PROGRAM ANALYSIS OF WEBASSEMBLY BINARIES

Von der Fakultät für Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines

DOKTORS DER NATURWISSENSCHAFTEN
(DR. RER. NAT.)

genehmigte Abhandlung

Vorgelegt von

DANIEL LEHMANN

aus Leipzig, Deutschland

Hauptberichter: Prof. Dr. Michael Pradel
1. Mitberichter: Prof. Dr. Andreas Zeller
2. Mitberichter: Prof. Dr. Mathias Payer

Tag der mündlichen Prüfung: 5. Juli 2022

Institut für Software Engineering der Universität Stuttgart
2022

Program Analysis of WebAssembly Binaries
Analyse von WebAssembly Binärprogrammen

Doctoral dissertation
Submitted by Daniel Lehmann
Advised by Prof. Dr. Michael Pradel

Oral defense on July 5, 2022
Examination committee:
Prof. Dr. Michael Pradel
Prof. Dr. Andreas Zeller
Prof. Dr. Mathias Payer

Institut für Software Engineering
Fakultät für Informatik, Elektrotechnik und Informationstechnik
Universität Stuttgart

ABSTRACT

WebAssembly is a rapidly expanding low-level bytecode that runs in browsers, on the server side, and in standalone runtimes. It brings exciting opportunities to the Web and has the potential to radically change the distribution model of software. At the same time, WebAssembly comes with new challenges and open questions, in particular regarding program analysis and security.

The goal of this dissertation is to answer such questions and to support developers with novel insights, datasets, and program analysis techniques for WebAssembly binaries. WebAssembly is frequently compiled from unsafe languages such as C and C++. That begs the question: What happens with memory vulnerabilities when compiling to WebAssembly? We start by analyzing the language and ecosystem and find severe issues, such as the inability to protect memory, missing mitigations, and new attacks that are unique to WebAssembly. To assess the risk in practice, we collect WasmBench, a large-scale dataset of real-world binaries, and study common source languages and usages of WebAssembly. To find and mitigate vulnerabilities leading to such attacks, we develop Fuzzm, the first binary-only greybox fuzzer for WebAssembly. Due to WebAssembly's novelty and its low-level nature, developers are also in dire need of techniques to help them understand and analyze WebAssembly programs. For that, we introduce Wasabi, the first dynamic analysis framework for WebAssembly. It employs static binary instrumentation, which requires us to address several technical challenges, such as handling WebAssembly's static types and structured control-flow. Finally, we present SnowWhite, a learning-based approach for recovering high-level types from WebAssembly binaries. Unlike prior work, also among other binary formats, it generates types from an expressive type language, and not by classification into few fixed choices.

This dissertation shows that program analysis of WebAssembly binaries has versatile applications and can be reliably and efficiently implemented. Given the young age yet steep trajectory of WebAssembly,

it is going to be an important language and binary format for years to come. We look forward to many more works in this area, and hope they can build on the results, techniques, and datasets put forth in this dissertation.

KEYWORDS WebAssembly, program analysis, software security, study, dataset, fuzzing, static binary instrumentation, neural software analysis, type recovery

ZUSAMMENFASSUNG

WebAssembly ist ein zunehmend genutzter Low-Level-Bytecode, der in Browsern, auf Servern und in unabhängigen Laufzeitumgebungen ausgeführt wird. WebAssembly eröffnet neue, spannende Nutzungsmöglichkeiten für das Web und hat das Potenzial, radikal zu verändern, wie Software verteilt wird. Gleichzeitig bringt WebAssembly neue Herausforderungen und offene Fragen mit sich, insbesondere in Bezug auf Programmanalyse und Sicherheit.

Diese Dissertation hat zum Ziel, jene Fragen zu beantworten und Softwareentwickler mit neuen Erkenntnissen, Datensätzen und Programmanalysetechniken für WebAssembly zu unterstützen. WebAssembly wird häufig aus unsicheren Quellsprachen wie C und C++ kompiliert. Das wirft die Frage auf, wie sich Speicherschwachstellen bei der Übersetzung nach WebAssembly verhalten. Wir gehen dieser Frage eingangs nach und finden schwerwiegende Probleme, zum Beispiel fehlenden Speicherschutz und neuartige Angriffe, die durch WebAssembly überhaupt erst ermöglicht werden. Um das Risiko solcher Angriffe einzuschätzen, stellen wir weiterhin einen großen Datensatz realistischer WebAssembly-Programme zusammen, WasmBench, und untersuchen die Quellsprachen und Verwendung von WebAssembly in der Praxis. Um Schwachstellen zu entdecken und Angriffe zu verhindern, entwickeln wir Fuzzm, den ersten Greybox-Fuzzer für WebAssembly-Binärprogramme. Da WebAssembly neu ist und auf niedriger Abstraktionsschicht operiert, benötigen Entwickler darüber hinaus dringend Techniken, die ihnen helfen, WebAssembly-Programme zu verstehen und zu analysieren. Zu diesem Zweck entwickeln wir Wasabi, das erste Dynamische-Analyse-Framework für WebAssembly. Es nutzt statische Instrumentierung, was uns vor mehrere technische Herausforderungen stellt, z. B. aufgrund der statischen Typen und strukturierten Kontrollflusskonstrukte in WebAssembly. Schließlich stellen wir SnowWhite vor, eine Programmanalyse, die mittels maschinellem Lernen Datentypen mit hohem Abstraktionsniveau aus WebAssembly rekonstruiert. Im Gegensatz zu früheren Arbeiten, auch solchen für an-

dere Binärformate, generiert unser Ansatz Datentypen mithilfe einer ausdrucksstarken Typsprache und nicht durch Auswahl aus wenigen festgelegten Klassen.

Diese Dissertation zeigt, dass die Analyse von WebAssembly-Binärprogrammen vielseitige Anwendungen hat, und zuverlässig und effizient implementiert werden kann. Angesichts des jungen Alters und der gleichzeitig starken Entwicklung wird WebAssembly auch in den kommenden Jahren eine wichtige Bytecode-Sprache sein. Wir sind gespannt auf viele weitere Arbeiten in diesem Bereich und hoffen, dass sie auf den Ergebnissen, Techniken und Datensätzen aufbauen können, die wir in dieser Dissertation vorstellen.

SCHLAGWÖRTER WebAssembly, Programmanalyse, Softwaresicherheit, Studie, Datensatz, statische Instrumentierung von Binärprogrammen, Neuronale Softwareanalyse, Rekonstruktion von Datentypen

ACKNOWLEDGEMENTS

Getting a PhD is quite a journey. For me, it was sometimes an exciting adventure, sometimes an uphill struggle, but ultimately a very rewarding experience of learning, perseverance, and growth. It was only possible because I had the best companions and mentors along the way anyone could ask for. Without them, I would have never embarked, stayed on course, and completed this five-year journey.

First and foremost, I want to thank my advisor, Michael Pradel. I am immensely grateful for the trust he placed in me right from the start, and the freedom and support he gave me to pursue my own ideas. Despite having many other responsibilities, he always made time for discussions, feedback, and answering my questions. Besides his direct support, I also learned a lot from his excellent example of how to conduct research. (Unfortunately, his impeccable time management has not yet rubbed off on me.) Thank you, Michael, for always being available and positive, for all the guidance along the way, and for encouraging me that not everything has to be (or ever can be) perfect.

I also want to thank the members of my PhD committee, Andreas Zeller and Mathias Payer, for committing their time to review my thesis and for the insightful questions and discussions that followed.

None of the research projects of the past five years would have been possible without the amazing academic collaborators, advisors, and hard-working students, whom I had the pleasure to work with, including Sandro Tolksdorf, Johannes Kinder, Aaron Hilbig, Martin Toldam Torp, Weihang Wang, Alan Romano, Frank Tip, and Michelle Thalakottur. Thank you for the inspiring discussions, whiteboard sketches, pair programming sessions, and writing marathons.

I am also grateful for two opportunities to peek into industry, in an internship at Microsoft Research with Patrice Godefroid and Marina Polishchuk, and in an internship at Google's V8 team with Clemens Backes and Jakob Kummerow. Working on a massive real-world codebase was truly humbling and made me appreciate the importance of software engineering quality and tools even more.

My PhD journey would not have been as enjoyable without my great colleagues at the Software Lab in Darmstadt and Stuttgart. Thank you to Cristian-Alexandru Staicu, Marija Selakovic, Jibesh Patra, Marina Billes, Andrew Habib, Luca Di Grazia, Moiz Rauf, Aryaz Eghbali, Matteo Paltenghi, and Islem Bouzenia, for the lunch, tea, and snack breaks, for philosophical discussions, emotional support in times of setback, and for creating a warm and intellectual environment. A big thanks also to Andrea Püchner and Katharina Plett, our team assistants with the superpower to make everything run smoothly even when faced with countless paperwork.

Finally, I want to thank my biggest sources of comfort, love, distractions, and happiness: my friends and family. Thank you to my study group, Ji-Ung, Magnus, Mark, Marvin, and Tim, for the many hours of (not) studying, sharing the passion for our subject, proofreading the thesis, and defense dry runs. Thank you to all my friends in Darmstadt who made the city feel like home, particularly Victoria, Gerrit, and Gregor, for sharing their own experiences of doing a PhD, and for our cooking nights. Thank you to my flatmates throughout the years, especially Sebastian, Robin, Johannes, and Julian, with whom even splitting the auxiliary cost statement can become a fun evening activity. Thank you to Inga, for all the good times, sailing through lockdown together, for nudging me out of well-trodden paths, for all the things I would not have experienced without you, and for enduring my occasional owl life. Thank you to Julia for introspective discussions and that I can always ask my big sister for advice, and to Fiona for showing me that there is more to life than a PhD. Lastly, I am deeply grateful to my parents, Horst and Jeanette, for their unconditional love, for always being excited about what I do, and for apparently setting me on an early path towards research by answering my many questions, including those about door handles.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1 INTRODUCTION

The *World Wide Web* is becoming ever more powerful and complex. Gone are the days when websites were essentially text documents, "hyperlinked", but not very interactive. Today, the Web is a software platform for rich *web applications*, offering messaging and conferencing,[1] music and video streaming,[2] social media,[3] navigation services,[4] and collaborative office software,[5] just to name a few examples. The companies providing such services are extremely valuable, with profits in the billions, and stock market capitalizations in the trillions of US dollars [Internet Companies]. Besides their economical importance, web applications also have clear advantages for users: Unlike classical native software, web applications need not be installed; are always up-to-date; can be conveniently accessed and shared via hyperlinks; and are portable across different devices, operating systems, and hardware architectures [Bleigh 2014].

Which underlying technologies enable the feat of web applications in the first place? The Internet for global data transmission certainly plays one part. The other part is allowing web browsers not to merely *display* content that was generated by a server, but to *execute* application code on the client side, namely the user's machine itself. Up until recently, *JavaScript* was the only client-side programming language directly supported by browsers.[6] JavaScript's role on the client side undoubtedly contributed to its success, making it one of the most used programming languages [StackOverflow Survey 2021; TIOBE Index 2022]. Another contributor is that developers prefer using a sin-

1 `https://slack.com`, `https://zoom.us`, `https://web.whatsapp.com`
2 `https://www.spotify.com`, `https://www.youtube.com`, `https://www.netflix.com`
3 `https://www.facebook.com`, `https://twitter.com`, `https://www.instagram.com`
4 `https://www.openstreetmap.org`, `https://www.google.com/maps`
5 `https://www.office.com`, `https://docs.google.com`
6 Other technologies, such as Java applets or Flash also allowed running code on the client side, but those required third-party browser plugins that had to be manually installed and updated. Frequent security issues and lack of integration with the Web led those technologies to being deprecated in favor of JavaScript.

gle language throughout the whole application stack, including the server side. This led to the emergence of the *Node.js* runtime and a large ecosystem of JavaScript packages and development tools,[7] popularizing JavaScript outside the browser as well.

JavaScript was famously designed in just ten days as a "silly little brother language" to languages like Java [Severance 2012]. Clearly, it has been much more successful than anticipated, and, arguably, than it should have been. JavaScript's inherent limitations become more and more apparent: Due to many dynamic language features, good JavaScript performance requires complex just-in-time (JIT) compilers, e.g., *SpiderMonkey* in Firefox or *V8* in Google Chrome and Node.js.[8] Achieving predictable performance is still difficult, and the complexity of JIT compilers makes them an attractive target for attackers [Groß and Burnett 2022]. JavaScript being a human-readable text format made sense for small scripts, but as a code distribution format, it is inadequate. The median website today contains almost half a megabyte of JavaScript code as text [Goel 2021]; just parsing and compiling this code accounts for a major portion of the load time of websites [Clark 2017]. Finally, more and more JavaScript code is *generated* rather than directly written. To reuse code from other languages, or just for newer language features and static error checking, several languages compile to JavaScript.[9] Curiously, that extends even to C and C++, which can be compiled to *asm.js*, a stylized subset of JavaScript.[10] While certainly an achievement, it can only be described as one giant hack. Inadvertently, JavaScript has become the main "code format for the Web", but certainly not because it was designed for it.

