Effective Call Graph Construction for Multilingual Programs

Wenlong Zheng 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
zwl21@mail.ustc.edu.cn bjhua@ustc.edu.cn*

***Abstract*—Multilingual programming is increasingly prevalent in modern software systems due to its capability to leverage the diverse and complementary features of different programming languages. However, while multilingual programming offers the technical advantages of harnessing the strengths and attributions of different languages, multilingual programs are prone to bugs and can further lead to security vulnerabilities, due to the semantics discrepancies between languages and the lacking of a holistic static analysis across language boundaries.**

In this paper, to fill the gap, we proposed PolyCall, a holistic program analysis framework to construct call graphs for multilingual programs, as call graph construction is one of the most typical and important inter-procedural program analysis which can further benefit the security enhancement and optimization. Our key insight for this work is to utilize WebAssembly, a recently proposed common program binary format for *execution*, as our intermediate representation for *analysis*. By translating both the host and guest languages in multilingual programs into this unified intermediate representation, we successfully eliminate both the language boundaries and the semantic discrepancies, demonstrating the capability of our approach for holistic program analysis. We have implemented a prototype for PolyCall and have conducted extensive experiments to evaluate it in terms of effectiveness and performance. Experimental results demonstrated that PolyCall can effectively construct call graphs for multilingual programs outperforming peer tools, with an acceptable analysis time of 0.01 millisecond per line approximately.

***Index Terms*—Multilingual Programming, Program analysis, Call graph**

I. Introduction

Multilingual programming is increasingly important in building real-world software systems, which allows developers to leverage the strengths of different languages. For example, a large spectrum of software systems in deep learning (*e.g.*, PyTouch [1], TensorFlow [2], Pillow [3], and NumPy [4]) are multilingual programs containing about 50% C/C++ code (for backends) and 40% Python code (for programming interfaces). Given the fact that multilingual software systems are playing a critical role in today's digital world, they should be secure and trustworthy.

Despite this security criticality, recent studies [5] [6] [7] have demonstrated that multilingual systems are error-prone and exploitable, due to two root causes: first, language feature discrepancies such as memory management [8], exception handling [9], and type system [10], make it difficult to analyze multilingual programs in a holistic framework. Second, the vulnerabilities in multilingual programs often arise between the language boundaries, rather than with a single language. As such, there is a pressing need to develop a unified analysis for multilingual programs.

Recognizing this urgent need, a large spectrum of studies have been conducted [11] [12] [13] [6] [14] for multilingual analysis. While prior studies have made considerable progress, they, unfortunately, still suffer from two limitations: first, existing intermediate representations for multilingual analysis are confined to specific language combinations thus lack generality. For example, prior studies such as MIRChecker [13] utilized MIR, an intermediate representation specific to Rust, to analyze Rust/C multilingual programs; while ILEA [15] proposed a specification language that extended JVML, an intermediate representation based on Java bytecode to module the Java and C. It remains unclear how to scale these intermediate representations to support other language combinations such as Python/C [6] or Go/C [16] .

Second, although prior program analysis are successful in analyzing single-language programs for diverse tasks (*e.g.*, fault localization [17], [18], and security scanning [19]), they are ineffective in analyzing multilingual programs due to the language disparity. As a result, heuristics have to be employed which are not only coarse-grain but also heavy-weighted. For example, PolyCruise [12], a multilingual program analysis for Python/C, utilized a hybrid two-layered approach which consists of semantics-independent language specific analyses with a language-agnostic dynamic analysis. Hence, the language specific analysis must be reimplemented for each language combination, while language-agnostic must be implemented with heuristics which might lead to high false positives and negatives [20].

Recognizing this security criticality and technical challenges, we argue that, effective intermediate representations and program analysis for multilingual programs should satisfy the following three requirements: **R1** *rich language support*: it should provide extensive support for diverse programming languages and be capable of handling the cross-language interoperations. **R2** *rigorous semantics*: the program analysis for multilingual programs should employ intermediate representations with rigorous and well-defined semantics to precisely

* Corresponding authors.

capture the programs behaviors and ensure the reliability of analysis. **R3** *uniform*: It is crucial to provide a uniform representation and analysis framework that abstracts away the boundaries and disparities between programming languages.

In this paper, we propose using WebAssembly (Wasm), an emerging portable instruction set allowing for safe program *execution* with near-native performance, as a unified platform for *analysis* to conduct program analysis for multilingual programs. Specifically, we present PolyCall, a unified analysis framework based on Wasm, for the call graph construction across language boundaries. We focus on the call graph construction analysis for two key reasons: first, it is one of the most typical inter-procedure program analysis to depict the connectivity between functions and reason about dependencies and flows across boundaries. Second, the identification of function-level properties helps uncover potential issues and optimization opportunities, enhancing both the security and performance of programs.

We argue that our proposal and design satisfies the aforementioned three requirements for multilingual program analysis: 1) Wasm has a rich language support. Many programming languages have officially supported or begun to support the compilation to Wasm: Rust [21] [22], C/C++ [23], have fully supported the compilation from them to Wasm, and Javascript [24], Python [25] and Go [26] have partially support compiling some features to Wasm. 2) Wasm is designed with promising security mechanisms. For example, Wasm incorporates a strong type system [27] to guarantee type safety, and applies intra-process lightweight sandboxing [28] to provide an isolated execution environment. Furthermore, Wasm linear memory module [29] can effectively mitigates common memory attacks such as return-oriented programming (ROP) [30] based attack. 3) Wasm provides a unified analysis platform for multilingual programs. Wasm is a portable instruction set architecture that offers a unified execution platform for support languages, hence, it is reasonable and actionable to utilize Wasm as a unified cross-language analysis platform to eliminate the boundary in multiligual programs. In a nutshell, Wasm is an ideal unified language set for multilingual programming analysis. On top of that, the PolyCall consists of 3 main components: 1) a unified and low-level language Wasm with a rigorously strong type system; 2) a transformation from multilingual programs to Wasm; and 3) a call graph construction on Wasm to connect and module the interoperation between the host and guest languages.

We have conducted extensive experiments to evaluate the effectiveness and performance of PolyCall. First, to evaluate the effectiveness, we testify PolyCall on a dataset we have created with multilingual programs of Rust/C and Python/C, respectively. Experimental results demonstrated that PolyCall can effectively construct call graphs for multilingual programs in achieving at most a 15.26% increase in edges and an 18.40% increase in node, outperforming existing analysis tools. Furthermore, PolyCall is efficient by incurring a analysis overhead of 0.01 milliseconds per line approximately.

