Finally, PowerPoly should provide a user-friendly interface and result report for users to analysis problems.

With these design goals in mind, we present, in Fig. 2, the architecture of PowerPoly. It requires two inputs: 1) the Wasm binary compiled from multilingual program. 2) user configuration to specify which security plugins to apply. With these inputs, PowerPoly operates in two key phases: 1) the static analysis (❶), which takes the Wasm binary as input, collects foreign functions used in the programs and modifies the binary instructions to make it compatible with each other. 2) the dynamic analysis (❷), reads modified Wasm binary as input, then applies vulnerability detection using security plugins through static instrumentation, finally executes instrumented Wasm binary in Wasm VM and outputs analysis result by result generator.

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

3.2 Static Analysis

The static analysis takes a compiled Wasm binary as input, and outputs an modified Wasm binary by FFI information and modification. Next, we present the detailed design of each component, respectively.

Function call analysis. Function calls in Wasm binaries are compiled as `call` or `indirect_call` instructions. Since different functions invoked by these instructions require different handling, we need to categorize them into distinct

types: 1) Functions that allocate heap memory, such as `dlmalloc` compiled from Rustc and `malloc` compiled from TinyGo. 2) Functions that release heap memory, such as `dlfree` compiled from Rustc and `free` compiled from TinyGo.
Modification. Wasm binaries compiled from different compilers and language combinations may not always be compatible with each other. For example, while Rustc generates Wasm binaries in version 1.0, the Wasm binaries generated by TinyGo for Go/C are with version 2.0 [53]. Therefore, we replace these novel instructions.

3.3 Dynamic Analysis

PowerPoly performs dynamic analysis through instrumented Wasm binary execution. We first apply security plugins for corresponding vulnerabilities, and reports the vulnerabilities found as the outputs.
Security plugin. We mainly focus on vulnerability detection based on the static instrumentation in PowerPoly. Different user configuration can trigger different security plugins.

The instrumentation algorithms for Wasm binary is shown in Algorithm 1. If users want to detect integer overflow, we insert a set of Wasm instructions to get operands and call vulnerability detection function before and after each related instruction, respectively (line 2 to 5). Moreover, if users wish to detect double free, we modify the code of the memory release and memory allocation function (line 7 to 8) to track freed memory areas. As for UaF, we first adjust the code of the memory release function, then insert UaF detection function before memory access instructions (line 10 to 13). To detect memory leak, we modify the memory release and memory allocation functions to log memory allocation information, and detect the vulnerabilities after the main function terminates (line 15 to 17). Finally, for null pointer dereference (NPD) detection, we insert NPD detection function before each load instruction (line 19 to 21). Next, we present the detailed design of some kinds of vulnerabilities, respectively.
Integer overflow. An integer overflow (IO) refers to the overflow of the result of arithmetic operation. It could lead to buffer overflow if an overflowed value is used for memory allocation.

The conditions for IO detection of each operation are shown in Table 1[42]. In order to detect IO, the *RelatedInstr* contains the instructions in first column, and function *IODetectionFunc* validates the conditions in second column of Table 1. Specifically, take `i32.mul` as a showcase, the template of *IODetectionFunc* is shown in Fig. 3. We validate if $r \neq 0$ (line 3 to 5), then validate if $r/a \neq b$ (line 6 to 11). If an IO occurs, the Wasm binary terminated (line 12 to 14). Otherwise, the function returns r as normal (line 15 to 20).
Memory Corruption. Memory corruption contains a set of vulnerabilities that affect data stored in memory, such as double-free and buffer overflow. To illustrate our algorithms, we take double-free bug as the example.

Double free (DF) bug caused by releasing an already freed memory for a second time. The `DFInstrumentation` function modify the functions that allocate and release memory.

Algorithm 1: Static instrumentation.

```
   Input: 𝕄: a Wasm module
 1 Function IOInstrumentation(𝕄):
 2     for each instruction i in 𝕄 do
 3         if i ∈ RelatedInstr then
 4             append (GetOperands, i);
 5             append (i, call IODetectionFunc);
 6 Function DFInstrumentation(𝕄):
 7     insert_prefix($free, DFPrefix);
 8     insert_postfix($malloc, DFPostfix);
 9 Function UaFInstrumentation(𝕄):
10     insert_prefix($free, UaFPrefix);
11     for each instruction i in 𝕄 do
12         if i ∈ RelatedInstr then
13             append (call UaFDetectionFunc, i);
14 Function MLInstrumentation(𝕄):
15     insert_prefix($free, MLPrefix1);
16     insert_prefix($malloc, MLPrefix2);
17     insert_postfix($main, MLPostfix);
18 Function NPDInstrumentation(𝕄):
19     for each instruction i in 𝕄 do
20         if i == load then
21             append (call NPDDetectionFunc, i);
```