1.1 WEBASSEMBLY

In 2015, the four major browser vendors[11] publicly announced working on *WebAssembly*, a fresh start for a portable, low-level bytecode [Bastien 2015]. Initially, WebAssembly was meant to complement JavaScript, in particular for compute-intensive parts of modern web applications. In that, it has already been very successful. Implemented by

7 https://nodejs.org, https://www.npmjs.com, https://webpack.js.org
8 https://spidermonkey.dev, https://v8.dev
9 https://www.typescriptlang.org, https://babeljs.io, https://github.com/google/j2cl, https://www.scala-js.org
10 http://asmjs.org/spec/latest/
11 Apple (Safari), Google (Chrome), Microsoft (IE, Edge), and Mozilla (Firefox).

all major browsers in 2017 [Wagner 2017], officially standardized in 2019 [WebAssembly Specification], WebAssembly is supported by 95% of all global browser installations as of April 2022.[12] It is used in several large web applications, including commercial products, such as AutoCAD and Figma,[13] and brings former desktop-only applications to the browser, such as Google Earth, the Unreal game engine, and even fully-fledged Adobe Photoshop.[14] As a fast, low-level, portable bytecode for browsers, it might fundamentally change software distribution as we know it, allowing web applications to take over use cases that were formerly reserved to native applications.

WebAssembly is not necessarily limited to only the Web either. Its simplicity and generality has sparked interest for using it outside the browser as well, e.g., in cloud and edge computing,[15] on resource constrained devices,[16] for smart contract systems,[17] and even as a bytecode for stand-alone runtimes.[18] Its young age notwithstanding, WebAssembly was awarded the ACM SIGPLAN Programming Languages Software Award in 2021.[19] WebAssembly and its ecosystem, although still evolving, have already gathered significant momentum and will be an important computing platform for years to come.

As the name implies, WebAssembly is designed as a portable, low-level compilation target, with good performance and a compact binary representation [Haas et al. 2017; WebAssembly Website]. Listing 1.1 shows an example program in both the binary format (`.wasm`) and the corresponding human-readable text format (`.wat`). The binary format is usually produced by compilers and consumed by runtimes. It is fast to send over the network and quick to parse. The text format is mostly

12 Statistic for all tracked devices on `https://caniuse.com/?search=WebAssembly`.

13 `https://web.autocad.com`, `https://www.figma.com`

14 `https://earth.google.com/web/`, `https://photoshop.adobe.com/`

15 [Hall and Ramachandran 2019; Shillaker and Pietzuch 2020] and industry offerings by Cloudflare [Varda 2018] and Fastly [Hickey 2019, 2018].

16 [Gadepalli et al. 2020; Gurdeep Singh and Scholliers 2019], WebAssembly Micro Runtime (WAMR, `https://github.com/bytecodealliance/wasm-micro-runtime`)

17 WebAssembly is the bytecode of EOSIO (`https://eos.io`), and discussed as the future bytecode for Ethereum 2.0 [McCallum 2019].

18 WASI, the WebAssembly System Interface, provides a portable operating system interface to stand-alone WebAssembly applications [Clark 2019; WASI Website]. Several WebAssembly runtimes support WASI, e.g., Wasmtime (`https://github.com/bytecodealliance/wasmtime`). An industry organization furthering WebAssembly and WASI is the Bytecode Alliance (`https://bytecodealliance.org`), with members such as Amazon, Arm, Google, Intel, Microsoft, Mozilla, and Siemens.

19 `http://www.sigplan.org/Awards/Software/`

```
;; WebAssembly magic bytes (\0asm) and version number (1.0).
00 61 73 6D         (module
01 00 00 00

;; Type section: 13 bytes long, 3 types.
01 0D 03
                      ;; Functions and instructions are statically typed.
                      ;; All types are declared first and used later.
  60 01 7E 00         (type $t0 (func (param i64) (result))) ;; See lines 19,
  60 01 7F 01 7E      (type $t1 (func (param i32) (result i64))) ;; 31, and
  60 00 00            (type $t2 (func (param) (result)))         ;; 38 below.

;; Import section: 14 bytes long, 1 import.
02 0E 01
                      ;; Import a print function from the host environment.
                      (import
  04 68 6F 73 74        "host"  ;; Imports have two-level names.
  05 70 72 69 6E 74     "print"
  00 00                 (func $print (type $t0)) ;; Type $t0 from above.
                      ) ;; End of import.

;; Memory section: 3 bytes long, 1 memory.
05 03 01
                      ;; Declares a memory of size 1 page (64KiB).
  00 01               (memory 1)   ;; Zero-initialized by default.

[...] ;; Function declaration section (omitted for brevity).

;; Code section, 21 bytes long, 2 functions.
0A 15 02
  0A 00               (func $load_and_increment (type $t1)
    42 01               i64.const 1  ;; Push a constant on the stack.
    20 00               local.get 0  ;; Push first argument on the stack.
    29 03 00            i64.load     ;; Load 64-bit int from that address.
    7C                  i64.add      ;; Add the two values on the stack.
    0B                ) ;; Implicit return at the end of the function.

  08 00               (func $main (type $t2)
    41 07               i32.const 7  ;; Push argument for next call.
    10 01               call $load_and_increment
    10 00               call $print  ;; This call leaves the stack empty.
    0B                )
                    ) ;; End of module.
```

Listing 1.1: An introductory WebAssembly program, showcasing several language features. The binary representation is shown `in gray` on the left, and the matching human-readable text representation on the right. The `wasm2wat` and `wat2wasm` programs convert between the two representations. The conversion is one-to-one, except for `;;comments` (which are lost in the binary) and `$indices` (which are just numbers in the binary; their name in the text format is only for human convenience).

for human understanding and for infrequent manual modification, similar to native assembly languages.

Conceptually, WebAssembly instructions execute on a stack-based virtual machine (VM). For example, the addition in line 35 pops two arguments from the implicit operand stack, and pushes the result to the stack again. In practice, WebAssembly bytecode is compiled to efficient native code, which is then executed without an operand stack.

As security is paramount when executing untrusted code on the Web, WebAssembly bytecode is more restricted than native code. Code is strictly separated from data, and instructions and functions are statically type-checked (e.g., lines 9, 19, and 32). WebAssembly programs execute in a sandbox, i.e., they can only access their own memory and call their own, declared functions. All interaction with the *host environment* must go through explicitly imported functions (line 16).

The bytecode and stack machine of WebAssembly is at first sight reminiscent of the JVM and its bytecode [JVM Specification]. However, there are several differences, most of them because WebAssembly is more low-level than Java. WebAssembly has no notion of objects or classes and no garbage collector. Instead, it features a flat, byte-addressable memory (line 25), which is organized and managed by the program itself. This makes WebAssembly a good compilation target from systems programming languages, such as C, C++, or Rust. We discuss the language and its ecosystem in more detail in Chapter 2.

1.2 PROGRAM ANALYSIS

To cope with the complexity of modern web applications, or any complex software system in general, developers often rely on program analysis. Program analysis can support developers in a wide range of everyday problems, e.g., when optimizing performance, understanding and debugging code, and in problems around software security, such as finding memory errors and mitigating exploitation. It is an integral part of many practical tools, e.g., compilers, linters, integrated development environments (IDEs), profilers, sanitizers, and fuzzers.

Program analysis techniques, including those discussed in this thesis, can be classified across several dimensions:

STATIC VS. DYNAMIC ANALYSIS Static analysis reasons about programs without actually executing them [Møller and Schwartzbach 2021].

This is desirable because it does not require program inputs. The downside of static analysis is that it needs to approximate the actual runtime behavior, rendering it often less precise than dynamic analysis. Conversely, dynamic analysis techniques do execute the program, e.g., on a test suite or on automatically generated inputs. The downside of dynamic analysis is that it typically cannot explore all program behaviors, due to the large or even infinite search space of possible behaviors.

CLASSICAL VS. NEURAL APPROACHES Classical program analysis is based on precise, logical reasoning, e.g., by collecting and solving constraints about a program. An example problem well suited to classical program analysis is to prove code unreachable to be able to remove it. However, not all problems can be expressed in such a framework. Especially problems with a human component or where there is no single correct answer are often better suited to statistical methods. An example is to determine which name is most "natural" for a certain variable. One particularly powerful statistical method of late are neural networks, giving rise to neural software analysis [Pradel and Chandra 2021]. There, machine learning models are trained on large amounts of program data labeled with the expected analysis output, and the model is then later queried on previously unseen problem instances.

PROGRAM REPRESENTATION One question is at which abstraction level programs are analyzed. The source code is what the developer originally wrote, but it is not always available, e.g., for third-party code or malware. Even if source code is available, an analysis on this level is tied to one particular programming language, which is problematic for software written in multiple languages. Machine code can also be analyzed, but this is challenging, e.g., due to information lost during compilation, and because there are many different hardware architectures. Finally, programs can also be analyzed in an intermediate representation (IR) that is somewhere between the source language and native code, e.g., compiler IRs or portable bytecode.

1.3 PROGRAM ANALYSIS OF WEBASSEMBLY BINARIES

In this dissertation, we focus on program analysis for WebAssembly, to help developers with understanding, optimizing, and improving the

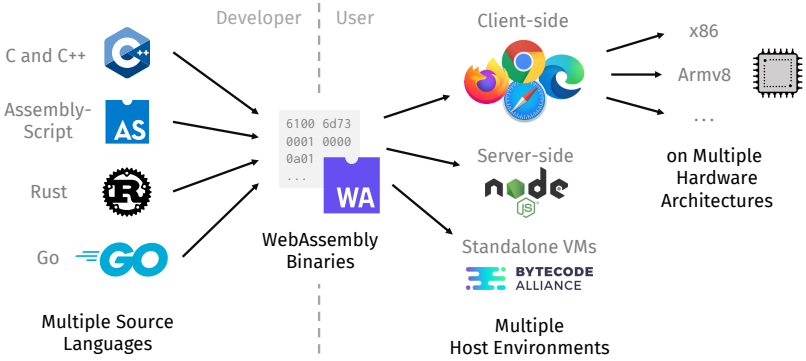


Figure 1.1: WebAssembly is the common bytecode between multiple source languages and multiple host environments.

security and correctness of WebAssembly applications on the Web and outside of browsers. The program analysis techniques in this thesis cover different points of the design space mapped out before. In particular, we employ both static and dynamic analysis, and both classical and neural approaches. In all cases, we analyze programs at the WebAssembly bytecode level, not source code or native machine code. In this dissertation, we often use the terms WebAssembly *bytecode* and WebAssembly *binaries* interchangeably.[20]

Why is program analysis of WebAssembly binaries worthwhile in particular? First, WebAssembly's position as the low-level bytecode for the Web makes it *highly relevant* today and more so in the future. If proponents of WebAssembly are to be believed, it has the potential to become a universal bytecode for applications outside the browser as well. Figure 1.1 shows that WebAssembly is the common interchange format between multiple source languages compiling to WebAssembly and different host environments executing it. Developing program analyses for each individual source language, or for each of the different runtimes does not scale. When source code is not available, there is also no choice but to analyze the WebAssembly bytecode directly.

Second, as a novel code format, analyzing WebAssembly is rife with *open challenges and opportunities*. When statically instrumenting Web-

20 Both terms contrast WebAssembly with source languages like JavaScript. *Binary* stresses a bit more its low-level nature and the executable file format. *Bytecode* focuses more on its portability, the language, and that it executes on an abstract machine.

Assembly binaries, the rewriting needs to be type-aware, otherwise the resulting binaries are no longer valid. Other characteristics of WebAssembly also need to be taken into account, e.g., structured control-flow with relative branch labels. At the same time, WebAssembly is a saner format for binary analysis and rewriting than native machine code. Static disassembly of WebAssembly is robust, and built-in validation helps with the reliability of static rewriting. We will discuss challenges and opportunities around WebAssembly in more detail in Section 1.4.

Last but not least, we believe research should yield *practical insights, datasets, and (prototype) tools* that are useful to developers and other researchers to build on. Web developers so far are mostly used to high-level JavaScript programs. With WebAssembly, e.g., compiled from C and C++ code, they require new tooling and guidelines for dealing with memory vulnerabilities, or to make sense of a binary without source code. We discuss the conceptual contributions of our work in Section 1.5 and how the chapters of the dissertation map to publications and released artifacts in Section 1.6.

DISSERTATION GOAL In summary, the goal of this dissertation is to produce novel insights, datasets, and techniques to support developers in practical problems around understanding, optimizing, and improving the security and reliability of WebAssembly applications. The methods we employ come from program analysis, and include static and dynamic as well as classical and neural approaches. In this work, we focus on WebAssembly binaries and show that analysis at this level can be reliably and efficiently implemented.

1.4 CHALLENGES AND OPPORTUNITIES

As a novel language and bytecode format, analyzing WebAssembly comes with several high-level challenges and opportunities. We categorize them into five broad topics. In the later chapters we expand on them specifically for each project.

WEBASSEMBLY-SPECIFIC LANGUAGE FEATURES (C1) When analyzing and instrumenting WebAssembly bytecode, its unique language features need to be taken into account. For example, instead of jumps to absolute code addresses, WebAssembly has structured control-flow with nested code blocks and branches that identify their target by rel-

ative numerical labels. Other language features of interest are the implicit operand stack, type-polymorphic instructions, and linear memory. These need to be handled in both static and dynamic analysis. At the same time, some language features can also be seen as an opportunity, as they make WebAssembly more amenable to program analysis than, e.g., native code.

DATA AVAILABILITY (C2) To focus research on relevant issues, it is useful to look at data from real-world usage of the language and the ecosystem. Similarly, for machine learning, one typically requires a large set of labeled data for training. Finally, sets of real-world programs are useful to test analysis implementations. Unfortunately, given WebAssembly's young age, the research community has not yet collected such datasets, which requires us to do so.