Contributions. To the best of our knowledge, this work is the *first* systematic study of call graph construction of multilingual programs via a unified analysis platform by leveraging Wasm. We shed light on the promising potentials of Wasm for program analysis, despite its initial design goal as an execution platform. To summarize, our work makes the following contributions:

- **A proposal of a unified program analysis framework for multilingual programs.** We showcase its capability by presenting PolyCall, the first unified call graph construction working across language boundaries.
- **Prototype implementation of PolyCall.** We implemented a prototype of PolyCall to validate our system design.
- **Evaluation of PolyCall.** We conducted extensive experiments to evaluate the effectiveness and performance of PolyCall.
- **Open source.** We make our software prototype, datasets and evaluation results publicly available in the interest of open science at:
  https://doi.org/10.5281/zenodo.8146194

Outline. The rest of this paper is organized as follows. Section II introduces the background knowledge. Section III presents the motivation for this work. Section IV describes the approach, and Section V presents the evaluations we conducted. Section VI discusses limitations and future work. Section VII describes related work, and Section VIII concludes.

II. Background

In this section, we first present the necessary background knowledge for this work: static program analysis and call graph (§ II-A), multilingual programming (§ II-B), and WebAssembly (§ II-C).

A. Static Program Analysis and Call Graph

Static program analysis. Static program analysis refer to automated techniques to analyze properties of software or systems without running them. Static analysis has been widely used in many scenarios (*e.g.*, bug detection [13] [31] [32], or program optimization [33]), due to its technical advantages: first, the static analysis offers the advantage of automation, significantly reducing the need for manual efforts and interventions. Second, unlike dynamic analysis which tracks and records program properties during runtime, static analysis normally analyzes programs without executing it, hence introducing no overhead. Third, since static analysis can be deployed before program execution, bugs or vulnerabilities can be detected at an early development stage.

Call graph. Call graph is an important data structure for static program analysis, with functions as graph nodes and calling relationship as graph edges. Call graph construction has been used in various program analysis tasks, such as fault localization [17] [18], security scanning [19], malware detection [34], among others [35] [36]. As call graph construction is an intrinsically inter-procedural program analysis enabling information

propagation across function boundaries, its completeness is essential to improve the accuracy of analysis results.

B. Multilingual Programming

Multilingual programming consists of program components developed by multiple programming languages, making it possible to take advantage of the strengths of diverse languages, or to reuse existing libraries or legacy code. Due to its technical advantages of programming flexibility and performance gains, multilingual programming has been widely used in many scenarios such as Web [37], deep learning [1], and scientific computation [4]. To support seamlessly interoperation between different languages, multilingual programming introduced foreign function interfaces (FFIs), in diverse syntax and terminologies. For example, Java supports Java Native Interface (JNI) [38], Rust supports FFI [39], while Python supports Python API [40]. Despite this syntactic disparity, these FFIs take similar functionalities such as type conversion, data layout organization, and (often optional) security checking.

C. WebAssembly

Brief history. Wasm was introduced by Google and Mozilla in 2015 [41] and quickly gained popularity, becoming a de facto standard language in browsers by 2017 [42]. In 2018, the first complete formal definition of Wasm was released [43], solidifying its specification. The W3C officially recognized Wasm as the fourth Web standard in 2019 [44], after HTML, CSS, and JavaScript. Over time, Wasm has evolved and matured, with the development of the WebAssembly System Interface (WASI) [45] and the ongoing work on the standard version 2.0 draft [43]. Today, Wasm is a stable and production-quality language that finds applications in both Web and standalone environments.

Features. Wasm is designed with a focus on safety, efficiency, and portability [46]. First, to ensure program safety, Wasm incorporates secure features such as strong typing, sandboxing isolation, and control flow integrity [27] [28]. Second, Wasm's virtual machine (VM) is optimized for space usage and execution performance, allowing it to leverage hardware capabilities effectively across different platforms. Third, WASI provides a standardized and safe system interaction interface, enabling Wasm programs to be deployed not only in browsers, but also in desktops and clouds.

Applications. Due to its advanced features and technical advantages, Wasm is gaining widespread adoption in both Web and non-Web domains. In the Web domain, Wasm became the fourth official Web language with full support by major browsers. In non-Web domains, Wasm has been adopted in a wide range of computing scenarios, such as cloud computing [47] [48], IoT [49], blockchain [50] [51] [52] [53], edge computing [54], video transcoder [55], and game engines [56]. In the future, the growing desire to secure cloud and edge computing infrastructures without sacrificing efficiency will make Wasm a even more promising language.

```
Go
G1  import "C"
G2  func main() {
G3    str := "From Golang"
G4    // convert Go string to C style
G5    cStr := C.CString(str)
G6    // invoke alloc defined in C side
G7    C.alloc(cStr, len(str))
G8    // invoke formatPrint defined in C side
G9    C.formatPrint()
G10   // release resource using API provided by C
G11   C.release()
G12 }
```

```
C
C1  char* a = NULL;
C2  void alloc(char* str, int n) {
C3    a = (char*)malloc(sizeof(char)*(n+1));
C4    strcpy(a, str);
C5  }
C6  void formatPrint() {
C7    printf("str is %s \n", a);
C8    free(a);
C9  }
C10 void release() {
C11   free(a);
C12 }
```

Fig. 1: Sample code illustrating a double-free vulnerability across Go and C.

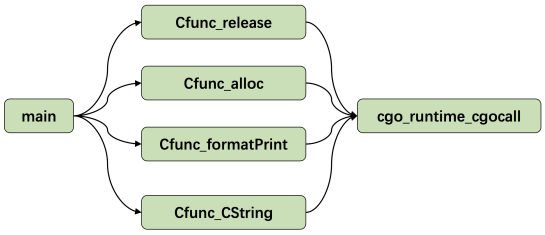

Fig. 2: The call graph generated by Go-callvis [57], for the Go/C program in Fig. 1, which does not contain graph edges for function calls across language boundaries between Go and C, leading to a false negative to detect the double-free bug.