The Wasm code that inserted in the front of memory release function is shown in Fig. 3, with the key idea of dirty value [42]. First, after an area of memory is freed, we replace the value stored in base address with a dirty value, which represents a very large and rarely used integer value (line 8 to 10). Then for memory free, we check the value stored in the base address and report a DF vulnerability when the value is a dirty value (line 1 to 6). Moreover, we need to modify memory allocation function, in order to eliminate the potential dirty value stored in the base address if an area of memory is reallocated just after being released.

Analysis result generator. The analysis result generator takes instrumented Wasm binaries as the input, and outputs the analysis result by generating execution result of the instrumented Wasm binaries through Wasm VM.

After static instrumentation, if any vulnerabilities exist in the instrumented Wasm binaries, the `unreachable` instruction will be triggered, and the execution of Wasm binary will be terminated while executing by Wasm VM. In detail, we wrap the vulnerability detection functions, including original functions for SBO and HO as well as other functions we designed and extended, with corresponding function names, so that we could find out which vulnerability is occurred by analyzing the calling stack while `unreachable` instruction is triggered.

Table 1: Conditions of integer overflow.

Operations	Condition
$r = a +_s b$	$(a > 0 \wedge b > 0 \wedge r < 0) \vee (a < 0 \wedge b < 0 \wedge r > 0)$
$r = a -_s b$	$(a > 0 \wedge b < 0 \wedge r < 0) \vee (a < 0 \wedge b > 0 \wedge r > 0)$
$r = a *_s b$	$r \neq 0 \wedge r/a \neq b$
$r = a \ll b$	$r \gg b \neq a$

```
(func <IODetectionFunc> (param <a> i32)
(param <b> i32) (param <r> i32)(result i32)
local.get <r>
i32.const 0
i32.ne
if (result i32) ;; r != 0
  local.get <r>
  local.get <a>
  i32.div_s
  local.get <b>
  i32.ne
  if (result i32) ;; r != 0 && r/a != b
    local.get <r>
    unreachable ;; terminate the Wasm binary
  else
    local.get <r> ;; return r
  end
else
  local.get <r> ;; return r
end)
```

```
local.get 0 ;; get the freed address
i64.load
i64.const <DirtyValue>
i64.eq
if ;; the loaded value is dirty value
    unreachable ;; terminate the Wasm binary
else
  local.get 0
  i64.const <DirtyValue>
  i64.store ;; set the value as dirty value
end
```

Fig. 3: Template of *IODetectionFunc* of `i32.mul` and *DFPrefix*.

4 Implementation

To validate our design, we implement a software prototype for PowerPoly, specifically for Rust/C and Go/C programs with ten plugins each detecting one type of vulnerabilities. Next, we highlight some implementation details.

Static analysis. We compile Rust/C and Go/C programs to the Wasm binary by using the official rust compiler `rustc` [40] and Go compiler `TinyGo` [46], and leverage the foreign function collector module of FFIChecker [28], to collect the information of foreign functions in Rust/C programs.

Dynamic analysis. We implement the dynamic analysis part for security plugins and analysis result generator, respectively. First, we implement security plugins by porting and extending the instrumentation module of a popular Wasm fuzzing tool Fuzzm [25]. The extended algorithms consist of 1,689 lines of Rust code. Then, we implement analysis result generator by executing instrumented Wasm binaries in the Wasmtime [6] VM. We select Wasmtime as our Wasm VM for two reasons: 1) Wasmtime is a popular Wasm VM with 14.9k GitHub stars, more than other Wasm VMs such as wasm3 [49] and WAMR [5]. 2) Wasmtime is designed with high security, low overhead features, and WASI support. We then execute the instrumented Wasm binaries using Wasmtime.

5 Evaluation

To understand the effectiveness of PowerPoly, we evaluate it on micro benchmarks and real-world Rust/C and Go/C programs. Specifically, our evaluation aims to answer the following questions:
RQ1: Effectiveness. Since PowerPoly is designed to provide vulnerability detection, does it effectively detect bugs in multilingual Rust/C and Go/C programs?
RQ2: Usefulness. Is PowerPoly useful in detecting real-world vulnerabilities in real-world Rust/C and Go/C applications?
RQ3: Overhead. As a tool to provide static instrumentation and dynamic analysis to detect vulnerabilities, it will inevitably increase the code size and the analysis time. Therefore, is PowerPoly guaranteed to bring low overhead?
RQ4: Compare with existing studies. Does PowerPoly outperform existing Rust/C program analysis studies?