SECURITY MODEL (C3) WebAssembly is often touted for its security. Indeed, some security aspects have been a concern early in the language design, e.g., ensuring that WebAssembly programs cannot access data beyond their own memory. However, these aspects focus mainly on one kind of threat, namely protecting the host from malicious WebAssembly binaries. Other aspects of WebAssembly's security model are understudied, e.g., how vulnerable WebAssembly programs are against attacks inside their own memory, and which exploits can result from that on different host environments.

LOW-LEVEL TYPES (C4) WebAssembly is statically typed, but it has only four low-level primitive types, namely for numbers. This impedes program understanding, for which more high-level types would be useful, e.g., pointers, arrays, or objects. Binary rewriting is also made more challenging by static types, as care must be taken that the resulting binaries are still type-correct. On the upside, built-in type checking is useful as a basic validation for binary rewriting, and low-level types can be fed as input features to machine learning models.

BINARY FORMAT (C5) The WebAssembly binary format is more low-level than source code. When we started our research, libraries for parsing and transforming its binaries were scarce, which prompted us to develop our own tooling. At the same time, WebAssembly offers more opportunities for binary analysis and static instrumentation than native code. It avoids problematic features, such as mixed code

and data, and can thus be reliably disassembled and instrumented. The well-specified binary format is what enables robust program analysis for WebAssembly at all.

1.5 OUTLINE AND CONTRIBUTIONS

We now outline how this dissertation is organized. The main content is in Chapters 2 to 7, which correspond to five research projects and publications (see Section 1.6). Of those chapters, Chapter 2 provides background on the WebAssembly language and ecosystem, drawing material and merging it from the mentioned publications. Chapters 4 to 7 then cover the individual projects in detail. After that, we discuss related work in Chapter 8 and end with conclusions, open questions, and an outlook into the future in Chapter 9.

At a high level, the main chapters can be grouped into two parts. In the first part (Chapters 2–4), we analyze fundamental language concepts, collect data about how WebAssembly is used in practice, and discuss the consequences of those findings, in particular for security. This part is more focused on understanding, conceptual findings, and generating insights. In the second part (Chapters 5–7), we present concrete program analysis and instrumentation techniques for WebAssembly binaries. This includes WASABI, a generic dynamic analysis framework, FUZZM, a binary-only fuzzer and binary hardening instrumentation, and SNOWWHITE, a neural type recovery approach.

In the following, we summarize the main contributions of this dissertation, grouped by the five projects it is based on. We also highlight how our work addresses the challenges identified in Section 1.4.

BINARY SECURITY Chapter 3 presents our work on the security of WebAssembly binaries. Protecting the host from malicious WebAssembly code has been an early goal in the language design, but it is less clear how secure WebAssembly programs themselves are from exploitation. We analyze the language and ecosystem, and find WebAssembly's linear memory, unmanaged data, the lack of compiler-inserted mitigations, and unsafe allocators to be problematic. The result is that vulnerabilities in memory-unsafe source languages can translate into vulnerable WebAssembly binaries. Surprisingly, while many of those classic vulnerabilities are no longer exploitable when compiling to native code, they are completely exposed when compiling to WebAssembly. More-

over, WebAssembly enables unique attacks, such as overwriting supposedly constant data or manipulating the heap using a stack overflow. Our findings contest prior claims that, e.g., compiler-inserted mitigations are unnecessary for WebAssembly. To demonstrate the severity of our findings, we also provide a library of attack primitives and build three exploit chains against vulnerable proof-of-concept applications. Depending on the host environment, these exploits lead to cross-site scripting in the browser, remote code execution on Node.js, and an arbitrary file write on stand-alone WebAssembly runtimes. We also discuss how the security of WebAssembly binaries could be improved with language changes, mitigations in the ecosystem, and concrete lessons to adopt for developers.

In terms of high-level challenges, this work focuses on the security model for WebAssembly (C3). We conclude that host security alone is not enough, and that there is a perhaps surprising lack of binary security in WebAssembly. We also analyze WebAssembly-specific language features (C1), such as linear memory, the difference between managed and unmanaged data, and compare the runtime type-checking of indirect calls to native control-flow integrity (CFI) schemes.

WASMBENCH Going beyond the manual analysis of the language and select examples in the previous chapter, more can be learned by looking at larger datasets of real-world programs. Unfortunately, such datasets do not exist for WebAssembly. Thus, Chapter 4 presents a dataset and study of 8,461 unique WebAssembly binaries that we gathered from a wide range of sources, including code repositories, package managers, and live websites. Through a combination of static analysis, statistics, and manual inspection, we study which source languages WebAssembly binaries are compiled from, what functions are commonly imported from the host environment, and how WebAssembly is used in the wild. For example, we find that two thirds of the binaries are compiled from C and C++. Consequently, the security issues discussed above potentially apply to a wide range of real-world code. Our findings also motivate further research and update previously held assumptions. Cryptomining, once a major use case for WebAssembly, has been marginalized, giving rise to more and more diverse, benign use cases. We also find that 29% of the binaries on the web lack useful information for program understanding, e.g., function names, which calls for techniques to decompile and reverse engineer WebAssembly.

With this work, we address the challenge of low data availability (C2) by collecting a dataset of WebAssembly binaries that is 58 times larger than the largest previous dataset. We make our dataset publicly available and use it ourselves in later research projects. When studying the dataset, several analyses focus again on security (C3), this time also measuring how many potentially dangerous imports there are, and which allocators are used in practice.

WASABI In Chapter 5, we present WASABI, the first general-purpose framework for dynamically analyzing WebAssembly. As discussed in Section 1.2, dynamic analysis can be vital to improve the performance, security, and reliability of applications. However, building such tools from scratch requires knowledge of low-level details of the language and its runtime, and manual modification of binaries is laborious and error-prone. Instead, WASABI provides an easy-to-use, high-level API that allows analyses to observe any instruction at runtime, with all its inputs and outputs. As the implementation strategy, WASABI employs static binary instrumentation, which automatically inserts calls to analysis hooks written in JavaScript into a WebAssembly binary. Our evaluation on benchmarks and real-world applications shows that WASABI (i) faithfully preserves the original program behavior, (ii) imposes an overhead that is reasonable for heavyweight dynamic analysis, and (iii) makes it straightforward to implement various dynamic analyses, including instruction counting, call graph extraction, memory access tracing, and taint analysis.

Binary instrumentation for WebAssembly faces several challenges due to static types (C4) and other language features, e.g., nested code blocks (C1). Instrumentation must be intertwined with type checking to handle type-polymorphic instructions. While instructions can have polymorphic type, functions in WebAssembly cannot. We thus devise on-demand monomorphization for the inserted analysis hooks. The lack of existing libraries (C5), also prompted us to develop our own parser and binary rewriter, on which we build in follow-up work.

FUZZM Chapter 6 presents FUZZM, the first binary-only fuzzer for WebAssembly. As discussed in Chapters 3 and 4, WebAssembly binaries are often compiled from memory-unsafe languages, causing exploitable vulnerabilities. This project addresses the problem of detecting and preventing such vulnerabilities. FUZZM consists of two parts.

First, it instruments WebAssembly binaries to detect spatial memory errors on the stack and heap at runtime. Additionally, it instruments the binaries to collect approximate coverage information as feedback for the fuzzer. Second, we efficiently combine the native fuzzer *AFL* and a WebAssembly VM running the instrumented program. We evaluate FUZZM with 28 real-world WebAssembly binaries, some compiled from well-known software projects, others found in the wild without access to their source code. Even though the programs run in a VM, FUZZM's performance is close to native AFL in terms of executions per second, generated inputs, and triggered crashes. Besides as an oracle for fuzzing, the instrumentation applied by FUZZM also serves as a stand-alone hardening technique to prevent exploitation of vulnerable binaries. It effectively prevents the exploits from Chapter 3 while imposing only between 2% to 35% runtime overhead.

This work focuses on vulnerable WebAssembly applications and how to prevent attacks against them (C3). It reuses code for static instrumentation from our earlier work on WASABI. WebAssembly's binary format (C5) makes it possible in the first place to reliably instrument production binaries without access to their source code.

SNOWWHITE As a low-level binary format, WebAssembly is less accessible to human inspection than source code, requiring laborious reverse engineering. An important first step when reverse engineering binaries is to recover the types of functions. Thus, in Chapter 7 we present SNOWWHITE, a neural approach for recovering high-level types from WebAssembly binaries. In contrast to prior learning-based type recovery for other binary formats, SNOWWHITE represents the types-to-predict in an expressive type language. It can describe a large number of complex types instead of the fixed, and usually small type vocabulary used previously. We formulate the type recovery as a sequence prediction task and build on the success of neural sequence-to-sequence models. We evaluate SNOWWHITE on a large-scale dataset of 6.3 million type samples extracted from over 300,000 WebAssembly object files. The results show that the type language is expressive, precisely describing 1,225 types instead the 7 to 35 types considered previously. Despite this expressiveness, the type prediction has high accuracy, exactly predicting 44.5% (75.2%) of all parameter types and 57.7% (80.5%) of all return types within the top-1 (top-5) predictions.

As a machine learning approach, SnowWhite suffers from the lack of available training data for WebAssembly (C2). This prompts us to collect the largest dataset of WebAssembly binaries with debug information, which is even larger than prior multi-architecture datasets for native code. We also address the challenge of WebAssembly types being fairly low-level (C4), while at the same time benefiting from them being available as input features for our model.

1.6 PUBLICATIONS AND OTHER ARTIFACTS

This dissertation is based on five research projects, four of which have previously appeared in peer-reviewed publications. The dissertation verbatim reuses material from those publications. We have also released all source code and datasets under a permissive license, to foster independent replication of our results, enable others to build on our work, and make our tools available to practitioners. The mapping from the main chapters of this dissertation to prior publications and released artifacts is as follows.

- **Chapter 3**: Everything Old is New Again: Binary Security of WebAssembly [Lehmann, Kinder, et al. 2020]. Daniel Lehmann, Johannes Kinder, and Michael Pradel. USENIX Security Symposium 2020. Exploits, dataset, and analysis code: `https://github.com/sola-st/wasm-binary-security`.
- **Chapter 4**: An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases [Hilbig et al. 2021]. Aaron Hilbig, Daniel Lehmann, and Michael Pradel. The Web Conference 2021 (www '21). Dataset, collection, and analysis code: `https://github.com/sola-st/WasmBench`.
- **Chapter 5**: Wasabi: A Framework for Dynamically Analyzing WebAssembly [Lehmann and Pradel 2019]. Daniel Lehmann and Michael Pradel. International Conference on Architectural Support for Programming Languages and Operating Systems 2019 (ASPLOS 2019). **Won a best paper award**. Project website with demo and tutorial material: `http://wasabi.software-lab.org`. Source code: `https://github.com/danleh/wasabi`.
- **Chapter 6**: Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly [Lehmann, Torp, et al.

2021]. Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. Source code and dataset: https://github.com/fuzzm/fuzzm-project.

- **Chapter 7**: Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries [Lehmann and Pradel 2022]. Daniel Lehmann and Michael Pradel. International Conference on Programming Language Design and Implementation 2022 (PLDI '22). Source code and dataset: https://github.com/sola-st/wasm-type-prediction.

During the PhD, the author also worked on the following research projects and publications that are not included in this dissertation.

- Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly [Romano, Lehmann, et al. 2022]. Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. IEEE Symposium on Security and Privacy 2022 (S&P '22). https://github.com/js2wasm-obfuscator/translator.
- Differential Regression Testing for REST APIs [Godefroid et al. 2020]. Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. International Symposium on Software Testing and Analysis 2020 (ISSTA 2020). https://github.com/microsoft/restler-fuzzer.
- Interactive Metamorphic Testing of Debuggers [Tolksdorf et al. 2019]. Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. International Symposium on Software Testing and Analysis 2019 (ISSTA 2019).
- Feedback-Directed Differential Testing of Interactive Debuggers [Lehmann and Pradel 2018]. Daniel Lehmann and Michael Pradel. Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering 2018 (ESEC/FSE 2018). https://github.com/sola-da/DifferentialDebuggerTesting.

2 BACKGROUND ON WEBASSEMBLY

We took a first peek at the WebAssembly language and its ecosystem in Section 1.1. To have sufficient background for later chapters, a more thorough introduction is in order. The first interaction with any language is through its syntax, so we discuss the text and binary formats in Section 2.1. Both formats represent the same fundamental language concepts, of which we give a tour in Section 2.2. Finally, we discuss the surrounding ecosystem of compilers, source languages, host environments, and WebAssembly runtimes in Section 2.3.

We cannot fully cover the language and ecosystem here. Instead, we focus on the most relevant aspects for later chapters. For more information, we refer the interested reader to the following resources:

- The official website [WebAssembly Website] provides a good overview and high-level intuitions. It also links to essential tools, answers frequently asked questions, and discusses the future roadmap.
- The official language specification [WebAssembly Specification] is the authoritative, technical source for exact details of the binary and text formats, execution semantics, and all language constructs.
- The documentation on MDN[1] focuses on practical information for Web developers working *with* WebAssembly (but not necessarily working *on* WebAssembly itself).
- The WebAssembly GitHub organization and its repositories[2] contain design documents and language proposals, meeting notes of the standards committee, and the source code for the reference interpreter, many fundamental tools, and other associated infrastructure.
- Finally, we also refer to the initial publication [Haas et al. 2017] that introduced WebAssembly to the academic community, to the up-

1 Formerly named the *Mozilla Developer Network*, https://developer.mozilla.org/en-US/docs/WebAssembly

2 https://github.com/webassembly

dated, more concise version of the latter [Rossberg et al. 2018], and to other publications on WebAssembly in Chapter 8.