III. Motivation

Multilingual programs are intrinsically difficult to analyze, due to the complexity of foreign function interfaces and disparities between language features. To put the discussion in perspective, we present, in this section, the motivation for our work by introducing bugs manifesting in multilingual programs as well as difficulties to analyze them.

Double-free bugs in Go/C programs. Double-free refers to a memory-safety issue that a memory pose security hazards to multilingual programs. Fig. 1 illustrates a double-free bug in Go/C multilingual programs. The Go function `main` initializes a string object `cStr` (line G5) and passes it to the `alloc` function in C (line C2), through Go's FFI `C.alloc` (line G7). The C code allocates heap space for the array `a` and fills it with the string `str` (line C3 to C4). Unfortunately, a double-free vulnerability is triggered as the the array `a` is released twice: the first one is at line G9 by invoking the C function `formatPrint` (line C6), and the second one is at line G11 by invoking the C function `release` at line C10.

This sample code not only illustrates vulnerabilities in Go/C multilingual programs but also justifies the holistic program analysis as any single-language analysis may miss such bugs due to their limited capability. To better illustrate the limitations of current analysis and tools, we present, in Fig. 2, the call graph for the program in Fig. 1 constructed by a state-of-the-art analysis tools, Go-callvis [57], dedicated for Go. The call graph in Fig. 2 generated by Go-callvis does not contain edges for function calls across the language boundary between Go and C, as a result, the analysis failed to detect the double-free bug.

Our tool PolyCall is capable of detecting cross-language vulnerabilities. Specifically, by compiling both the Go and C programs in Fig. 1 into Wasm, PolyCall uniformly processes functions from the host side and guest side. In particular, PolyCall can eliminate the language boundaries (*i.e.*, the Go FFI) by placing Wasm function declarations and definitions into a single Wasm module. As a result, PolyCall can construct a complete call graph for such multilingual programs, with graph edges from the `main` function to the `formatPrint` and `release` function, respectively. With this complete call graph, memory vulnerabilities such as the double-free can be detected without difficulties.

Buffer overflow and use-after-free bugs in Rust/C programs. A buffer overflow bug is caused by memory accesses out of the desired boundary, whereas a use-after-free is caused by reusing already reclaimed objects. Fig. 3 presents sample code illustrating these two kinds of bugs in Rust/C multilingual programs. The Rust function allocates a heap object `heap_obj` (line R2), then passes it to the C function `foo` (line R5). The C function `foo` might trigger a buffer overflow bug (line C6), as it has no knowledge of the length of the incoming vector. Furthermore, the Rust code might trigger a use-after-free bug (line R8), as the heap object `heap_obj` has been reclaimed by the C code (line C8). However, the Rust side has no idea of what happened from the C side and then continually operates on the released memory, leading to a use-after-free. The bugs in this sample code is subtle, as not only the C function violates the ownership security mechanism [58] provided by Rust but also the vector length in Rust cannot be propagated to C by inter-procedural constant propagation algorithms, due to the lack of a complete call graph.

Our tool PolyCall is capable of constructing the complete call graph that explicitly depicts the call chain. Specifically, in this case, the call chain `main`→`foo`→`free` gives necessary information to detect such bugs. Furthermore, inter-procedural constant propagation leveraging this call chain helps to detect the buffer overflow bug, by propagating the constant of vector length `3` of `heap_obj` from the caller `main` to the callee `foo`.

Before closing the discussion of the motivation for this work, it is should be noted that although we have focused on two specific language combinations (*i.e.*, Go/C and Rust/C) due to space limit, the problem of call graph construction analysis also manifests in other multilingual programs as well (*e.g.*, Python/C and Javascript/C [6]). In these scenarios, PolyCall remains to be effective due to the intrinsic similarity of the problem despite of different languages.

IV. Approach

In this section, we present our approach to conducting the study, by introducing the overall architecture of PolyCall (§ IV-A), followed by each component, including language selection and compilation (§ IV-B), call graph construction (§ IV-C), call graph construction on other static analysis tools (§ IV-D), and completeness analysis (§ IV-E).

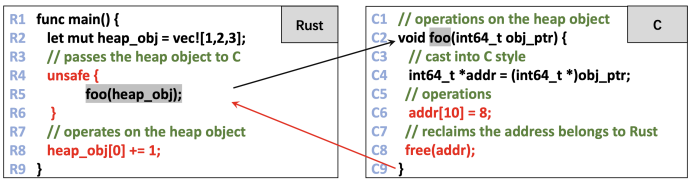

Fig. 3: Sample code illustrating a use-after-free bug, due to the mishandling the object from Rust to C.

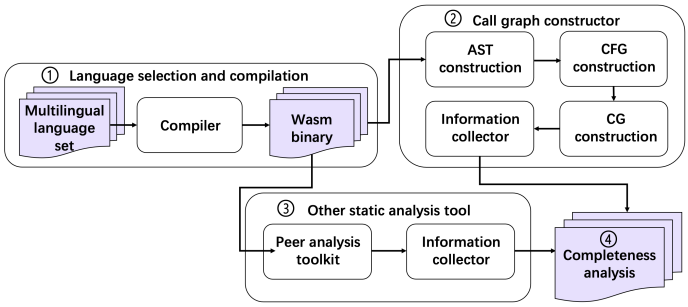

Fig. 4: The workflow of call graph construction for multilingual programs.

A. Architecture

We present, in Fig. 4, the overall architecture of PolyCall for call graph construction analysis of multilingual programs. First, we select the multilingual dataset developed with different languages then compile them to Wasm (①). Second, we conduct call graph construction on Wasm and generate an informative report for each call graph (②). Third, we apply our dataset to other static analysis tool and collect the CG information (③), and compare the completeness of CGs (④).

B. Language Selection and Compilation

For multilingual analysis, we select two mainstream programming languages that support cross-language interoperation with C/C++, and can be compiled to Wasm to some degree, that is, our language set includes Rust which fully support compilation to Wasm, and Go that basically support Wasm with some features in the process. Each multilingual program is then compiled to Wasm. Since there is no unified compiler that supports our language set, we pick out two outstanding compilers for multilingual programs respectively. More specifically, for Rust, we choose its official compiler, *i.e.*, rustc [59], with support compilation to Wasm, while for Go we selected TinyGo [60], an actively maintained toolchain that is intended for use in small places such as microcontrollers and Wasm, with 13.1k stars, following a criterion used in prior work [12], which indicates popularity. In a nutshell, we select the official compiler as well as the popular and well-maintained compiler for the sake of reliability.