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 used two datasets to conduct the evaluation: 1) micro-benchmarks, containing a total of 34 vulnerable Rust/C and Go/C programs; and 2) real-world benchmarks, containing a total of 50 vulnerable programs from real-world CWEs and 21 real-world applications.
Micro-benchmark. We constructed a micro-benchmark consisting of 17 test cases in each language combination. Some of the vulnerabilities that in programs are selected from FFIChecker[28], including double free, use after free, and memory leak. Others are manually created since the limitations of FFIChecker. These test cases are collected for two reasons: 1) some test cases used in FFIChecker are suitable for our analysis since these programs are Rust/C programs with vulnerabilities across FFI; and 2) due to the limitations of FFIChecker to analysis functions in dynamically linked C libraries and function pointers, we manually conducted test cases contains these situations, as well as test cases that with IO vulnerabilities and in Go/C combination.
Real-world Benchmark. CWE [8] is a set of vulnerable programs written in C which contain various vulnerabilities such as buffer overflow and integer overflow. Conducting our PowerPoly on well-established vulnerability sets is an effective way to validate the usefulness of our framework. We added Rust and Go wrapper to each C code to turn them into Rust/C and Go/C programs.

Moreover, applying PowerPoly on real-world Rust/C and Go/C projects is an effective way to validate the usefulness of our framework. We selected real programs in each language combination followed three principles: 1) the projects should be open source. Therefore, we collected the projects from GitHub or from the open source of prior works. 2) the projects could be compiled to Wasm easily, so the projects we chose are those could be compiled to Wasm. 3) the projects should have a number of discovered memory vulnerabilities or be written

Table 2: Experimental results on real-world-benchmarks.

Dataset	Total	Success	Recall	F1
CWE(Rust)[8]	25	21	84%	91.3%
CWE(Go)[8]	25	21	84%	91.3%
Real(Rust) [28] [13]	11	11	100%	100%
Real(Go) [28]	10	10	100%	100%
Total	71	63	88.7%	94.0%

by memory-unsafe language in order to show the usefulness of PowerPoly. Finally, we selected 10 programs used from FFIChecker [28] and 1 projects from GitHub which been detected by FFIChecker as real Rust/C benchmark, and transform those 10 programs to Go/C as real Go/C benchmark.

5.2 Evaluation Metrics

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

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

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

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

F1 score can reflect the overall accuracy of analysis tools.

5.3 RQ1: Effectiveness

To answer RQ1, we first apply PowerPoly to micro-benchmarks. We first compiled these Rust/C and Go/C programs to Wasm binaries and applied instrumentation for them. Then, we applied PowerPoly for dynamic analysis to each case respectively.

The column 8 in Table 3 presents the experimental result. The experimental results demonstrate that 31 test cases in total are effectively detected after being instrumented by PowerPoly, and 3 test cases are failed to be detected. Consequently, the recall of PowerPoly is 91.2%, the precision is 100%, resulting in an F1 score of 95.4%, which illustrates that PowerPoly is effective in detecting various vulnerabilities in Rust/C and Go/C programs.

In order to find out the reasons for the failure to detect vulnerabilities for these test cases, we manually analyzed this test case and concluded the root causes. First, when the Rust/C programs attempt to multiply two i32 operands,

Table 3: Experimental results on micro-benchmarks.