2.1 BINARY AND TEXT FORMAT

Listing 1.1 shows a WebAssembly program side-by-side in the binary and the human-readable text format. Although neither of them is the focus of our research, it is important to discuss them in order to understand later code examples and particularities of the language. We start with the binary format, as it is the typical representation of WebAssembly programs stored on disk and sent over the network. The binary format is also what our analysis tools in Chapters 3 to 7 take as input.

BINARY FORMAT The binary format is designed to have the following properties. First, it is *compact*. Instructions have a single-byte opcode (e.g., `7C` for the `i64.add` instruction); more bytes can follow for immediate arguments (e.g., in `i32.const` *c*, the immediate *c* is the constant pushed on the stack). All integers are encoded in the variable-length LEB128 format (Little Endian Base 128), which is also known from DWARF debugging information [DWARF 5 Standard]. In LEB128, the most significant bit of each byte marks whether more bytes follow. E.g., the integer 7 is encoded as a single byte `07`, whereas the integer 1337 is encoded as two bytes `B9 A0`. Program elements (types, functions, variables, etc.) are identified in the binary by integer *indices*. E.g., the function declared in line 31 of Listing 1.1 is identified by the function index 1, as evident by the second byte of the binary encoding of the call in line 40. Strings and names are only present in the binary when importing or exporting program elements, in the data section for initializing memory, or in (optional) debug information.

Second, the binary format is *easy* and *reliable* to decode and validate. We use the terms *decoding*, *parsing*, or *disassembling* a WebAssembly binary interchangeably. Parsing via recursive descent is simple and does not require lookahead. There is no ambiguity as to how the current input shall be parsed, unlike in native binaries, where static disassembly and even identification of function boundaries can be challenging [Andriesse et al. 2016]. WebAssembly also strictly separates code and data, and there is no self-modifying code. The binary format is promised to be backwards compatible in case of future additions to the language.

Third, parsing the binary format is *fast*, and can be done in a *parallel* and *streaming* fashion. A binary contains multiple *sections*, e.g., the type section starting in line 6 or the code section starting in line 30 of Listing 1.1. Each section declares its length in bytes upfront, such that an optimized parser can jump ahead to parse subsequent sections in parallel. The order of sections is such that information required for validation and compilation is available early. E.g., the type section comes in the very beginning of the binary and the section declaring all functions comes before the code section containing the function bodies. A streaming baseline compiler [Backes 2018] can start generating native code from the binary even when it is not fully resident in memory yet, e.g., while still downloading the binary from the Internet.

The most important command-line tools for inspecting and manipulating binaries are Binaryen[3] and the WebAssembly Binary Toolkit (WABT).[4] There are also libraries for working with WebAssembly binaries in different programming languages,[5] although many of those did not exist when we started our research in early 2018.

TEXT FORMAT WebAssembly binaries (`.wasm`) can be converted into and generated from the human-readable text format (`.wat`) with the `wasm2wat` and `wat2wasm` programs, respectively. In practice, WebAssembly binaries are frequently compiled from other languages (see Section 2.3), but small programs can also be written manually. We will use the text format for code examples, even if we actually take the binary format as input. As evident from Listing 1.1, the text format is based on *S-expressions*, i.e., parenthesized n-ary trees. This makes parsing the text format simple and unambiguous. The program elements in the text format correspond closely to the abstract language constructs, so we will discuss them in more detail in Section 2.2.

The text format abstracts over some low-level details of the binary format, similar to how assembly languages add convenience over writing machine code directly. Instead of referring to functions, variables, or types by their numerical index only (e.g., `call 1` would be valid syntax for calling the function with index 1), indices can also be written as `$name`. The name is simply converted to an integer in the binary for-

3 https://github.com/WebAssembly/binaryen
4 https://github.com/WebAssembly/wabt
5 E.g., https://github.com/bytecodealliance/wasm-tools, https://github.com/xtuc/webassemblyjs, https://github.com/wasdk/wasmparser

mat. Furthermore, elements in the text format need not be laid out in the same order as in the binary format. E.g., functions can be written with their type inlined into the declaration, such as

```
(func $foo (type (param i32) (return i64)) ...)
```

and `wat2wasm` takes care of generating the necessary entries in the type section of the binary. Similarly, import and export names can be directly attached to declarations. Finally, the text format supports comments, which are lost in the binary.

2.2 LANGUAGE CONCEPTS

Regardless of the representation, programs use the same fundamental language concepts, which we discuss in the following.

MODULE STRUCTURE Figure 2.1 gives a simplified abstract syntax for WebAssembly binaries. One WebAssembly binary corresponds to one *module* with several elements. A module contains multiple *functions*, multiple *globals* (global variables), at most one *table* (used in indirect calls) and at most one *memory*. Functions, globals, tables, and memories can be imported from the host environment (as in the example of Listing 1.1), or locally defined in the module. They can also be exported under (potentially) multiple names. Locally defined functions have a function body in the code section, and locally defined globals have an initialization expression. Functions and globals are ordered. Their implicit *index* uniquely identifies them.

STACK AND VARIABLES As evident from the grammar in Figure 2.1, instructions in WebAssembly do not take explicit operands or registers as arguments. Instead, they execute on an abstract *stack machine*. Instructions pop their inputs from an implicit *operand stack* and push their results to the stack again. To store common subexpressions, for mutable data, and to duplicate stack values, WebAssembly also features an unlimited number of *locals* (local variables). Those need to be declared in the beginning of a function body (see *code* in Figure 2.1).

The `local.set` and `global.set` instructions pop a value from the stack and write it to the corresponding variable. Conversely, `local.get` and `global.get` push the current value of a variable onto the stack and `local.tee` writes a local, but keeps the value on the stack. A function can only access its own local variables and its own operand stack. Only

$$
\begin{aligned}
\mathit{module} &:= \texttt{module}\ \mathit{function}^*\ \mathit{global}^*\ \mathit{table}^?\ \mathit{memory}^? && \text{(modules)}\\
\\
\mathit{function} &:= \texttt{func}\ \mathit{type}_{\mathit{func}}\ (\mathit{import} \mid \mathit{code})\ \mathit{export}^*\\
\mathit{global} &:= \texttt{global}\ \mathit{type}_{\mathit{val}}\ (\mathit{import} \mid \mathit{init})\ \mathit{export}^*\\
\mathit{table} &:= \texttt{table}\ \mathit{import}^?\ \mathit{idx}_{\mathit{func}}{}^*\ \mathit{export}^*\\
\mathit{memory} &:= \texttt{memory}\ \mathit{import}^?\ \mathit{byte}^*\ \mathit{export}^*\\
\\
\mathit{import} &:= \texttt{import}\ (\texttt{"module name"}, \texttt{"name"})\\
\mathit{export} &:= \texttt{export}\ \texttt{"name"}\\
\mathit{code} &:= (\texttt{local}\ \mathit{type}_{\mathit{val}})^*\ \mathit{instr}^*\\
\mathit{init} &:= \mathit{instr}^*\\
\\
\mathit{instr} &:= \texttt{nop} \mid \texttt{unreachable} && \text{(instructions)}\\
&\quad \mid \mathit{type}_{\mathit{val}}\texttt{.const}\ \mathit{value}\\
&\quad \mid \mathit{unary} \mid \mathit{binary}\\
&\quad \mid \texttt{local.}(\texttt{set} \mid \texttt{get} \mid \texttt{tee})\ \mathit{idx}_{\mathit{local}} \mid \texttt{global.}(\texttt{set} \mid \texttt{get})\ \mathit{idx}_{\mathit{global}}\\
&\quad \mid \texttt{drop} \mid \texttt{select}\\
&\quad \mid \mathit{type}_{\mathit{val}}\texttt{.load} \mid \mathit{type}_{\mathit{val}}\texttt{.store} \mid \texttt{memory.size} \mid \texttt{memory.grow}\\
&\quad \mid \texttt{block}\ \mathit{instr}^*\ \texttt{end} \mid \texttt{loop}\ \mathit{instr}^*\ \texttt{end} \mid \texttt{if}\ \mathit{instr}^*\ \texttt{else}\ \mathit{instr}^*\ \texttt{end}\\
&\quad \mid \texttt{br}\ \mathit{label} \mid \texttt{br\_if}\ \mathit{label} \mid \texttt{br\_table}\ \mathit{label}^*\ \mathit{label}\\
&\quad \mid \texttt{call}\ \mathit{idx}_{\mathit{func}} \mid \texttt{call\_indirect}\ \mathit{type}_{\mathit{func}} \mid \texttt{return}\\
\mathit{unary} &:= \texttt{i32.eqz} \mid \ldots \mid \texttt{f32.neg} \mid \ldots \mid \texttt{f32.convert\_s/i32} \mid \ldots\\
\mathit{binary} &:= \texttt{i32.add} \mid \ldots \mid \texttt{i32.eq} \mid \ldots\\
\\
\mathit{type}_{\mathit{val}} &:= \texttt{i32} \mid \texttt{i64} \mid \texttt{f32} \mid \texttt{f64} && \text{(types, labels, indices)}\\
\mathit{type}_{\mathit{func}} &:= [\mathit{type}_{\mathit{val}}{}^*] \to [\mathit{type}_{\mathit{val}}{}^?]\\
\mathit{label} &\in \mathbb{N}\\
\mathit{idx}_{\mathit{func} \mid \mathit{global} \mid \mathit{local}} &\in \mathbb{N}
\end{aligned}
$$

Figure 2.1: A (simplified) abstract syntax for WebAssembly modules.

```
1 i32.const 42  ;; Simple type: [] → [i32]
2 local.get $x  ;; Type depends on context, here on type of local $x.
3 call $f       ;; Type depends on context, here on type of function $f.
4 drop          ;; Polymorphic type [τ] → [], where τ depends on the type
5               ;; at the top of the abstract type stack at this point.
```

Listing 2.1: Example for type checking WebAssembly instructions, including polymorphic ones.

global variables (and memory) are shared across functions. The evaluation stack, locals, and globals are managed by the WebAssembly VM.

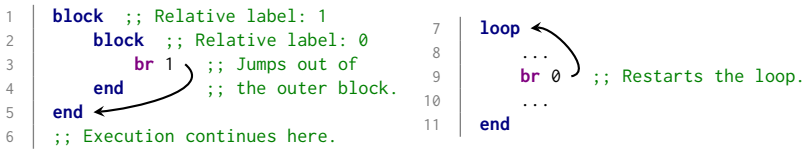
TYPES Every function, instruction, and variable in WebAssembly is statically typed. There are only four primitive *value types*: 32 and 64-bit integers (`i32`/`i64`) and single and double precision IEEE 754 floating-point numbers (`f32`/`f64`). In particular, there are no classes, objects, arrays, or other aggregate types. Source-level types are thus lowered to primitive types (and data in memory) during compilation to WebAssembly. Functions and instructions have a *function type*, with an unlimited number of inputs and up to one result.[6]

Sequences of instructions are type checked by statically "evaluating" them with an abstract type stack in place of the runtime operand stack. Listing 2.1 shows an example. Most instructions have a fixed type (line 1). Some instructions require a context for type checking, e.g., the type of other functions or variables (lines 2 and 3). Few instructions have a polymorphic type, namely `drop` (which removes the current stack top), and `select` (which pushes one of two values depending on a condition). Their type does not depend on a fixed context, but on the current type stack, i.e., the previously executed instructions (line 4).

CONTROL-FLOW WebAssembly uses *structured control-flow*, which is unusual for a low-level bytecode. The grammar in Figure 2.1 shows that `block`, `loop`, `if`, and `else` organize instructions into *well-nested blocks*, terminated by a matching `end` instruction. Blocks may be nested arbitrarily deep. The left side of Listing 2.2 gives an example.

Branch instructions (`br`, `br_if`, and `br_table`) can only target blocks in which they are enclosed. The branch target is encoded as a relative block *label*. This label is a non-negative integer, where 0 indicates

6 A later language extension allows an unlimited number of results as well. We focus on the initial version of WebAssembly here.



Listing 2.2: Nested blocks and structured control-flow in WebAssembly.

that the branch is targeting the immediately enclosing block, 1 is that block's parent block, and so on. Depending on the target block type, the branch either jumps out of the block (thus a forward jump, for `block` and `if`/`else`), or restarts the block (for `loops`, thus a backward jump). Listing 2.2 illustrates the difference with two examples.

For conditional control-flow, `br_if` l pops an `i32` condition c from the stack and jumps to target l if c is non-zero. The `if`/`else` blocks add no expressiveness, but can sometimes encode the same behavior more compactly. Multi-way branches are supported via `br_table` $l_0 \ldots l_n, d$. It pops an `i32` from the stack and uses it as an index i into its branch table, which is a list of l_i labels that is statically encoded into the instruction. If the index is out of bounds (i.e., $i > n$), it jumps to the default label d.

Comparing WebAssembly's control-flow with native code, there are two notable restrictions. First, all branch targets are statically declared. In particular, there are no unrestricted indirect jumps, such as `jmp *%reg` in x86, which limits abuse by runtime attacks. Second, nested blocks and relative branch labels rule out irreducible control-flow, i.e., jumps into the middle of blocks or loops. For most programs in most source languages, this is not a limitation. If programs have irreducible control-flow (e.g., using `goto`), it can always be encoded as structured control-flow, e.g., with the Relooper algorithm [Zakai 2011].