C. Call Graph Construction

The key step of our work is the call graph construction for multilingual programs. We propose using the binary instruction set, Wasm, as the unified analysis platform. The

construction module takes in the Wasm programs, constructs the call graphs, and outputs the graphs information. To be more specific, the call graph construction consists of the following steps: 1) constructing the abstract tree (AST) of Wasm to obtain the structure of the program; 2) the control flow graph (CFG) construction of Wasm based on AST; 3) the call graph (CG) construction on CFG; 4) the informative report generation for the edges and nodes information of the CG.

D. Peer Static Analysis Tools

We apply our dataset to other popular static analysis tools for comparison. We select static tools according to three criteria: 1) active maintenance; 2) popularity; and 3) stability. First, we focused on toolkits that are still well maintained with an active community or recent commit to the project. Second, we focus on popular toolkits by selecting the one with more than 1,000 stars, following a criterion used in prior work [12], which indicates popularity. Third, we select projects with at least 10 releases that indicates an enhancement and milestones for new features and bugs solving.

E. Completeness Analysis

To better illustrate the completeness of the call graph construction, we utilize the information collected during the construction. We adopt two indicators: 1) the number of edges in CG; and 2) the number of nodes in CG. If the nodes and edges are more than CG constructed by peer tools, it indicates the high-completeness of the CG. To guarantee the correctness of the CG, we manually inspects the results to make sure the surge of the edges and nodes are contributing to the construction and connection with guest sides CG.

V. Evaluation

In this section, we present experiments to evaluate PolyCall. We first present the research questions guiding the experiments (§ V-A), the benchmark we created (§ V-C), and the experimental results in terms of the effectiveness, usefulness, and performance (§ V-E to § V-G). We then present the case study of the call graph construction in real-world multilingual programs (§ V-H).

A. Research Questions

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

RQ1: Effectiveness. As PolyCall is proposed for call graph construction, is it effective in constructing call graphs in multilingual programs?

RQ2: Usefulness. As PolyCall makes of the Wasm as the unified construction platform, how does PolyCall perform in real-world multilingual programs?

RQ3: Performance. As PolyCall makes of the static program analysis, what is its performance in call graph construction?

B. Experimental Setup

All experiments and measurements are performed on a server with two 8 physical Intel Xeon Silver 4110 core CPUs and 4 GB of RAM running Ubuntu 22.04.

C. Datasets

We use two datasets to conduct the evaluation: 1) micro-benchmarks; and 2) real-world benchmarks, containing a total of 34 (14+20) test cases. The concrete code cases are available in our open source.

Micro-benchmark. Evaluating the effectiveness of PolyCall needs a benchmark suit that comprises multilingual language programs from two different programming languages. However, such a benchmark is currently missing (for the best of our knowledge) and curating for large real world systems may not be feasible. Hence, we decide to take the first step to constructing the **MultiBench** manually, a micro-benchmark encompassing both Go/C and Rust/C multilingual programs, as presented in Table I. We manually construct 14 test cases with 7 basic cross-language programs written in the Go version and the Rust version, respectively (as presented by the 2nd column).

Currently, while **MultiBench** consists of two different programming languages to manifest the applicability of PolyCall, we will maintain and augment it by incorporating more benchmarks and test cases and covering other multilingual programming languages.

Real-world benchmark. In addition to the micro-benchmark, we also evaluate PolyCall against 20 real-world multilingual programs as shown in Table II. All of them are collected from two different sources (the 2nd column): 1) the benchmark from the open source of prior studies, and 2) the test cases from GitHub.

Table II presents these systems with their information included, such as the source of the cases (the 2nd column), the name of the test cases in our dataset (the 3rd column), the call graph properties, *i.e.*, the number of edges and nodes, before and after applying to PolyCall (from 4th to 9th column), as well as the construction time of each call graph (the last column). Despite the origin test cases are all multilingual programs, many of them have short call chains that are unable to show the experimental results clearly. On top of that, we added wrappers to original codes to extend the call chains in the guest side, making the experimental result more explicit and straightforward.

Evaluating PolyCall on this macro benchmark demonstrates the effectiveness and usefulness of PolyCall on ubiquitous multilingual applications.

D. Peer Tools

To accurately assess and showcase the capabilities of PolyCall, it is imperative to select fine-grained call graph construction tools for peer comparison. Unfortunately, current call graph construction tools are not available due to the following factors: first, few tools are currently available. Despite prior works toute their tools, many of them are not public available.

TABLE I: Experimental results on micro-benchmark.

Languages	Programs	Edges before	Edges after	Edges ratio	Nodes before	Nodes after	Nodes ratio	Construction time(s) / per line (ms)
Golang/C	pwd.wat	1,696	1,743	97.30%	594	621	95.65%	3.26 / 0.01
	sqrt.wat	402	454	88.55%	173	212	81.60%	0.68 / 0.01
	arith.wat	1,695	1,697	99.88%	594	595	99.83%	3.14 / 0.01
	callback.wat	407	410	99.27%	174	176	98.86%	0.53 / 0.01
	runtime.wat	1,696	1,744	97.25%	595	623	95.51%	2.97 / 0.01
	loop_call.wat	1,728	1,730	99.88%	601	602	99.83%	2.91 / 0.01
	simple_print.wat	407	459	88.67%	174	213	81.69%	0.70 / 0.01
Rust/C	rpwd.wat	768	878	84.74%	211	246	85.77%	0.18 / 0.01
	rsqrt.wat	767	876	87.56%	210	245	85.71%	0.18 / 0.01
	rarith.wat	768	879	87.37%	211	247	85.43%	0.18 / 0.01
	rsqrtadd.wat	769	881	87.29%	212	247	85.83 %	0.18 / 0.01
	rcallback.wat	1,103	1,225	90.04%	242	274	88.32%	0.20 / 0.01
	rloop_call.wat	1,103	1,225	90.04%	242	274	88.32%	0.19 / 0.01
	rsimple_print.wat	767	876	87.56%	210	245	85.71%	0.18 / 0.01

Second, public tools have not been widely used and are not well maintained. After a thorough investigation and validation of related tools accessible on GitHub, we found out that the most used tool dedicated to Rust, that is, a crate named cargo-call-stack [61], owns only 516 stars and with a pile of bugs reported by users such as build failure, incompatibility on universal architectures and functionality missing remain unsolved, and the project is stale with the last commit dates back to Feb 28th, 2023. Third, these tools perform on a relatively high level in comparison with Wasm, leading to a huge disparity in the granule of call chains. On top of that, a direct comparison would bring no reference value.