Test Case	Type	Wasm LoC BI / AI	LoC Overhead	IT (s)	EXE Time / BI / AI (s)	EXE Time Overhead	PowerPoly / SOTA
1	DF_1	22649 / 22673	0.1%	0.003	0.012 / 0.015	25.0%	✔ / ✔
2	DF_2	22718 / 22742	0.1%	0.003	0.010 / 0.012	20.0%	✔ / ✘
3	IO_i32add	22666 / 31584	39.3%	0.006	0.011 / 0.012	9.1%	✔ / ✘
4	IO_i32mul	26721 / 27671	3.6%	0.005	0.011 / 0.013	18.2%	✘ / ✘
5	IO_i32shl	26715 / 27559	3.2%	0.004	0.009 / 0.011	22.2%	✔ / ✘
6	IO_i32sub	26707 / 28219	5.7%	0.004	0.014 / 0.016	14.3%	✔ / ✘
7	IO_i64add	26711 / 37418	40.1%	0.008	0.012 / 0.015	25.0%	✔ / ✘
8	IO_i64mul	26806 / 27798	3.7%	0.004	0.015 / 0.020	33.3%	✔ / ✘
9	IO_i64shl	26733 / 27577	3.2%	0.004	0.010 / 0.011	10.0%	✔ / ✘
10	IO_i64sub	26725 / 28243	5.7%	0.004	0.011 / 0.015	36.4%	✔ / ✘
11	ML	28908 / 28953	0.2%	0.004	0.010 / 0.013	30.0%	✔ / ✔
12	NPD	27252 / 32069	17.7%	0.005	0.016 / 0.022	37.5%	✔ / ✘
13	UAF_1	22692 / 30113	32.7%	0.005	0.009 / 0.013	44.4%	✔ / ✔
14	UAF_2	22761 / 30212	32.7%	0.005	0.009 / 0.012	33.3%	✔ / ✘
15	HO	23937 / 24141	0.9%	0.003	0.010 / 0.012	20.0%	✔ / ✘
16	SBO_1	30649 / 36280	18.4%	0.005	0.019 / 0.038	100.0%	✔ / ✘
17	SBO_2	30649 / 36280	18.4%	0.005	0.023 / 0.038	65.2%	✘ / ✘
1	DF_1	119398 / 119514	0.1%	0.007	0.032 / 0.032	0%	✔ / ✘
2	DF_2	119591 / 119707	0.1%	0.007	0.032 / 0.033	3.1%	✔ / ✘
3	IO_i32add	95707 / 106511	11.3%	0.006	0.028 / 0.032	14.3%	✔ / ✘
4	IO_i32mul	95707 / 96260	5.8%	0.006	0.026 / 0.028	7.7%	✔ / ✘
5	IO_i32shl	95707 / 96118	4.3%	0.006	0.026 / 0.028	7.7%	✔ / ✘
6	IO_i32sub	95732 / 97182	1.5%	0.007	0.027 / 0.028	3.7%	✔ / ✘
7	IO_i64add	95708 / 106512	11.3%	0.011	0.028 / 0.033	17.9%	✔ / ✘
8	IO_i64mul	95708 / 96261	5.8%	0.006	0.026 / 0.028	7.7%	✔ / ✘
9	IO_i64shl	95708 / 96119	4.3%	0.005	0.025 / 0.029	16%	✔ / ✘
10	IO_i64sub	95733 / 97183	1.5%	0.007	0.028 / 0.030	7.1%	✔ / ✘
11	ML	96046 / 96183	0.1%	0.006	0.029 / 0.030	3.4%	✔ / ✘
12	NPD	95730 / 108298	13.1%	0.016	0.027 / 0.030	11.1%	✔ / ✘
13	UAF_1	119428 / 149711	25.4%	0.030	0.032 / 0.039	21.9%	✔ / ✘
14	UAF_2	119621 / 149994	25.4%	0.028	0.032 / 0.040	25%	✔ / ✘
15	HO	96218 / 96519	0.3%	0.005	0.027 / 0.030	11.1%	✔ / ✘
16	SBO_1	119585 / 124661	4.2%	0.009	0.033 / 0.066	100.0%	✔ / ✘
17	SBO_2	119585 / 124661	4.2%	0.009	0.034 / 0.066	94.1%	✘ / ✘

LoC: Line of Code; BI: Before Instrumentation; AI: After Instrumentation; IT: Instrumentation Time; SOTA: FFIChecker and Govulncheck.

the generated Wasm binaries do not simply use an `i32.mul` instruction, but extend two operands to i64 then use `i64.mul` and `i32.wrap_i64` to obtain the result. Since we did not implement instrumentation for type conversion instructions, PowerPoly could not detect the overflow in this case, and thus fails to report, as the situation in test case 4 of Rust/C programs. Second, since SBO or HO detection function utilized canary insertion [25], when the buffer on the stack or heap overflows a few bytes and does not reach the canary, PowerPoly still considers this memory access as a legitimate one, thus fails to report, as the situation in test case 17 of both two language combinations.

5.4 RQ2: Usefulness

To answer RQ2, we apply PowerPoly to real-world benchmark. We first compiled each program to Wasm, then apply PowerPoly to the generated results. We recorded the vulnerabilities detected by PowerPoly in these programs, and compared them with their pre-annotated vulnerabilities, finally counted the number of vulnerabilities PowerPoly detected in real-world programs.

For the experimental results of this test set that Table 2 shows, out of the 71 vulnerabilities, PowerPoly successfully found 63 of them, while 8 were not detected. These results yield a recall of 88.7%, a precision of 100%, and an F1 score of 94.0%. This shows that when applying PowerPoly to real-world

programs, PowerPoly can still identify potential vulnerabilities in programs and is still useful in real-world programs.