FUNCTION CALLS The `call` instruction implements direct calls. The instruction has the same type as the called function. It pops the argument values from the caller's evaluation stack, executes the function, and pushes the result onto the caller's stack again. Inside a function, the n arguments are available as local variables with indices $0 \ldots n-1$. This allows pushing arguments multiple times onto the stack via `local.get`. At the end of a function, the last value on the stack is returned implicitly. There is also an explicit `return` instruction, e.g., for early returns.

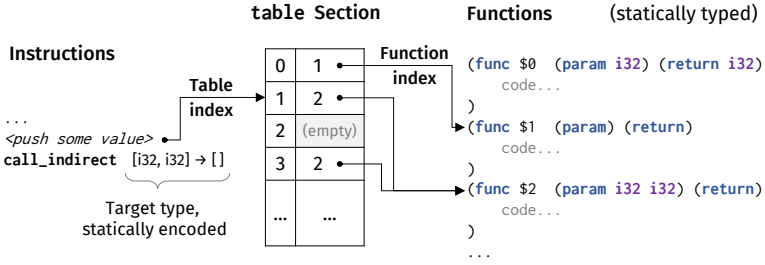


Figure 2.2: Indirect function calls via the module's table.

The `call_indirect` instruction is used to implement runtime dispatch, e.g., for function pointers or virtual functions. Figure 2.2 illustrates how it works. The instruction pops an `i32` value from the stack, which it uses to index into the *table* that is part of the module (cf. Figure 2.1). The table maps the index to a function, which is subsequently called. Functions can be in the table multiple times and not every table entry must be filled. Besides the table index, the `call_indirect` instruction also pops the function arguments from the stack, like a regular call. The VM checks at runtime that the target function is type-compatible with the function type that is statically encoded into the `call_indirect` instruction. If not, execution fails with a runtime type error.

UNMANAGED, LINEAR MEMORY For storing long-lived and aggregate data, each WebAssembly module has at most one *linear memory*. It can simply be thought of as a global array of consecutive bytes. Unlike memory in other bytecode languages, there is no garbage collection provided by the VM and the memory is under complete control of the program, which is why we call it *unmanaged*. The memory is zero-initialized by default; segments of it can also be initialized at program startup with byte sequences from the *data* section, like so:

```
;; Explicitly initialized memory at offset 1024.
(data (i32.const 1024) "a null-terminated string\00")
```

The memory can be grown at runtime in page-size increments (64 KiB) using the `memory.grow` instruction, and its current size can be queried with `memory.size`. For efficient dynamic allocation, a WebAssembly binary typically includes its own allocator code, which provides functions such as `malloc` and `free` to the rest of the program.

The memory spans a 32-bit address space, and `i32` serves as the pointer type.[7] While the memory contents are just untyped bytes, *t*.`load` and *t*.`store` instructions are typed. E.g., an `f32.load` instruction takes an `i32` value from the operand stack, reads four bytes from memory at that address, and pushes the result as an `f32` value onto the stack again. If the address is larger than the current size of the memory, execution fails with a runtime error.

It should not be possible to access data in the underlying native system memory from inside a WebAssembly program, except for the region that is allotted for linear memory by the VM. VMs implement this via bounds checking or (on 64-bit architectures) by installing guard pages above and below the linear memory region. This makes WebAssembly a good technology for software fault isolation (SFI) and memory sandboxing [Narayan, Disselkoen, Garfinkel, et al. 2020; Zakai 2020]. In Chapter 3, we analyze the implications of linear memory for the security of WebAssembly programs themselves.

LANGUAGE EXTENSIONS What we have described so far is version 1.0 of WebAssembly, often referred to as the *MVP* (Minimum Viable Product). Since its standardization in 2019 [WebAssembly Specification], several extensions have been proposed and some of them recently standardized.[8] For a proposal to become standardized, among other requirements, it must have a formal specification and at least two independent, production-quality implementations, e.g., in two different browser engines.[9] Even when standardized, extensions are optional and not all runtimes support them. For that reason, most compilers generate binaries that do not use extensions by default, and applications that do use them, may provide a fallback (sometimes at the cost of lower performance).

There are extensions that add only a small number of instructions, such as additional float-to-integer conversions,[10] but also more extensive changes, e.g., adding vector types and hundreds of SIMD operations,[11] which can be very useful for high-performance signal process-

7 There is a proposal for a 64-bit address space language extension, but it is not yet standardized or widely supported. See https://github.com/WebAssembly/memory64.
8 https://webassembly.org/roadmap/, https://github.com/WebAssembly/proposals
9 https://github.com/WebAssembly/meetings/blob/main/process/phases.md
10 https://github.com/WebAssembly/spec/blob/master/proposals/nontrapping-float-to-int-conversion/Overview.md
11 https://github.com/WebAssembly/simd

ing. Other proposals strive to remove language restrictions, such as allowing multiple return values and 64-bit memory addresses.[12]

Different research projects in this dissertation handle extensions differently. When we developed WASABI (Chapter 5), no extensions were standardized yet, so it assumes WebAssembly binaries of version 1.0. We do not think there are principled hindrances to adapting it for the currently standardized extensions, beyond the engineering effort required. As FUZZM (Chapter 6) builds on the same infrastructure, it inherits this limitation. On the other hand, our study and the dataset in Chapter 4 contains several binaries with extensions. Our work on the binary security of WebAssembly (Chapter 3) is orthogonal to most language extensions, and we discuss the potential positive impact of extensions to mitigate our attacks in Section 3.6. Finally, our latest project, SNOWWHITE in Chapter 7, processes its input data with `wasmparser`,[13] which can handle all currently standardized language extensions.

2.3 ECOSYSTEM

Besides the language, there is also an ecosystem of related technologies around WebAssembly. For example, Figure 1.1 shows different source languages that compile to WebAssembly and multiple host environments, where WebAssembly binaries can be run. We discuss this ecosystem in the following.

HOST ENVIRONMENT WebAssembly modules are executed in a *host environment*, such as the browser, Node.js, or a stand-alone WebAssembly runtime. The host is responsible for *instantiating* the module, that is, providing imported functions and setting up execution of the module. In browsers, JavaScript code can instantiate and run a WebAssembly binary through the `WebAssembly.instantiate` function and related APIs.[14] Exported elements of the module can be accessed from the host, allowing, e.g., JavaScript code to call exported WebAssembly functions.

Because WebAssembly modules share nothing by default and WebAssembly has no standard library, a WebAssembly program without a host environment cannot perform I/O or access the network. Such

12 Both extensions were mentioned earlier in this section.
13 https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasmparser
14 https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API

functionality is provided through imported host functions. The module's memory can also be imported or exported to allow shared access with the host. In the browser, any JavaScript function can be imported into a WebAssembly module, including client-side APIs such as `XmlHttpRequest`, `document.write`, or `eval`. Other host environments provide other APIs to WebAssembly modules, e.g., modules running in Node.js could import `exec` to execute shell commands, and modules running on a stand-alone VM may import functions for file system access. Because WebAssembly supports only number types, non-primitve data, e.g., strings or objects, may require marshalling through memory and then passing around pointers as numbers.[15]

WASI In principle, every host environment can provide its own set of functions to WebAssembly modules. However, to make not only the WebAssembly bytecode portable, but also the binaries portable across different host environments, there is an ongoing effort to standardize on the interface between the host environment and the module. The WebAssembly System Interface (WASI) specifies several basic APIs [Clark 2019; WASI Website], similar to a syscall interface for an operating system or POSIX. This includes functions and datatypes for filesystem access, network sockets, clocks, random number generation, etc.

Compilers and source languages targeting WebAssembly can then generate binaries with imports conforming to this interface. E.g., there is a (partial) `libc` implementation on top of WASI. On the runtime side, there is experimental WASI support in Node.js[16] and in several standalone runtimes, such as Wasmtime, Wasmer, and WAVM.[17] Our FUZZM work in Chapter 6 assumes binaries that use WASI, to execute them in the Wasmtime runtime that we integrated into AFL.

COMPILERS AND SOURCE LANGUAGES Its low-level instructions and unmanaged memory make WebAssembly a good compilation target from systems languages, such as C, C++, and Rust. Indeed, the first compiler for WebAssembly was *Emscripten*,[18] which is based on Clang and compiles C and C++ code for running it in browsers. Originally, Emscripten compiled to asm.js, which heavily influenced WebAssem-

15 A language extension adds first-class *reference types* to WebAssembly, which allows to pass objects between the host and the WebAssembly module. See https://github.com/WebAssembly/reference-types.

16 https://nodejs.org/api/wasi.html

17 [Wasmtime Website], https://wasmer.io, https://wavm.github.io

18 [Zakai 2011], https://emscripten.org

bly and subsequent got superseded by it as outlined in the introduction. In addition to compiling the source program, Emscripten also generates JavaScript code, e.g., for emulating a filesystem in memory, or making `printf` in C output to the browser console. Emscripten can compile large, complex C and C++ projects to WebAssembly, including the applications mentioned in the introduction.

Various WebAssembly compilers from other source languages exist as well, e.g., the Rust compiler,[19] Go,[20] AssemblyScript, which is a subset of TypeScript,[21] or the Asterius compiler for Haskell. Another strategy is to compile language runtimes to WebAssembly, e.g., Pyodide is a port of CPython to WebAssembly,[22] and Microsoft's Blazor allows to run CIL (i.e., .NET code) in browsers with WebAssembly.[23]

19 https://www.rust-lang.org/what/wasm
20 https://tinygo.org/docs/guides/webassembly/
21 https://www.assemblyscript.org
22 https://pyodide.org, https://github.com/pyodide/pyodide
23 https://docs.microsoft.com/de-de/aspnet/core/blazor/

3 ON THE BINARY SECURITY OF WEBASSEMBLY

In the previous chapter (Chapter 2), we have introduced WebAssembly. It has some notable differences to native code, e.g., structured control-flow and sandboxed memory, often with the explicit goal of protecting the host against malicious WebAssembly binaries. However, an equally important aspect of WebAssembly security is less understood: Are WebAssembly binaries themselves secure against runtime attacks? Many WebAssembly programs are compiled from memory-unsafe languages (we will give a more detailed account in Chapter 4). Can vulnerabilities in those unsafe source languages translate to vulnerable WebAssembly binaries? If yes, how can they be exploited?

In this chapter, we answer those questions and compare how WebAssembly stacks up against native code. We find that many classic vulnerabilities that, due to common mitigations, are no longer exploitable in native binaries, are completely exposed in WebAssembly. Moreover, WebAssembly enables unique attacks, such as overwriting supposedly constant data or manipulating the heap using a stack overflow. We present a set of attack primitives for overwriting sensitive data in memory and triggering unexpected behavior by diverting control flow or manipulating the host environment. We also provide a set of vulnerable proof-of-concept applications on three different WebAssembly host environments (browsers, Node.js, and a stand-alone VM), and attack them with end-to-end exploits. An empirical risk assessment on real-world binaries and SPEC CPU programs compiled to WebAssembly shows that our attack primitives are likely to be feasible in practice. Overall, our findings show a perhaps surprising lack of binary security in WebAssembly. On a positive note, we discuss potential protection mechanisms to mitigate the resulting risks.

This chapter shares large parts of its material with the corresponding publication [Lehmann, Kinder, et al. 2020]. The author of this dissertation is also the main author of that paper and did all of the implementation, evaluation, and the majority of the writing.

3.1 MOTIVATION AND CONTRIBUTIONS

WebAssembly is often touted for its safety and security. For example, both the initial publication [Haas et al. 2017] and the official website [WebAssembly Website] highlight security on the first page. Indeed, in WebAssembly's core application domains, security is paramount: on the client side, users run untrusted code from websites in their browser; on the server side in Node.js, WebAssembly modules operate on untrusted inputs from clients; in cloud computing, providers run untrusted code from users; and in smart contracts, programs may handle large sums of money.

There are two main aspects to the security of the WebAssembly ecosystem: (i) *host security*, the effectiveness of the runtime environment in protecting the host system against malicious WebAssembly code; and (ii) *binary security*, the effectiveness of the built-in fault isolation mechanisms in preventing exploitation of otherwise benign WebAssembly code. Attacks against host security rely on implementation bugs [Plaskett et al. 2018; Silvanovich 2018] and therefore are typically specific to a given WebAssembly VM. Attacks against binary security – the focus of this chapter – are specific to each WebAssembly program and its compiler toolchain. The design of WebAssembly includes various features to ensure binary security. For example, the memory maintained by a WebAssembly program is separated from its code, the evaluation stack, and the data structures of the executing VM. To prevent type-related crashes and attacks, binaries are designed to be easily type-checked, which they are statically before execution. Moreover, WebAssembly programs can only jump to designated code locations, a form of fault isolation that prevents many classic control-flow attacks.

Despite all these features, the fact that WebAssembly is designed as a compilation target for languages with manual memory management, such as C and C++, raises a question: *To what extent do memory vulnerabilities affect the security of WebAssembly binaries?* The original WebAssembly paper addresses this question briefly by saying that "at worst, a buggy or exploited WebAssembly program can make a mess of the data in its own memory" [Haas et al. 2017]. A WebAssembly design document on security[1] concludes: "common mitigations such as data

1 https://github.com/WebAssembly/design/blob/master/Security.md#memory-safety

execution prevention (DEP) and stack smashing protection (SSP) are not needed by WebAssembly programs".