As a result, we selected Go-callvis [57], a call graph constructor designed for Go considering the aforementioned criterion in Section IV-D for the upcoming case study, whereas for Rust, we have to deploy a compromise approach to perform the emulation, which would be discussed in Section V-E.

E. Effectiveness

To better demonstrate the effectiveness of PolyCall, we apply PolyCall to micro-benchmarks. Despite there exist available call graph construction tools, many of them come with the coarse granular level, *e.g.*, can only perform on source languages or middle representation level, in comparison with Wasm. For example, the aforementioned call graph in Fig. 2 has 14 edges and 6 nodes in total, while the program in Wasm achieves 465 edges and 217 nodes. On top of that, we manually modify the test cases and translate them into Wasm for a same-level comparison since such a huge disparity caused by different language levels can not be mitigated easily. More specifically, we first erase the calling chain from the guest side to emulate the black box situation in ubiquitous analysis tools to construct a sub-benchmark labeled with "before", and then be compiled to Wasm. Second, we compile the original programs into Wasm as the "after" sub-benchmark. Finally, we analyze the output information of the call graph collected during the construction process.

We present, from the third to the eighth columns of Table I, the experimental result. In summary, PolyCall successfully constructed all multilingual call graphs that call chains from the guest sides are complete after our manual investigation. More specifically, there exists an increment after applying PolyCall to **MultiBench**, that is, 64.29% (9/14) of them with a growth of around 10% in terms of edges and nodes (*i.e.*, the 2nd row, the 7th row, and all of the sub-benchmark in Rust), while cases (*e.g.,* the 1st row, and the 3rd row) increase less. We scrutinize the source codes and discover that the short call chains from the guest sides in these test cases contributes to the modest growth. In a nutshell, PolyCall can effectively reveal the call chain from both sides regardless of the complexity of calling relationships.

Summary: PolyCall can successfully construct the complete call graph in multilingual programs, achieving growth on edges and nodes of 10% approximately in 64.29% test cases, demonstrating its effectiveness.

F. Usefulness

To answer **RQ2** by demonstrating the usefulness of PolyCall, we apply it to our second benchmark, real-world multilingual programs. Table II presents the experiment results on this benchmark. Among the test cases, we employ the metrics from the micro-benchmark and find out PolyCall successfully constructs all the call graphs with the number of edges and nodes mounted. The percentage increase in edges and nodes varied across the data, showcasing a range of changes. The minimum increase was observed in the 14th and 15th rows, with a modest 0.37% difference in edges. In contrast, the maximum edge increase of 12.49% occurred in the twelfth row, exhibiting a significant change. Furthermore, the percentage increase in nodes ranged from a minimum of 0.8% in the 16th row to a substantial 18.40% in the 1st row. These variations highlight the diverse impact of the length of call chains from the guest side on edges and nodes throughout the data.

TABLE II: Experimental results on real-world multilingual systems.

Languages	Source	Name	Edges before	Edges after	Edges ratio	Nodes before	Nodes after	Nodes ratio	Construction time(s) / per line (ms)
Golang/C	[62]	case1	402	454	88.55%	173	212	81.60%	0.70 / 0.01
	[63]	case2	1,725	1,777	97.10%	603	630	95.71%	2.93 / 0.01
	[64]	case3	920	969	94.94%	355	385	92.21%	1.54 / 0.01
	[65]	case4	433	486	89.09%	186	224	83.04%	0.76 / 0.01
	[66]	case5	1,762	1,828	96.39%	613	645	95.04%	3.31 / 0.01
	[66]	case6	1,730	1,783	97.03%	602	630	95.56%	2.94 / 0.01
	[67]	case7	1,698	1,745	97.32%	595	622	95.66%	3.01 / 0.01
	[68]	case8	1,793	1,843	97.29%	625	652	95.86%	2.90 / 0.01
	[69]	case9	1,053	1,109	95.00%	394	428	92.10%	1.92 / 0.01
	[70]	case10	1,704	1,750	97.37%	596	623	95.67%	2.96 / 0.01
Rust/C	[71]	case1	789	792	99.62%	218	220	99.10%	0.18 / 0.01
	[71]	case2	778	889	87.51%	216	253	85.38%	0.18 / 0.01
	[71]	case3	789	792	99.62%	218	220	99.10%	0.17 / 0.01
	[71]	case4	797	800	99.63%	220	222	99.10%	0.16 / 0.01
	[71]	case5	808	811	99.63%	219	221	99.10%	0.17 / 0.01
	[71]	case6	832	836	99.52%	248	250	99.20%	0.22 / 0.01
	[71]	case7	790	794	99.50%	219	221	99.10%	0.17 / 0.01
	[71]	case8	788	792	99.50%	217	219	99.10%	0.16 / 0.01
	[71]	case9	797	801	99.50%	230	233	98.71%	0.18/ 0.01
	[71]	case10	797	801	99.50%	226	229	98.69%	0.53 / 0.02

Summary: From 20 in-the-wild multilingual programs, PolyCall successfully constructs complete call graphs, demonstrating its usefulness on real-world Wasm programs.

G. Performance

To answer **RQ3** by investigating the performance introduced by PolyCall, we apply PolyCall to micro-benchmark and macro-benchmark (14 + 20) and each multilingual program was analyzed 10 rounds to calculate the average time of call graphs construction.

TABLE I and TABLE II (the last column) presents the performance of PolyCall, that is time for call graphs construction on Wasm, and the time distribution to each line of Wasm code.

Experimental results demonstrated that PolyCall is stable in call graph construction in multilingual applications: the time distribution is consistent with around 0.1 millisecond on each line, whereas the time spent on the entire programs varies from 0.16 seconds to 3.31 seconds, depending on the code size of Wasm modules.

Summary: PolyCall is effective in call graph construction for multilingual programs with 64.71% (22 / 34) of test cases under 1 second.

H. Case Study

To show PolyCall's capability of call graph construction in practice and to understand PolyCall's effectiveness, we present, as a showcase, a multilingual call graph constructed by PolyCall for comparison.