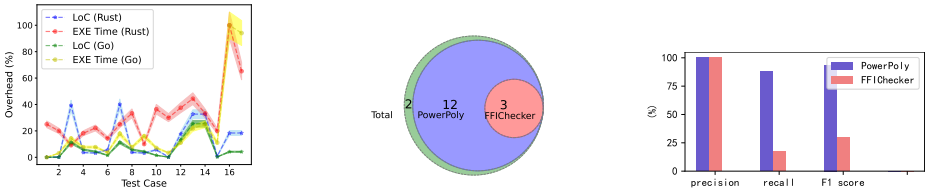
Furthermore, we investigate the root cause of the 8 failed cases. After manually inspecting these Wasm cases, we discovered that the reasons for the detection failure were similar to the reasons mentioned in Section 5.3, 2 test cases of them did not modify the canary due to insufficient stack buffer overflow bytes, and 6 test cases only read addresses beyond the buffer without writing, leaving the canary unchanged. As a result, PowerPoly failed to detect them. However, overall, the detection failure of these 8 vulnerabilities does not affect the usefulness of PowerPoly in real-world projects. These are not caused by the design defects of PowerPoly itself, but the limitations of the binary instrumentation and runtime detection technology used by PowerPoly.

5.5 RQ3: Overhead

To answer RQ3, we investigate the overhead of PowerPoly, including: 1) time spent on Wasm binary instrumentation; 2) increase in the code size of Wasm binary; and 3) execution time of Wasm binary. To this end, we first compiled the micro- benchmark to Wasm binaries and record the code size, then run each binary 20 rounds to calculate the average execution time, following prior work [23]. We then applied PowerPoly to generate instrumented Wasm binaries, then repeat the above process on each of these binaries. Finally, we calculated the changes in code size and execution time.

Columns 3 to 7 in table 3 presents the overhead that static instrumentation impose on Wasm binaries, where columns 3 and 4 represent the LoC(line of code) of the Wasm binary before and after static instrumentation as well as the overhead, column 5 represents the time spent on instrumentation, columns 6 and 7 represent the execution time of Wasm binary before and after instrumentation as well as its overhead.

The results showed that PowerPoly could instrument test cases in micro-benchmark for less than 0.03 second, that means that PowerPoly could complete static instrumentation effectively. Then, we compared the change in the LoC of generated Wasm binary before and after the instrumentation, and the results showed that the size of instrumented Wasm binary increased by 13.3% on average in Rust/C, which ranges from 0.1% to 40.1%, and increased by 7.0% on average in Go/C from 0.1% to 25.4%, thus did not cause an excessive increase in code size. The results showed that the execution time increases ranges from 9.1% to 100.0% in Rust/C, with an average of 32.0%, and it increases ranges from 0% to 100.0% in Go/C, with an average of 20.7%. In summary, we present the changes in file size and execution time introduced by PowerPoly in Fig. 4a. Compared with similar tools that use static instrumentation [25], the increase in code size and execution time caused by PowerPoly is also at a low level. Therefore, PowerPoly brings an acceptable code size overhead and execution time increase to the generated Wasm binaries.



(a) The file sizes and execution time changes introduced by PowerPoly.

(b) Bug detection capability.

(c) Evaluation metrics.

Fig. 4: A comparison of PowerPoly and state-of-the-art tool FFIChecker.

5.6 RQ4: Compare with other framework

To answer RQ4, we compare PowerPoly with the state-of-the-art Rust/C program analyzer FFIChecker [28] to evaluate their effectiveness on micro-benchmark. We ran both PowerPoly and FFIChecker on micro-benchmark respectively, and compared their execution results. To the best of our knowledge, there has not been a cross-language program analysis tool for Go/C yet, so we chose an official Go vulnerability checker Govulncheck [45] for comparison. As the result, Govulncheck could not find any cross-language vulnerabilities.

The experimental results are shown in last 2 columns of Table 3. PowerPoly successfully detect 15 cases of 17, containing 7 common vulnerabilities, while FFIChecker could only detect 3 kinds of these vulnerabilities, containing DF, UaF and ML. We also compare their detection capabilities and evaluation metrics in Fig. 4b. This shows that PowerPoly detects all the 3 vulnerabilities detected by FFIChecker. Moreover, PowerPoly achieves a significantly higher recall (88.2%) and F1 score (93.7%) than FFIChecker (17.6% and 29.9%) , demonstrating that PowerPoly could detect more kinds of vulnerabilities and provide more effective detection than FFIChecker, due to the limitations of its analysis algorithms which only focus on heap memory management issues and could not analyze C code from dynamically linked libraries and function pointers.