This chapter analyzes to what extent WebAssembly binaries can be exploited and demonstrates that the above answers miss important security risks. Comparing the exploitability of WebAssembly binaries with native binaries, e.g., on x86, shows that WebAssembly re-enables several formerly defeated attacks because it lacks modern mitigations. One example are stack-based buffer overflows. They are effective again, because WebAssembly binaries do not deploy stack canaries. Moreover, we find attacks not possible in this form in native binaries, such as overwriting string literals in supposedly constant memory. If such manipulated data is later interpreted by critical host functions, e.g., as JavaScript code, this can lead to further system compromise. Our work mostly focuses on binaries compiled with LLVM-based compilers, such as Emscripten and Clang for C and C++ code, or the Rust compiler, since they are currently the most popular compilers targeting WebAssembly, as we will show in Chapter 4.

After our analysis of the deployed (and missing) security features in WebAssembly, we take the position of an active adversary and identify a set of attack primitives that can be used to build end-to-end exploits. Our attack primitives span three dimensions: (i) obtaining a write primitive, i.e., the ability to write memory locations in violation of source-level semantics; (ii) overwriting security-relevant data, e.g., constants or data on the stack and heap; and (iii) triggering a malicious action by diverging control flow or manipulating the host environment. Figure 3.1 provides an overview of the attack primitives and (missing) defenses discussed for WebAssembly.

To show that our attack primitives are applicable in practice, we discuss a set of vulnerable example WebAssembly applications and demonstrate end-to-end exploits against each one of them. The attacked applications cover three different WebAssembly platforms: client-side web applications in browsers, server-side applications on Node.js, and applications for stand-alone WebAssembly VMs.

In our quantitative evaluation, we then estimate the feasibility of attacks against other binaries. We collect a set of binaries from real-world web applications and compiled from large C and C++ programs of the SPEC CPU benchmark suite. Regarding overwriting data, we find that one third of all functions make use of the unmanaged (and

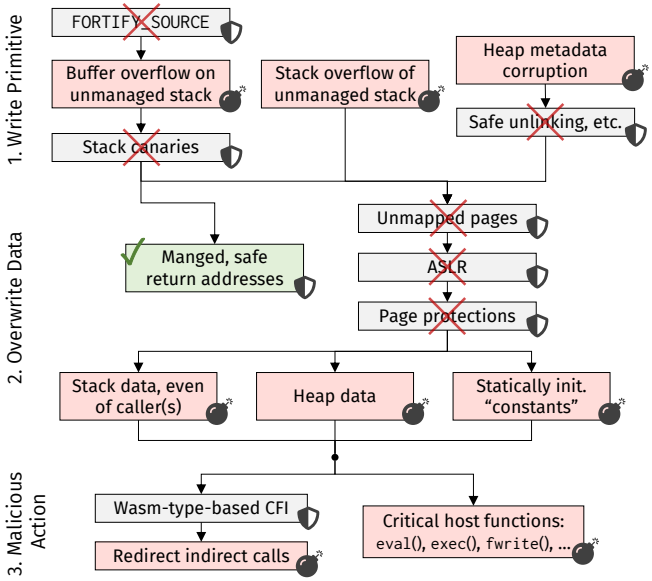


Figure 3.1: An overview of attack primitives (💣) and (missing) defenses (🛡) in WebAssembly, later detailed in this chapter.

unprotected) stack in linear memory. Regarding control-flow attacks, we find that every second function can be reached from indirect calls that take their target directly from linear memory. We also compare WebAssembly's runtime type-checking of indirect calls with control-flow integrity defenses for native code.

Our work improves upon initial discussions of WebAssembly binary security in the non-academic community [Bergbom 2018; Denis 2018; Foote 2018; McFadden et al. 2018] by providing a systematic analysis, a generalization of attacks, and data on real binaries (see Chapter 8 for a more detailed comparison).

CONTRIBUTIONS In summary, this chapter contributes:

- An in-depth *security analysis of WebAssembly's linear memory* and its use by programs compiled from languages such as C, C++, and Rust. We also analyze which common memory protections are missing from WebAssembly, and how this can make some code less secure than when compiled to a native binary (Section 3.2).

- A set of *attack primitives*, derived from our analysis and generalized from previous work, along with a discussion of mitigations that the WebAssembly ecosystem does, or does not, provide (Section 3.3).
- A set of *example vulnerable applications* and *end-to-end exploits*, which show the consequences of our attacks on three different WebAssembly platforms (Section 3.4).
- *Empirical evidence* that both data and control-flow attacks are likely to be feasible, measured on WebAssembly binaries from real-world web applications and compiled from large C and C++ programs (Section 3.5).
- A discussion of possible *mitigations* to harden WebAssembly binaries against the described attacks (Section 3.6). We make our attack primitives, end-to-end exploits, and analysis tool publicly available[2] to aid in this process.

3.2 SECURITY ANALYSIS OF LINEAR MEMORY

We now begin our security analysis of WebAssembly binaries and focus first on one of their key components: linear memory. We analyze how compilers arrange program data in linear memory and investigate how and which standard memory protection mechanisms are applied.

3.2.1 *Managed vs. Unmanaged Data*

We distinguish managed and unmanaged data in WebAssembly. *Managed* data, i.e., local variables, global variables, values on the evaluation stack, and return addresses, reside in dedicated storage handled directly by the VM. WebAssembly code can only interact with managed data implicitly through instructions, but not directly modify its underlying storage. E.g., `local.get 0` reads the local with index 0, but at no point is the actual, underlying address of the local visible to the program. *Unmanaged* data is all data that resides in linear memory. It is completely under the control of the program and typically organized by compiler-generated code.

There are several reasons for putting unmanaged data in linear memory. Since WebAssembly has only four primitive types and because

2 https://github.com/sola-st/wasm-binary-security

managed data can hold instances of only those primitive types, all non-scalar data, such as strings, arrays, or structs, must be stored in linear memory. Because managed data has no address, any variable whose address is ever taken in the source program, e.g., function out parameters, must also be stored in linear memory.

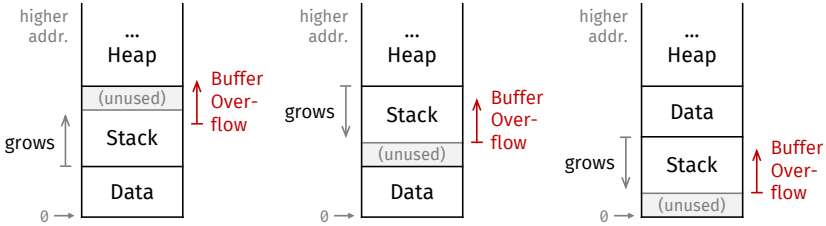
Because non-scalar data or data whose address is taken can appear in the source program as function-scoped, global data, or data with dynamic lifetime, the compiler sets aside distinct areas in linear memory for a stack, a heap, and static data. We will refer to the compiler-created stack in linear memory as the *unmanaged stack* to distinguish it from the managed evaluation stack, which holds intermediate values of instructions, and the managed call stack, which holds locals and return addresses.[3] Importantly, this means a lot of data lies in unmanaged linear memory, not under protection of the VM, but instead under full control of memory read and write instructions in the program.

3.2.2 *Memory Layout*

Native ELF binaries[4] contain sections for zero-initialized data (`.bss`), read- and writable data (`.data`), read-only data (`.rodata`), code (`.text`), a stack, and a heap. The compilers we analyze – Emscripten, Clang, and the Rust compiler – all perform a similar subdivision of the linear memory in WebAssembly binaries (Figure 3.2). The heap must always be placed at the end of linear memory, such that it can grow towards higher addresses and make use of additional memory when it is requested from the host environment. Below the heap are the stack and static data. Since there is no read-only memory in WebAssembly (more on that shortly), there is no distinction between `.data` and `.rodata`. Since linear memory is always zero-initialized, there is also no need for a dedicated `.bss` section. In other words, `.data`, `.rodata`, and `.bss` are not explicitly distinguished in WebAssembly. In the following, when we refer to the *data* section in linear memory, we mean all such data

3 This is a frequent point of confusion, so to reiterate, there are three distinct types of stacks in a WebAssembly program: The *evaluation stack* of the WebAssembly abstract machine, as introduced in Section 2.2, which is managed by the VM; the *managed call stack*, which contains the currently active functions, their locals, and return addresses, also managed by the VM; and the *unmanaged stack* in linear memory, for function-scoped unmanaged data. The latter is the security critical one in this work.

4 Other native binary formats, such as PE binaries on Windows, have analogous sections, but for readability we compare only with ELF here.



(a) `emcc` 1.39.7 (*fastcomp* backend, deprecated).

(b) `emcc` 1.39.7 (*upstream* backend), `clang` 9 (WASI).

(c) `clang` 9 (WASI with `stack-first`), `rustc` 1.41 (WASI).

Figure 3.2: Linear memory layouts for different compilers and backends.

that is valid for the whole lifetime of the program, e.g., statically initialized string constants, global arrays, or zero-byte ranges.

The memory layout, i.e., the order of stack, heap, and data in linear memory, depends on the compiler. Figure 3.2a shows that the *fastcomp* backend of Emscripten (the first WebAssembly backend and thus frequently used until its deprecation in October 2019[5]) places the static data at the beginning of linear memory, followed by the stack, and then the heap. The stack grows upwards (i.e., towards higher addresses) in this configuration. More recently, LLVM has gained its own, in-tree WebAssembly backend,[6] also called the *upstream* backend, which is used by Emscripten, Clang, and the Rust compiler. That is, in most WebAssembly binaries produced today, the stack grows downwards (similar to ARM and x86). The difference between Figure 3.2b and 3.2c is in the relative order of stack and statically-allocated data in linear memory. In Emscripten and Clang, static data comes first by default. In Rust and in Clang with the linker option `-stack-first`, the stack comes first and static data sits between stack and heap.

3.2.3 *Memory Protections*

One of the most basic protection mechanisms in native programs is virtual memory with *unmapped pages*. A read or write to an unmapped page triggers a page fault and terminates the program, hence an attacker must avoid writing to such addresses. WebAssembly's *linear*

5 https://emscripten.org/docs/introducing_emscripten/release_notes.html
6 https://v8.dev/blog/emscripten-llvm-wasm

Table 3.1: Exploitation under different layouts of linear memory.

Compiler	Data that can be corrupted with a…	
	Stack-based Buffer Overflow	Stack Overflow
`emcc` 1.39.7 (*fastcomp*)	heap	heap
`emcc` 1.39.7 (*upstream*)	caller, heap	data
`clang` 9 (WASI)	caller, heap	data
`clang` 9 (WASI with `-stack-first`)	caller, static data, heap	∅
`rustc` 1.41 (WASI)	caller, static data, heap	∅

memory, on the other hand, is a single, contiguous memory space without any holes, so every pointer $\in [0, max\_mem]$ is valid. As long as the attacker stays within this bound, any read or write will succeed. This is a fundamental limitation of linear memory with severe consequences. Since one cannot install guard pages between static data, the unmanaged stack, and the heap, overflows in one section can silently corrupt data in adjacent sections. Table 3.1 shows that stack-based buffer overflows and stack overflows are thus very powerful attack primitives in WebAssembly. They can, depending on the layout, overwrite data in the heap, in a caller's stack frame, and static data. We will exploit this in the attack primitives of Section 3.3.

Virtual memory in native execution also allows setting *page protection flags*, i.e., marking pages exclusively as readable, writable, or executable. In WebAssembly, linear memory is non-executable by design, as it cannot be jumped to. However, WebAssembly does not allow marking memory as read-only. Instead, all data in linear memory is always writable! This is another quite surprising limitation of linear memory and enables one of our attack primitives in Section 3.3.

As an additional probabilistic defense in native execution, *address space layout randomization* (ASLR) [PaX Team 2002] randomly arranges the stack, heap, and code in the address space at runtime. For a successful attack, the attacker thus first has to obtain a pointer, e.g., to the heap, via an information disclosure vulnerability. In WebAssembly, there is no ASLR. WebAssembly linear memory is arranged deterministically, i.e., stack and heap positions are predictable from the compiler and program. Even if one were to add some form of ASLR to WebAssembly,

linear memory is addressed by 32-bit pointers, which likely does not provide enough entropy for strong protection [Shacham et al. 2004].

3.3 ATTACK PRIMITIVES

This section presents attack primitives that can be used to exploit vulnerabilities in code compiled to WebAssembly. The attack primitives span three dimensions from which a full attack can be constructed. The first dimension is about obtaining a write primitive, i.e., the ability of an attacker to use a vulnerability for unexpected writes to memory. The second dimension corresponds to the data that can be overwritten. The third dimension is about triggering security-compromising behavior. In principle, the primitives in these three dimensions can be freely combined. For example, a write primitive from the first dimension can overwrite any data from the second dimension to trigger any kind of misbehavior from the third dimension.

Figure 3.1 gives an overview of the three dimensions of attack primitives (💣) and mitigations designed to counter them (🛡). As discussed in detail in the following, many of the standard mitigations used when compiling to native binaries are unused or unavailable when compiling to WebAssembly (shown as crossed out in the figure).

Some of the attack primitives described here are based on existing ideas for exploiting vulnerabilities in C and C++ code compiled to native binaries. The novelty lies in the way these attacks and existing mititations transfer, or do not transfer, to WebAssembly. Other attack primitives (e.g., Section 3.3.1.2 and 3.3.2.3) have never been possible in modern native systems with virtual memory and are presented here for WebAssembly for the first time.