Fig. 5 and Fig. 6 presents the source code and its call graph constructed by Go-callviz [57] and PolyCall for comparison. The source code excerpt is originated from the public repository [68] with a small adaptation for illustration. It is notable that we have included a complete list of the complete call graphs in our open source.

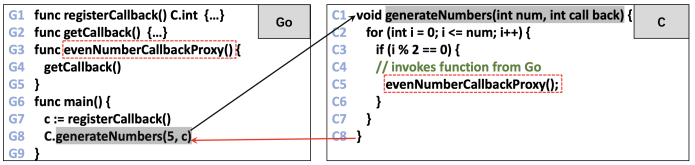

Fig. 5: Go/C interoperation through FFI.

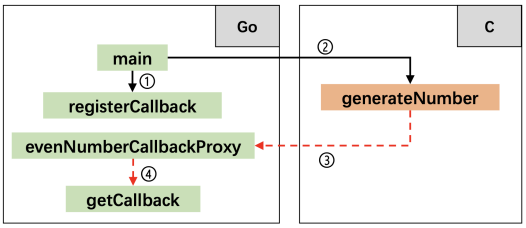

Fig. 6: The call graph of the Go/C program presented in Fig. 5.

Fig. 6 showcases the complete call graph with the call chains invisible to Go-callviz in red lines. The left side depicts the call graph from the Go side, and the call chain (①) (line G6 to G7) is visible for both PolyCall and Go-callviz. Then the control flows to the C side (line G8) and the call chain (②) across the boundary can be tracked as well. Further, `generateNumbers` (line C1) on the C side invokes the host-side function, `evenNumberCallbackProxy` (line C5), returns the control to Go, and this cross-boundary behavior (③) can only be visible to PolyCall. Moreover, Go-callviz is incapable for detecting the coming call chain (④) on the Go side, while PolyCall is able the construct the call graph for that. In conclusion, PolyCall is able to construct the complete call graph, while the call chains originated from the C side are invisible to Go-callviz.

VI. Discussion

In this section, we discuss some possible enhancements to this work and directions for future work as well as the threats to validity in this paper. It should be noted that this work represents the first step to proposing a unified and effective call graph construction analysis for multilingual programs.

Source languages and tool chains. While PolyCall's design targets the unified execution binary set Wasm, it should be portable for any multilingual programs and toolchains which support the compilation to Wasm, the current experiment focuses on multilingual programs written in Rust/C and Go/C, and some of them are compiled through third party compiler. Meanwhile, Javascript's support for compiling to Wasm, is becoming mature, but the current compiler only supports single-language compilation due to the dynamic feature of language itself. Hence it is interesting to investigate how cross-language applications written in Javascrpit/C can be compiled into Wasm and how to construct an integral call graph for two distinct Wasm modules. We leave this one of our future work.

Analysis framework. PolyCall is capable of the call graph construction, which illustrates the call relationship among the functions by utilizing the unified language set Wasm. On top of that, it provides us with an opportunity to exploit the advantages of Wasm and further extend the PolyCall to a more comprehensive analysis framework with accurate information tracing based on the theories such as the data flow analysis and control flow analysis. Hence, we leave this as our future work.

Vulnerability detection. PolyCall offers a comprehensive call graph structure for multilingual programs, facilitating seamless tracing of call chains between the host and guest languages. This powerful feature allows for an in-depth analysis of the call graph, reinforcing the ability to the identification of potential security vulnerabilities such as untrusted library function calls [72], and privilege escalation [73]. As a result, it is intriguing to explore the defects in multilingual programs using PolyCall, with the aim of uncovering potential bugs that may have evaded detection by existing tools. This aspect will be the focus of our future work.

Threats to validity. As any work on program analysis, there are potential threats to the validity of our work. We attempt to remove these threats where possible and mitigate the effects when removal is not possible. Specifically, we have applied our benchmark to the available call graph construction tools and PolyCall for comparison. However, the effectiveness of PolyCall can not be illustrated through the direct comparison of the discrepancy of the edges and nodes as our metrics to measure the call graph completeness due to the granular difference. To mitigate this, we further emulate the black-box situation, that is, invisible call relationships from the guest side, of the current tools by modifying the call chains on the guest sides. After a thorough investigation of the behavior of these tools and our emulation results, we believe the experimental result is convincing and reliable.

VII. Related Work

In recent years, there has been a significant amount of research on both call graph construction for multilingual programs and Wasm. However, this work stands for a novel contribution to these fields.

Call graph construction. We present the chronology of the call graph construction. Ryder et al. [74] made significant contributions to the field of call graph analysis by first introducing the acyclic representation of the call graph and the general algorithm for constructing it. Subsequently, numerous solutions for constructing static and dynamic call graphs have been proposed. For example, Murphy et al. [75] conducted an empirical study to evaluate the completeness of call graphs generated by various static extractors. Their work provided insights into the spectrum of call graph completeness and the effectiveness of different static analysis techniques; and Mary et al. [76] presented an efficient algorithm for call graph computing, enabling more effective program understanding and optimization. Following that, dedicated call graphs designed for specific applications have occurred. In addition to general-purpose call graph construction techniques, specialized call graph solutions have been developed for specific programming languages and applications. Nielsen et al. [19] proposed a novel approach to call graph construction for Node.js applications, and Salis et al. [77] utilized call graphs to trace the data flow in Python. These studies primarily concentrate on single-language programs, as a result, are not suitable for addressing the challenge of constructing cross-language call graphs in our study.

Multilingual programs security. There have been many studies on the security of multilingual applications. Mergendahl et al [16] introduce the threat model for analyzing cross-language attacks on Rust and Go. Li et al. [11] design and implement a pass in LLVM to detect cross-language memory vulnerability in Rust/C. Jiang et al. [12] present a dynamic analysis framework, PolyCruise, to perform information flow analysis on multilingual applications. Costanzo et al. [78] formally verify end-to-end security of software systems that consist of both C and assembly. Hu et al. [14] formalize the Rust/C programs representable by an intermediate representation and effectively detect memory safety vulnerabilities and integer overflows. Our work differs from the above efforts in that we propose using a completely unified analysis platform, Wasm, which has a promising ability to detect cross-language bugs.