5.7 Case Study

To illustrate the ability of PowerPoly to provide vulnerability detection, we demonstrate how our approach can safeguard the vulnerability illustrated in Fig. 1.

PowerPoly eliminates the language boundaries by translating Rust/C program to one Wasm binary. In this case, `c_func` (line w3 to w5) and `rust_fn` (line w9 to w13) as well as other functions including functions in dynamically linked libraries are translated into one Wasm module (❹ and ❺), which makes multilingual program analysis same as single-language program.

In this Wasm binary, the object is freed in C function firstly (❻). The automatically released in Rust is compiled into a `drop_in_place` function (line w12), and the object is freed again in this function (❼). Since our algorithm inserts the code in Fig. 3 in the front of `free`, while the same address is attempted to be released, the `unreachable` instruction is triggered, thus a DF bug could be detected. Moreover, FFIChecker could not analyze this situation since it could not analyze the `wrapper1` function called by function pointer.

6 Discussion

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

Other language combinations. Since we focused on Rust/C and Go/C programs in the prototype, real-world multilingual systems may use various language combinations. Fortunately, adding support for other language combinations into current framework is convenient, since the analysis algorithms on Wasm are independent. Hence, adding support for other language combinations only requires the addition of translation to Wasm binaries without changing the analysis algorithms. We plan to study other language combinations such as Python/C and Java/C programs for future work.

Other vulnerabilities. Even though integer overflow and various common memory security vulnerabilities can be detected by PowerPoly effectively, there are other types of vulnerabilities. Specifically, it is important to investigate concurrency security such as race condition and deadlock. Specifically, we could extend PowerPoly to detect concurrency vulnerabilities by enhancing the support for signal and related algorithms of Wasm[51] [36].

7 Related Work

Existing studies relevant to this work can be boardly classified into two categories: the multilingual program security and Wasm security studies.

Multilingual program security. Many recent studies have focused on the security issues of different multilingual applications. Morrisett et al. [44] extend JVML to model the semantics of C to perform multilingual analysis across Java/C programs. Li et al. [28] design and implement a static analysis tool utilized LLVM to identify multilingual memory vulnerability in Rust/C programs. Jiang et al. [27] present PolyCruise, a dynamic analysis framework, for information flow analysis in python multilingual systems. However, these studies have their limitations. We implemented our analysis with the aid of Wasm, thus eliminating the language boundaries.

Wasm security study. There have been a lot of works related to improving the security of Wasm in the past years. Jiang et al.[20] propose WasmFuzzer to generate initial seeds for fuzzing at the Wasm bytecode level and design a systematic set of mutation operators for Wasm bytecode. Arteaga et al. [3] propose an

approach to achieve code diversification for Wasm, by generating multiple program variants from an input program. Daniel et al. [25] proposed Fuzzm, which protects the heap and stack insertion canaries in the Wasm linear memory area to achieve the protection of Wasm memory. However, these protections of Wasm are not thorough enough. We implement analysis by extending existing tools as well as various of vulnerability detection algorithms for certain vulnerabilities. Therefore, such an analysis mechanism has universality.

8 Conclusion

In this work, we present an effective approach for unified multilingual program analysis through WebAssembly. Our method leverages Wasm as a unified intermediate representation (IR), eliminating language boundaries by translating multilingual programs into Wasm. First, we design a static analysis to analyze function calls and modify Wasm binaries. Next, we design a dynamic analysis to detect vulnerabilities through security plugins. We implement a prototype called PowerPoly and conduct extensive experiments. Our evaluation results demonstrate that PowerPoly effectively provides program analysis for multilingual programs, with acceptable overhead. Overall, this work represents a new step towards multilingual program analysis, making multilingual programs safer by reducing vulnerabilities at and across language boundaries.