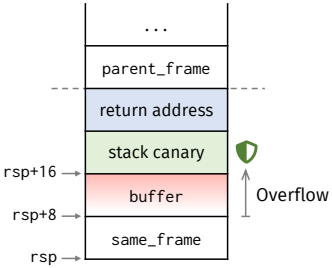
3.3.1 *Obtaining a Write Primitive*

Given a WebAssembly binary compiled from vulnerable C or C++ code, there are several ways for an attacker to obtain a write primitive. In particular, we discuss those types of attacks for which there are effective mitigations on native platforms, but not in WebAssembly.[7]

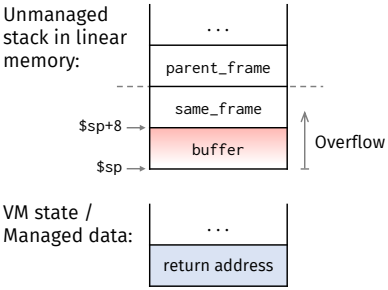
7 We do not discuss attack primitives that are possible in WebAssembly but neither novel nor specific to this platform. E.g., integer overflows exist in WebAssembly just as they do in x86 or ARM.

```
1  void parent() {
2    char parent_frame[8] = "BBBBBBBB"; // Also overwritten
3    vulnerable(readline());
4    // Dangerous if parent_frame is passed, e.g., to exec
5  }
6  void vulnerable(char* input) {
7    char same_frame[8] = "AAAAAAAA"; // Can be overwritten
8    char buffer[8];
9    strcpy(buffer, input); // Buffer overflow on the stack
10 }
```

(a) Vulnerable C program, overflowing `buffer` on the stack.



(b) Stack layout on x86-64 with canaries and reordering.



(c) Unmanaged stack and VM state in WebAssembly.

Figure 3.3: Example of a stack-based buffer overflow and its exploitability in WebAssembly.

3.3.1.1 *Stack-based Buffer Overflow*

Stack-based buffer overflows have been widely exploited in native code [Aleph One 1996] and, by now, there exist several mitigations against them. We show that, contrary to current beliefs, stack-based buffer overflows are exploitable in WebAssembly.

Figure 3.3 shows C code prone to overflow because line 9 fails to perform bounds checking. Figure 3.3b shows the stack layout when compiling this code with a modern compiler to x86. The stack contains local variables of the current function (`same_frame` and `buffer`), local variables of parent functions (`parent_frame`), saved registers (if any), and the return address. An overflow of `buffer` could overwrite data on the stack, in particular return addresses. However, modern compilers

mitigate this kind of attack in several ways. To detect buffer overflows, compilers place *stack canaries* (or stack cookies) [Cowan et al. 1998] above local data. Before a function returns, the canary value on the stack is compared with the original value. If the values do not match, this indicates a buffer overflow and execution is aborted. To minimize the data that could be overwritten, compilers also reorder local variables on the stack. In many cases, the compiler can also prevent potential buffer overflow vulnerabilities through semantics-preserving code transformations. For example, the `FORTIFY_SOURCE` flag allows the compiler to replace `strcpy` with `strncpy` if the length of the string is known.

Do stack-based buffer overflows affect WebAssembly? Because WebAssembly VMs isolate managed data, in particular, return addresses, it is tempting to get a strong (and false) sense of security, as illustrated by the following quote from an official language design document:[8]

> "Compared to traditional C/C++ programs, [WebAssembly's] semantics obviate certain classes of memory safety bugs. Buffer overflows, which occur when data exceeds the boundaries of an object and accesses adjacent memory regions, cannot affect local or global variables [...]. *Thus, common mitigations such as data execution prevention (DEP) and stack smashing protection (SSP) are not needed by WebAssembly programs.*"

While the premise is true (return addresses and locals are safely managed), the conclusion (in italics) is not. Buffer overflows *can* compromise data in WebAssembly because parts of the function-scoped data of the source program is stored on the unmanaged stack in the linear memory (as we discussed in Section 3.2.1).

Figure 3.3c illustrates the problem by showing the unmanaged stack in linear memory (top), as well as the internal state of the WebAssembly VM that stores the return addresses of calls (bottom). An overflow while writing into a local variable on the unmanaged stack, e.g., `buffer`, may overwrite other local variables in the same and even in other stack frames upwards in the stack, e.g., `parent_frame`. Because overflows can also write to data in the parent function (as we show above) and even to other memory sections (as we show later), the primitive is more powerful and the use of stack canaries more important than previously realized [Bergbom 2018; McFadden et al. 2018].

8 https://github.com/WebAssembly/design/blob/master/Security.md#memory-safety

3.3.1.2 *Stack Overflow*

Another write primitive are stack overflows, which occur due to excessive or infinite recursion or when a local buffer of variable size is allocated on the stack, e.g., using `alloca`. If an attacker controls the size of stack allocations, or provides corrupted input data that violates internal assumptions of recursive functions, she may trigger a stack overflow. For example, recursive implementations of functions operating on trees or lists often assume acyclicity; a cyclic data structure passed to such a function can then lead to infinite recursion.

On most native platforms, stack overflows will cause the program to crash as the stack grows into a special *guard page* that separates the stack from other areas of memory. In WebAssembly, such protections do not exist for the unmanaged stack, so an attacker-controlled stack overflow can be used to overwrite potentially sensitive data following the stack (as discussed for different memory layouts in Section 3.2.2).

3.3.1.3 *Heap Metadata Corruption*

Another primitive an attacker may use to write memory in WebAssembly programs is to corrupt heap metadata of the memory allocator statically linked into a WebAssembly binary. Because WebAssembly defines only a low-level `memory.grow` operation, and no allocator is provided by the host environment, compilers include a memory allocator as part of the compiled binary (cf. Section 2.2). Since the binary is often downloaded from the Internet right before execution, the code size of the allocator is an important consideration. The Emscripten compiler therefore lets developers choose between the default allocator, based on `dlmalloc`, and the simplified allocator `emmalloc` that reduces the final code size. Similarly, Rust programs can choose a more lightweight allocator when compiling to WebAssembly, called `wee_alloc`.[9]

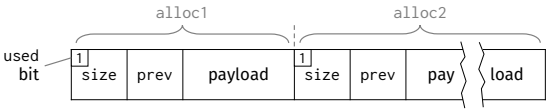
While standard allocators, such as `dlmalloc`, have been hardened against a variety of metadata corruption attacks, simplified and lightweight allocators are often vulnerable to them. We find both `emmalloc` and `wee_alloc` to be vulnerable to metadata corruption attacks, which we illustrate for a version of `emmalloc` in the following.[10]

9 https://github.com/rustwasm/wee_alloc

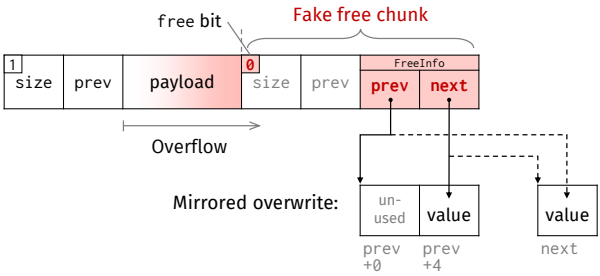
10 Recently, `emmalloc`'s implementation was slightly changed, but it is still vulnerable against this type of attack. We provide an exploit against the newer version as well in our supplementary material.

```
1  struct FreeInfo { FreeInfo* prev; FreeInfo* next; };
2  struct Chunk {
3    size_t used : 1; size_t size : 31;
4    Chunk* prev;
5    union {  // Depending on whether the chunk is free or not.
6      char payload[];
7      FreeInfo freeInfo;
8    };
9  };
10 // Called on alloc2, before merging it into alloc1.
11 void removeFromFreeList(Chunk* chunk) {
12   FreeInfo* freeInfo = chunk->freeInfo;
13   freeInfo->prev->next = freeInfo->next; // mirrored
14   freeInfo->next->prev = freeInfo->prev; // write
15 }
```

(a) Excerpt from the `emmalloc` allocator (edited for clarity).



(b) Heap layout before the overflow: two adjacent chunks.



(c) Heap layout after an overflow of `alloc1`: manipulated metadata causes a mirrored write to a chosen location when executing `free`.

Figure 3.4: Example of a heap metadata corruption in `emmalloc` after an overflow on the heap.

When deallocating a chunk of memory by calling `free`, allocators try to merge as many adjacent free chunks as possible into a single larger one to avoid fragmentation. This gives rise to the classical *unlink exploit* [Anonymous 2001; Kaempf 2001] shown in Figure 3.4. Since `emmalloc` is a *first-fit* allocator, it will return the first chunk in the free list large enough to satisfy an allocation request. Thus, two directly following allocation requests yield two chunks adjacent to each other in memory, such as `alloc1` and `alloc2` in Figure 3.4b. Lines 1 to 9 of `emmalloc`'s source code in Figure 3.4a show that the metadata of each chunk starts with a bit indicating whether the current chunk is free or not, the chunk's size, a pointer to the preceding chunk, and finally either the payload (raw bytes) or a `FreeInfo` struct, which in a benign allocation makes that chunk part of a doubly linked list of free chunks.

Given an overflow of data in `alloc1` (e.g., due to a `memcpy` with the wrong length), an attacker can write to the directly adjacent metadata of `alloc2` to clear the `used` bit and set up a "fake" `FreeInfo` struct (Figure 3.4c). Finally, when `alloc1` is freed, the allocator checks whether there is an opportunity to merge the newly freed chunk with an adjacent free chunk. Because the manipulated metadata identifies the following chunk as free, the allocator calls `removeFromFreeList` to unlink it in preparation for merging the two. In line 13 of Figure 3.4a, the unlinking code of `emmalloc` then writes the attacker-controlled value of the `next` field into the `next` field of another `FreeInfo` struct (i.e., to an offset of 4 bytes) at the attacker-controlled address in `prev`. This allows the attacker to write an arbitrary value to an arbitrary address. Due to line 14, there additionally is a mirrored write into the location pointed to by `next`. Thus, to avoid a runtime error terminating execution, both `prev` and `next` must be valid pointers. Since Emscripten allocates a stack of at least 5 MiB by default, values below 5×2^{20} can in all likelihood be safely written. This is more than sufficient for overwriting function table indices (see Section 3.3.3.1), which are at most in the range of thousands.

The methods discussed so far for obtaining write primitives are by no means exhaustive, but the most direct methods from the traditional exploit arsenal that currently do not have mitigations in WebAssembly. Other possible attacks may exploit format string vulnerabilities, use-after free and double-free vulnerabilities, single-byte buffer overflows, or perform more sophisticated attacks on memory management.

3.3.2 *Overwriting Data*

The second dimension of our attack primitives corresponds to the data that can be overwritten to gain additional control over the execution.

3.3.2.1 *Overwriting Stack Data*

The unmanaged stack in linear memory contains function-scoped data, such as arrays, structs or any value that has its address taken. With a given fully-flexible write primitive, an attacker can overwrite any potentially critical local data including function pointers represented as function table indices or arguments to security-critical functions.

In contrast to native code, there are no return addresses on the unmanaged stack. Hence, a purely linear stack-based buffer overflow cannot easily take control of the execution. However, the overflow can reach all currently active call frames if the stack is growing downwards, as it does in most configurations (see Section 3.2.2). Because there are no return addresses or stack canaries, the overflow can overwrite local data of all calling functions without risking early termination.

3.3.2.2 *Overwriting Heap Data*

The heap commonly contains data with longer lifetime and will store complex data structures across different functions. Targeted writes to heap data are straightforward in WebAssembly due to the fully deterministic memory allocation (Section 3.2.3). To make matters worse, even a linear stack-based buffer overflow of sufficient length can corrupt heap data. The reasons are that the heap comes after the stack in any compiler configuration (Section 3.2.2) and that no mechanism, such as guard pages, mitigates such attempts.

Note that with a single linear memory, there is no way to avoid the fundamental risk of either stack overflows or stack-based buffer overflows. If the stack grows upwards, a stack overflow can silently corrupt heap data. If the stack grows downwards, stack-based buffer overflows are the culprit.

3.3.2.3 *Overwriting "Constant" Data*

The following is the perhaps most surprising target of a data overwrite, as it is impossible in modern native platforms. Many programming lan-

guages allow protecting data from being overwritten by declaring it constant. This is enforced not just by the type system, but also at runtime by placing the data in read-only memory. As WebAssembly has no way of making data immutable in linear memory, an arbitrary write primitive can change the value of any non-scalar constant in the program, including, e.g., all string literals. Even more restricted write primitives allow modification of constant data: a stack overflow with the memory layout of Figure 3.2b can write into constant data; similarly, a stack-based buffer overflow can reach constant data in the memory layout of Figure 3.2c. As a result, an attacker with either of those capabilities can overwrite any supposedly constant data, compromising the guarantees intended by the programming language. We will show two examples of exploits caused by this surprising aspect of WebAssembly linear memory in the next section.

3.3.3 *Triggering Unexpected Behavior*

Given a write primitive (Section 3.3.1) and a choice of data to overwrite (Section 3.3.2), there are several ways for an attacker to trigger unexpected behavior. This is the third dimension of our attack primitives.

3.3.3.1 *Redirecting Indirect Calls*

The closest equivalent to native control-flow attacks in WebAssembly is the redirection of indirect function calls. This allows for executing code that normally would not be executed in a given context.

In Section 2.2 and Figure 2.2, we have illustrated indirect function calls in WebAssembly. An attacker may redirect an indirect call by overwriting an integer in linear memory that eventually serves as an index into the *table* section. As described in Section 3.3.2, this integer value may be a local variable on the unmanaged stack, part of a heap object, in a vtable, or even a supposedly constant value.