Program analysis on Wasm. Program analysis for Wasm has been extensively researched and several notable studies have contributed to this field. Haas et al. [27] proposed an operational semantics and type system for ensuring the safety of Wasm programs. Stiévenart et al. [79] developed an information flow analysis algorithm, while Lopes et al. [31] introduced a vulnerability detection framework using a code property graph. Chen et al. [50] proposed a fuzzing framework specifically for Wasm smart contracts. Additionally, Szanto et al. [80] and Fu et al. [81] conducted taint analysis to track data propagation. However, these studies do not focus on detecting

bugs originated from cross-language programs. In contrast, our study introduces a novel notion to detect cross-language bugs by constructing a call graph on the unified and secure Wasm.

VIII. CONCLUSION

This paper presents POLYCALL, the first unified program analysis framework for analyzing multilingual applications. At the core of POLYCALL is the rigorous specification language Wasm, as the compilation from various multilingual programs to it on which call graphs can be constructed non-distinctively. We have implemented a prototype system for POLYCALL and conducted extensive experiments. Experimental results show that POLYCALL can effectively construct call graphs for cross-language applications, with negligible overhead. This work represents a new step towards holistic analysis of multilingual programming, making the promise of secure multilingual programming a reality.

REFERENCES

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, Curran Associates, Inc., 2019.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning,"

[3] "Pillow." https://pillow.readthedocs.io/en/stable/index.html.

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with numpy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[5] G. Tan and J. Croft, "An empirical security study of the native code in the jdk," in *Proceedings of the 17th Conference on Security Symposium*, SS'08, (USA), pp. 365–377, USENIX Association, July 2008.

[6] M. Hu, Q. Zhao, Y. Zhang, and Y. Xiong, "Cross-language call graph construction supporting different host languages," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 155–166, Mar. 2023.

[7] "Torch.kthvalue returns random value when the k is invalid · issue #68813 · pytorch/pytorch." https://github.com/pytorch/pytorch/issues/68813.

[8] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, "Rid: Finding reference count bugs with inconsistent path pair checking," *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 531–544, Mar. 2016.

[9] S. Li and G. Tan, "Finding bugs in exceptional situations of jni programs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 442–452, Association for Computing Machinery, Nov. 2009.

[10] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," *ACM Transactions on Programming Languages and Systems*, vol. 30, pp. 18:1–18:63, Aug. 2008.

[11] 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.), vol. 13556, pp. 680–700, Cham: Springer Nature Switzerland, 2022.

[12] W. Li, J. Ming, X. Luo, and H. Cai, "Polycruise: A cross-language dynamic information flow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2513–2530, 2022.

[13] Z. Li, J. Wang, M. Sun, and J. C. Lui, "Mirchecker: Detecting bugs in rust programs via static analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, (Virtual Event Republic of Korea), pp. 2183–2196, ACM, Nov. 2021.

[14] S. Hu, B. Hua, L. Xia, and Y. Wang, "Crust: Towards a unified cross-language program analysis framework for rust," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, (Guangzhou, China), pp. 970–981, IEEE, Dec. 2022.

[15] G. Tan and G. Morrisett, "Ilea: Inter-language analysis across java and c," *ACM SIGPLAN Notices*, vol. 42, pp. 39–56, Oct. 2007.

[16] S. Mergendahl, N. Burow, and H. Okhravi, "Cross-language attacks," in *Proceedings 2022 Network and Distributed System Security Symposium*, (San Diego, CA, USA), Internet Society, 2022.

[17] B. Yang, J. Wu, and C. Liu, "Mining data chain graph for fault localization," in *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, pp. 464–469, July 2012.

[18] X. Chen, J. Jiang, W. Zhang, and X. Xia, "Fault diagnosis for open source software based on dynamic tracking," in *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*, pp. 263–268, Nov. 2020.

[19] B. B. Nielsen, M. T. Torp, and A. Møller, "Modular call graph construction for security scanning of node.js applications," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, (Virtual Denmark), pp. 29–41, ACM, July 2021.

[20] W. Zhu, Z. Feng, Z. Zhang, J. Chen, Z. Ou, M. Yang, and C. Zhang, "Callee: Recovering call graphs for binaries with transfer and contrastive learning," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2357–2374, IEEE Computer Society, Dec. 2022.

[21] "The rust programming language - the rust programming language." https://doc.rust-lang.org/book/.

[22] "Compiling from rust to webassembly - webassembly — mdn." https://developer.mozilla.org/en-US/docs/WebAssembly/Rust_to_Wasm, May 2023.

[23] "Compiling a new c/c++ module to webassembly - webassembly — mdn." https://developer.mozilla.org/en-US/docs/WebAssembly/C_to_Wasm, May 2023.

[24] "Bytecodealliance/javy: Js to webassembly toolchain." https://github.com/bytecodealliance/javy.

[25] "Compile python to webassembly (wasm) — unofficial python development (victor's notes) documentation." https://pythondev.readthedocs.io/wasm.html.

[26] "Webassembly." https://github.com/golang/go/wiki/WebAssembly.

[27] 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), pp. 185–200, ACM, June 2017.

[28] "Security - webassembly." https://webassembly.org/docs/security/.

[29] "Design/security.md at main · webassembly/design." https://github.com/WebAssembly/design/blob/main/Security.md#memory-safety.

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

[31] T. Brito, P. Lopes, N. Santos, and J. F. Santos, "Wasmati: An efficient static vulnerability scanner for webassembly," *Computers & Security*, vol. 118, p. 102745, July 2022.

[32] P. D. Schubert, B. Hermann, and E. Bodden, "Phasar: An inter-procedural static analysis framework for c/c++," in *Tools and Algorithms for the Construction and Analysis of Systems* (T. Vojnar and L. Zhang, eds.), Lecture Notes in Computer Science, (Cham), pp. 393–410, Springer International Publishing, 2019.

[33] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers on - CASCON '10*, (Toronto, Ontario, Canada), pp. 214–224, ACM Press, 2010.

[34] X. Hu, T.-c. Chiueh, and K. G. Shin, "Large-scale malware indexing using function-call graphs," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, (Chicago Illinois USA), pp. 611–620, ACM, Nov. 2009.

[35] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, and H. Yin, "Patchscope: Memory object centric patch diffing," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, (Virtual Event USA), pp. 149–165, ACM, Oct. 2020.