References

1. WebAssembly on Cloudflare Workers (Oct 2018), http://blog.cloudflare.com/webassembly-on-cloudflare-workers/
2. Apple: Safari (2024), https://www.apple.com/safari/
3. Arteaga, J.C., Malivitsis, O., Pérez, O.V., Baudry, B., Monperrus, M.: CROW: Code Diversification for WebAssembly. In: Proceedings 2021 Workshop on Measurements, Attacks, and Defenses for the Web (2021). https://doi.org/10.14722/madweb.2021.23004
4. Bian, W., Meng, W., Wang, Y.: Poster: Detecting webassembly-based cryptocurrency mining. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 2685–2687 (Nov 2019). https://doi.org/10.1145/3319535.3363287
5. Bytecodealliance: Wasm-micro-runtime: Webassembly micro runtime (wamr) (2024), https://github.com/bytecodealliance/wasm-micro-runtime
6. Bytecodealliance: Wasmtime: A standalone runtime for WebAssembly (2024), https://github.com/bytecodealliance/wasmtime
7. Chen, W., Sun, Z., Wang, H., Luo, X., Cai, H., Wu, L.: WASAI: Uncovering vulnerabilities in Wasm smart contracts. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 703–715. ACM, Virtual South Korea (Jul 2022). https://doi.org/10.1145/3533767.3534218
8. CWE: CWE-658:Weaknesses in Software Written in C (4.14) (2024), https://cwe.mitre.org/data/definitions/658.html
9. Emscripten-Core: Emscripten: Emscripten: An LLVM-to-WebAssembly Compiler (2024), https://github.com/emscripten-core/emscripten

10. Evans, A.N., Campbell, B., Soffa, M.L.: Is rust used safely by software developers? In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 246–257 (Jun 2020). https://doi.org/10.1145/3377811.3380413
11. Firefox: Language details of the Firefox repo (2024), https://4e6.github.io/firefox-lang-stats/
12. Furr, M., Foster, J.S.: Checking type safety of foreign function calls. ACM Transactions on Programming Languages and Systems **30**(4), 18:1–18:63 (Aug 2008). https://doi.org/10.1145/1377492.1377493
13. gchers: Emd - a simple Rust library for computing the Earth Mover's Distance (2024), https://github.com/gchers/rust-emd
14. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 185–200. ACM, Barcelona Spain (Jun 2017). https://doi.org/10.1145/3062341.3062363
15. Hall, A., Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. pp. 225–236. IoTDI '19, Association for Computing Machinery, New York, NY, USA (Apr 2019). https://doi.org/10.1145/3302505.3310084
16. Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with numpy. Nature **585**(7825), 357–362 (Sep 2020). https://doi.org/10.1038/s41586-020-2649-2
17. Hu, M., Zhang, Y.: The python/c api: Evolution, usage statistics, and bug patterns. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 532–536 (Feb 2020). https://doi.org/10.1109/SANER48275.2020.9054835
18. Hu, M., Zhao, Q., Zhang, Y., Xiong, Y.: Cross-Language Call Graph Construction Supporting Different Host Languages. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 155–166. IEEE, Taipa, Macao (Mar 2023). https://doi.org/10.1109/SANER56733.2023.00024
19. Hu, S., Hua, B., Xia, L., Wang, Y.: CRUST: Towards a Unified Cross-Language Program Analysis Framework for Rust. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS). pp. 970–981. IEEE, Guangzhou, China (Dec 2022). https://doi.org/10.1109/QRS57517.2022.00101
20. Jiang, B., Li, Z., Huang, Y., Zhang, Z., Chan, W.K.: WasmFuzzer: A Fuzzer for WasAssembly Virtual Machines. In: The 34th International Conference on Software Engineering and Knowledge Engineering. pp. 537–542 (Jul 2022). https://doi.org/10.18293/SEKE2022-165
21. Kelton, C., Balasubramanian, A., Raghavendra, R., Srivatsa, M.: Browser-Based Deep Behavioral Detection of Web Cryptomining with CoinSpy. In: Proceedings 2020 Workshop on Measurements, Attacks, and Defenses for the Web. Internet Society, San Diego, CA (2020). https://doi.org/10.14722/madweb.2020.23002
22. Kochhar, P.S., Wijedasa, D., Lo, D.: A Large Scale Study of Multiple Programming Languages and Code Quality. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). pp. 563–573. IEEE, Suita (Mar 2016). https://doi.org/10.1109/SANER.2016.112