[36] S. Xi, S. Yang, X. Xiao, Y. Yao, Y. Xiong, F. Xu, H. Wang, P. Gao, Z. Liu, F. Xu, and J. Lu, "Deepintent: Deep icon-behavior learning for detecting intention-behavior discrepancy in mobile apps," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communi-*

cations Security, CCS '19, (New York, NY, USA), pp. 2421–2436, Association for Computing Machinery, Nov. 2019.
[37] "Language details of the firefox repo." https://4e6.github.io/firefox-lang-stats/.
[38] "Java native interface specification contents." https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html.
[39] "Ffi - the rustonomicon." https://doc.rust-lang.org/nomicon/ffi.html.
[40] "Python/c api reference manual." https://docs.python.org/3/c-api/index.html.
[41] "Going public launch bug · issue #150 · webassembly/design." https://github.com/WebAssembly/design/issues/150.
[42] "Roadmap - webassembly." https://webassembly.org/roadmap/.
[43] "Webassembly core specification." https://www.w3.org/TR/wasm-core-1/.
[44] "World wide web consortium (w3c) brings a new language to the web as webassembly becomes a w3c recommendation." https://www.w3.org/2019/12/pressrelease-wasm-rec.html.en.
[45] "Standardizing wasi: A system interface to run webassembly outside the web – mozilla hacks - the web developer blog." https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface.
[46] "Webassembly high-level goals - webassembly." https://webassembly.org/docs/high-level-goals/.
[47] M. Kim, H. Jang, and Y. Shin, "Avengers, assemble! survey of webassembly security solutions," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, (Barcelona, Spain), pp. 543–553, IEEE, July 2022.
[48] "Homepage — wasmcloud." https://wasmcloud.com/.
[49] R. Liu, L. Garcia, and M. Srivastava, "Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 94–105, 2021.
[50] W. Chen, Z. Sun, H. Wang, X. Luo, H. Cai, and L. Wu, "Wasai: Uncovering vulnerabilities in wasm smart contracts," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, (Virtual South Korea), pp. 703–715, ACM, July 2022.
[51] C. Kelton, A. Balasubramanian, R. Raghavendra, and M. Srivatsa, "Browser-based deep behavioral detection of web cryptomining with coinspy," in *Proceedings 2020 Workshop on Measurements, Attacks, and Defenses for the Web*, (San Diego, CA), Internet Society, 2020.
[52] N. He, R. Zhang, H. Wang, L. Wu, X. Luo, Y. Guo, T. Yu, and X. Jiang, "Eosafe: Security analysis of eosio smart contracts," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1271–1288, 2021.
[53] W. Bian, W. Meng, and Y. Wang, "Poster: Detecting webassembly-based cryptocurrency mining," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, (London United Kingdom), pp. 2685–2687, ACM, Nov. 2019.
[54] "Serverless edge compute solutions — fastly." https://www.fastly.com/products/edge-compute.
[55] "Scalar.video - let your creativity run wild on an infinite canvas." https://scalar.video/.
[56] "Torch2424/wasmboy: Game boy / game boy color emulator library, written for webassembly using assemblyscript. demos built with preact and svelte.." https://github.com/torch2424/wasmBoy.
[57] "Ofabry/go-callvis: Visualize call graph of a go program using graphviz." https://github.com/ofabry/go-callvis.
[58] "Understanding ownership - the rust programming language." https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html.
[59] "What is rustc? - the rustc book." https://doc.rust-lang.org/rustc/what-is-rustc.html.
[60] "Tinygo - go compiler for small places." https://github.com/tinygo-org/tinygo, July 2023.
[61] J. Aparicio, "Cargo-call-stack." https://github.com/japaric/cargo-call-stack, July 2023.
[62] "Karalabe/xgo: Go cgo cross compiler." https://github.com/karalabe/xgo.
[63] "Mit-ll/cross-language-attacks." https://github.com/mit-ll/Cross-Language-Attacks.
[64] "Ianlancetaylor/cgosymbolizer: Experimental symbolizer for cgo backtraces." https://github.com/ianlancetaylor/cgosymbolizer.
[65] "Sbinet/go-python: Naive go bindings to the cpython2 c-api." https://github.com/sbinet/go-python.
[66] "Aleksi/cgo-by-example: Not updated for years. how to mix c and go with cgo." https://github.com/AlekSi/cgo-by-example.
[67] "Giorgisio/cgo: Cgo by example." https://github.com/giorgisio/cgo/tree/master.
[68] "Andreiavrammsd/cgo-examples: Examples of calling c code from go." https://github.com/andreiavrammsd/cgo-examples.
[69] "Fananchong/test_cgo_coredump: Cgocrash." https://github.com/fananchong/test_cgo_coredump.
[70] C. Pliakas, "Cgo example." https://github.com/cpliakas/cgo-example, June 2023.
[71] "Lizhuohua/rust-ffi-checker." https://github.com/lizhuohua/rust-ffi-checker.
[72] Q. Zhang, Z. Sura, A. Kundu, G. Su, A. Iyengar, and L. Liu, "Stackvault: Protection from untrusted functions," July 2019.
[73] J. Zhu, B. Chu, and H. Lipford, "Detecting privilege escalation attacks through instrumenting web application source code," in *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, SACMAT '16, (New York, NY, USA), pp. 73–80, Association for Computing Machinery, June 2016.
[74] B. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 216–226, May 1979.
[75] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *ACM Transactions on Software Engineering and Methodology*, vol. 7, pp. 158–191, Apr. 1998.
[76] M. W. Hall and K. Kennedy, "Efficient call graph analysis," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 227–242, Sept. 1992.
[77] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "Pycg: Practical call graph generation in python," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1646–1657, May 2021.
[78] D. Costanzo, Z. Shao, and R. Gu, "End-to-end verification of information-flow security for c and assembly programs," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, (Santa Barbara CA USA), pp. 648–664, ACM, June 2016.
[79] 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), pp. 13–24, IEEE, Sept. 2020.
[80] A. Szanto, T. Tamm, and A. Pagnoni, "Taint tracking for webassembly," No. arXiv:1807.08349, arXiv, July 2018.
[81] W. Fu, R. Lin, and D. Inge, "Taintassembly: Taint-based information flow control tracking for webassembly," No. arXiv:1802.01050, arXiv, Feb. 2018.