23. Lehmann, D., Pradel, M.: Wasabi: A Framework for Dynamically Analyzing WebAssembly (arXiv:1808.10652) (Aug 2018)
24. Lehmann, D., Thalakottur, M., Tip, F., Pradel, M.: 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. pp. 892–903. ACM, Seattle WA USA (Jul 2023). https://doi.org/10.1145/3597926.3598104
25. Lehmann, D., Torp, M.T., Pradel, M.: Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly (arXiv:2110.15433) (Oct 2021)
26. Li, S., Tan, G.: Finding bugs in exceptional situations of jni programs. In: Proceedings of the 16th ACM Conference on Computer and Communications Security. pp. 442–452. CCS '09, Association for Computing Machinery, New York, NY, USA (Nov 2009). https://doi.org/10.1145/1653662.1653716
27. Li, W., Ming, J., Luo, X., Cai, H.: Polycruise: A cross-language dynamic information flow analysis. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2513–2530 (2022)
28. Li, Z., Wang, J., Sun, M., Lui, J.C.S.: Detecting Cross-language Memory Management Issues in Rust. In: Atluri, V., Di Pietro, R., Jensen, C.D., Meng, W. (eds.) Computer Security – ESORICS 2022, vol. 13556, pp. 680–700. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-17143-7_33
29. Li, Z., Wang, J., Sun, M., Lui, J.C.: MirChecker: Detecting Bugs in Rust Programs via Static Analysis. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 2183–2196. ACM, Virtual Event Republic of Korea (Nov 2021). https://doi.org/10.1145/3460120.3484541
30. Liu, R., Garcia, L., Srivastava, M.: Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices p. 12 (2021)
31. LLVM: The LLVM Compiler Infrastructure Project (2024), https://llvm.org/
32. Lucet: Lucet Takes WebAssembly Beyond the Browser | Fastly (2024), https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime
33. Mao, J., Chen, Y., Xiao, Q., Shi, Y.: Rid: Finding reference count bugs with inconsistent path pair checking. ACM SIGARCH Computer Architecture News **44**(2), 531–544 (Mar 2016). https://doi.org/10.1145/2980024.2872389
34. Mergendahl, S., Burow, N., Okhravi, H.: Cross-language attacks. In: Proceedings 2022 Network and Distributed System Security Symposium (2022). https://doi.org/10.14722/ndss.2022.24078
35. Mozilla: Standardizing wasi: A system interface to run webassembly outside the web (2024), https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface
36. Ning, P., Qin, B.: Stuck-me-not: A deadlock detector on blockchain software in rust. Procedia Computer Science **177**, 599–604 (2020). https://doi.org/10.1016/j.procs.2020.10.085
37. Oracle: Java native interface specification contents (2024), https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html
38. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems. vol. 32. Curran Associates, Inc. (2019)

39. Qin, B., Chen, Y., Yu, Z., Song, L., Zhang, Y.: Understanding memory and thread safety practices and issues in real-world rust programs. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 763–779 (Jun 2020). https://doi.org/10.1145/3385412.3386036
40. Rust: What is rustc? - the rustc book (2024), https://doc.rust-lang.org/rustc/what-is-rustc.html
41. Rustc: Mir (2024), https://rustc-dev-guide.rust-lang.org/mir/index.html
42. Sun, H., Zhang, X., Su, C., Zeng, Q.: 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. p. 483–494. ASIA CCS '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2714576.2714605, https://doi.org/10.1145/2714576.2714605
43. Tan, G., Croft, J.: An empirical security study of the native code in the jdk. In: Proceedings of the 17th Conference on Security Symposium. pp. 365–377. SS'08, USENIX Association, USA (Jul 2008)
44. Tan, G., Morrisett, G.: Ilea: Inter-language analysis across java and c. ACM SIGPLAN Notices **42**(10), 39–56 (Oct 2007). https://doi.org/10.1145/1297105.1297031
45. Team, G.: Govulncheck v1.0.0 is released! (2024), https://go.dev/blog/govulncheck
46. TinyGo-Org: Tinygo - go compiler for small places (2024), https://github.com/tinygo-org/tinygo
47. V8: V8 JavaScript engine (2024), https://v8.dev/
48. W3: Webassembly becomes a w3c recommendation (2024), https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en
49. wasm3: wasm3: A fast WebAssembly interpreter and the most universal WASM runtime. (2024), https://github.com/wasm3/wasm3
50. wasmerio: py2wasm (2024), https://github.com/wasmerio/py2wasm
51. Watt, C., Rossberg, A., Pichon-Pharabod, J.: Weakening webassembly. Proceedings of the ACM on Programming Languages **3**(OOPSLA), 1–28 (Oct 2019). https://doi.org/10.1145/3360559
52. WebAssembly: Execution — WebAssembly 2.0 (Draft 2024-04-28) (2024), https://webassembly.github.io/spec/core/exec/index.html
53. WebAssembly: Index of Instructions — WebAssembly 2.0 (Draft 2024-04-28) (2024), https://webassembly.github.io/spec/core/appendix/index-instructions.html
54. WebAssembly: Roadmap - webassembly (2024), https://webassembly.org/roadmap/
55. WebAssembly: Structure — WebAssembly 2.0 (Draft 2024-04-28) (2024), https://webassembly.github.io/spec/core/syntax/index.html
56. WebAssembly: Webassembly core specification (2024), https://www.w3.org/TR/wasm-core-1/