WebAssembly has two mechanisms that limit an attacker's ability to redirect indirect calls. First, not all functions defined in or exported into a WebAssembly binary appear in the table for indirect calls, but only those that may be subject to an indirect call. Second, all calls, both direct and indirect, are type checked. As a result, an attacker can redirect calls only within the equivalence class of functions of the same type, similar to type-based control-flow integrity [Abadi et al. 2005].

In Section 3.5 we measure to what extent these mechanisms reduce the available call targets an attacker can choose from.

3.3.3.2 *Code Injection into the Host Environment*

WebAssembly modules can interact with their host environment in various ways to cause externally visible effects. One such way is to invoke the notorious `eval` function of a JavaScript host environment, which interprets a given string as JavaScript code. To access `eval`, WebAssembly modules compiled via Emscripten can use `emscripten_run_script` and related APIs, which execute JavaScript code in the host environment. This is available both in browsers and in server-side code executing with Node.js.[11] In browsers, also any function that adds code to the document (e.g., `document.write`) can serve as an `eval`-equivalent for constructing exploits. In Node.js, there is no browser sandbox, so APIs for interfacing with the operating system give even more options for code injection, e.g., the `exec` function of the `child_process` module.

Using the primitives described in Section 3.3.1 and Section 3.3.2, an attacker may inject malicious code by overwriting the argument passed to an `eval`-like function. For example, suppose a WebAssembly usually invokes `eval` with a "constant" string of code stored in linear memory, then an attacker could overwrite that constant with malicious code.

3.3.3.3 *Application-Specific Data Overwrite*

Depending on the application, there can be other sensitive targets for data overwrites. For example, a WebAssembly module issuing web requests through an imported function could be made to contact a different host by overwriting the destination string, to initiate cookie stealing. As a further example, several interpreters and runtimes have been compiled to WebAssembly, e.g., to execute CIL/.NET code directly in the browser (Section 2.3). These kinds of environments contain many opportunities for significantly altering program behavior, e.g., by overwriting bytecode then interpreted by the runtime.

11 https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html#interacting-with-code-call-javascript-from-native

Table 3.2: Overview of our end-to-end attacks, using different combinations of attack primitives on three host environments.

Section	3.4.1	3.4.2	3.4.3
Host environment	Browsers (client-side)	Node.js (server-side)	Wasmtime (stand-alone runtime)
Write primitive	Stack-based buffer overflow (CVE-2018-14550)	Heap metadata corruption	Stack-based buffer overflow
Overwritten data	Image tag in DOM string	Function index	String literals
Location of data	Heap	Stack	"Constant" data
Malicious behavior	Cross-site scripting in JavaScript via `document.write()`	Inject arbitrary shell command into `exec()`	Write arbitrary content to chosen file using `fprintf()`

3.4 END-TO-END ATTACKS

We now demonstrate several end-to-end attacks, using different attack primitives from Section 3.3 and targeting different host environments. These attacks substantiate our claim that the current lack of mitigations in the WebAssembly ecosystem are problematic and enable realistic attacks. We make the attacks publicly available, providing a benchmark to guide and evaluate future work on hardening WebAssembly.

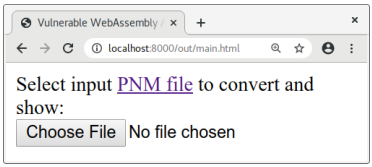
Table 3.2 gives an overview of the end-to-end attacks. They cover different host environments that support WebAssembly: the browser, where we demonstrate a cross-site scripting attack; Node.js, where we show a remote code execution attack; and stand-alone WebAssembly VMs, such as Wasmtime [Wasmtime Website], where we show an arbitrary file write attack.

3.4.1 *Cross-Site Scripting in Browsers*

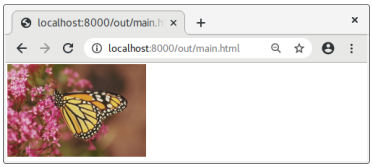
This attack shows that including vulnerable code compiled to WebAssembly into a client-side web application can enable attacks known from JavaScript-based applications, such as cross-site scripting (XSS).

```
1  void main() {
2    std::string img_tag = "<img src='";
3    pnm2png("input.pnm", "output.png");  // CVE-2018-14550
4    img_tag += file_to_base64("output.png") + "'>";
5    emcc::global("document").call("write", img_tag);
6  }
```

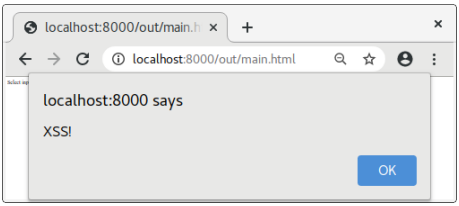
(a) Excerpt of C++ code (to be compiled with Emscripten) that uses the vulnerable C library.



(b) In the benign case: Select a PNM image and…



(c) …convert it to PNG with a C library, fully on the client side.



(d) A malicious input can overflow a buffer on the stack, then corrupt a string on the heap, which is later used in DOM manipulation.

Figure 3.5: Example of cross-site scripting caused by using the vulnerable `libpng` library (CVE-2018-14550).

As an example, consider an image sharing service where users upload and view images. The service provides a web application that converts images between different formats on the client side, using a version of the *libpng* image codec library compiled to WebAssembly (Figure 3.5). Given a file to be converted to PNG, the application calls libpng and then displays the image by calling a DOM manipulation function, such as `document.write`, provided by the JavaScript host environment.

Version 1.6.35 of libpng suffers from a known buffer overflow vulnerability [CVE-2018-14550], which can be exploited when converting a PNM file to a PNG file. When the library is compiled to native code

with modern compilers on standard settings, stack canaries prevent this vulnerability from being exploited. In WebAssembly, the vulnerability can be exploited unhindered by any mitigations.

To exploit the vulnerability for cross-site scripting, an attacker provides a malicious image to another user who then displays it using the web application. Figure 3.5a shows a minimal version of such an application. During normal execution, the application converts the image (line 3), encodes it with Base64 in a data URL, copies it into an `img` tag (line 4), and then adds the tag into the document (line 5). Since the image is embedded into the DOM as a base64-encoded string, it normally cannot lead to XSS. However, exploiting the stack-based buffer overflow in libpng allows the attacker to overwrite higher addresses, including the heap, which holds the C++ string with the `img` tag (line 2). The attacker can then replace the `img` tag with arbitrary other content, e.g., a `script` tag that displays an alert, which will then get passed to `document.write`.

Depending on how the input data is provided, the above scenario can lead to both non-persistent and persistent XSS attacks. In the non-persistent variant, the attacker tricks a user into processing a malicious image with the web application, which then immediately triggers the attack in the user's browser. In the persistent variant, the attacker uploads the malicious input image to some storage and then shares it with others, who will be attacked once they download and convert the input in their browser with the vulnerable WebAssembly code.

3.4.2 *Remote Code Execution in Node.js*

In the next attack, we demonstrate that including vulnerable WebAssembly in a Node.js application can enable remote code execution.

As an example, consider a server that accepts requests to log the IDs of customers that have been happy or unhappy about some product. Listing 3.1a shows an excerpt of the code running in the server application. The `handle_request` function receives three attacker-controlled arguments: `input1`, which describes whether the customer was happy; `input2`, which is supposed to be the length of the string in `input1`; and `input3`, which contains the ID of the customer. Depending on the customer's happiness, the code calls `log_happy` or `log_unhappy`, which is selected by assigning the respective function to the function pointer `func`.

```
1  // Functions supposed to be triggered by requests
2  void log_happy(int customer_id) { /* ... */ }
3  void log_unhappy(int customer_id) { /* ... */ }
4
5  void handle_request(char *input1, int input2, char *input3) {
6    void (*func)(int) = NULL;
7    char *happiness = malloc(16);
8    char *other_allocation = malloc(16);
9    memcpy(happiness, input1, input2); // Heap overflow
10   if      (happiness[0] == 'h') func = &log_happy;
11   else if (happiness[0] == 'u') func = &log_unhappy;
12   free(happiness); // Unlink exploit overwrites func
13   func(atoi(input3)); // 3rd input is passed as argument
14 }
15
16 // Somewhere else in the binary:
17 void exec(const char *cmd) { /* ... */ }
```

(a) Sample application that calls one of two logging functions depending on its input. It suffers from a heap overflow, which causes an arbitrary write on `free`, allowing to redirect `func` to `&exec`. Then `input3` can be chosen as the address of an injected string.

```
1  (func $log_happy   (param i32) (result) ...)
2  (func $log_unhappy (param i32) (result) ...)
3  (func $exec        (param i32) (result) ...)
```

(b) Excerpt of the function section for the binary compiled from (a), showing that `exec`, `log_happy`, and `log_unhappy` all have the same low-level WebAssembly type signature [`i32`] → [ ].

Listing 3.1: Example of a remote code execution exploit.

The code contains a heap overflow vulnerability at line 9. In the absence of safe unlinking and other mitigations (we use the `emmalloc` allocator for our proof of concept) an attacker can use the overflow to obtain an arbitrary write primitive through the classic heap metadata corruption attack (see Section 3.3.1.3). If the function pointer `func` is compiled into a variable in linear memory (which is the case, e.g., for all function pointers in vtables), the attacker can use the write primitive to manipulate it and redirect the call (Section 3.3.3.1). The absence of ASLR simplifies such an attack further, as the address to overwrite is deterministic.

One possible target for redirecting the call is the `exec` function that can also be found in the binary (line 17). While `exec` and the `log_*` func-

```c
// Append "constant" string into "constant" file.
FILE *f = fopen("file.txt", "a");
fprintf(f, "Append constant text.");
fclose(f);

// Somewhere else in the binary:
char buf[32];
scanf("%[^\n]", buf); // Stack-based buffer overflow!
```

(a) Excerpt of a C program with a stack-based buffer overflow that overflows into the 'constant' memory section, causing an arbitrary file write.

```
(data (i32.const 65536) "%[^\0a]\00      ;; Format string for scanf().
                         file.txt\00     ;; Filename.
                         a\00            ;; Open mode for fopen().
                         Append constant text.\00...") ;; Content.
```

(b) Excerpt of the data section for the binary compiled from (a), which shows that the string literals for the filename, the contents to be written, and even the open mode are all located in regular (writable) linear memory.

Listing 3.2: Example of an arbitrary file write exploit.

tions have different C++ types, all three functions have identical types on the WebAssembly level (Listing 3.1b). The reason is that both integers and pointers are represented as `i32` types in WebAssembly, i.e., the redirected call passes WebAssembly's type check. The final challenge is to pass an arbitrary command into `exec`, which is similar to the injection of shellcode in native exploitation. One option is to inject a suitable command string into the heap when overwriting the function index, and to then pass a decimal string with the address of the command string as `input3`.

3.4.3 *Arbitrary File Write in a Stand-Alone VM*

As discussed in Section 2.3, WebAssembly is starting to establish itself as a universal bytecode beyond web applications. There are multiple virtual machines for running stand-alone WebAssembly applications, including Wasmtime, Wasmer, and WAVM. The interface for such applications is WASI, and Clang can compile compatible binaries.

This attack demonstrates that, despite stand-alone WebAssembly VMs being advertised as a secure platform for executing C/C++ code, Web-

Assembly currently enables attacks impossible in modern native execution platforms. Listing 3.2a shows an excerpt of an apparently harmless application that appends a constant string to a statically known file. Somewhere else in the program, the code suffers from a textbook buffer overflow, which enables an attacker to overwrite data on the stack. Compiled to a native target, exploiting the buffer overflow cannot influence the file I/O, which is entirely based on string literals stored in the read-only memory pages loaded from the `.rodata` section.

When running on a stand-alone WebAssembly VM, this vulnerability can be exploited for an arbitrary file write. The strings for filename and contents are stored in the unmanaged linear memory, as shown in Listing 3.2b. They can be overwritten by a stack-based buffer overflow of sufficient length if data lies above the stack (see Section 3.2.2). As a result, the attacker can write arbitrary data into an arbitrary file by overwriting the filename and contents strings. In our exploit, even the file open mode `"a"` (append) is changed to `"w"` by simply overwriting the corresponding string in the data section.

3.5 QUANTITATIVE EVALUATION

To better understand how realistic the attacks described so far are in practice, we now present a quantitative evaluation on real-world WebAssembly binaries. We address the following research questions:

RQ1 *How much data is stored on the unmanaged stack?* This question is relevant because the unmanaged stack serves both as an entry point to obtain a write primitive, e.g., via a stack-based buffer overflow, and as a target for overwrites, e.g., to manipulate sensitive data. (Section 3.5.2)

RQ2 *How common are indirect calls and how many functions can be reached from indirect calls?* These questions are relevant to understand the risk for control-flow divergence by redirecting indirect calls. (Section 3.5.3)

RQ3 *How does WebAssembly's type checking of indirect call targets compare to current control-flow integrity (CFI) defenses for native binaries?* Since the runtime validation of indirect call targets performed by the WebAssembly VM resembles CFI defenses, we compare both in terms of CFI equivalence classes and class sizes. (Section 3.5.4)

We make our full dataset and the tools we developed to obtain them available at https://github.com/sola-st/wasm-binary-security.

3.5.1 *Experimental Setup and Analysis Process*

PROGRAM CORPUS The binaries we analyze in our quantitative evaluation are split into two groups. First, we collect a set of nine binaries from real-world, deployed WebAssembly applications: Adobe's Document Cloud View SDK[12] renders and annotates PDFs in the browser; Figma[13] is a collaborate user-interface design web application; the 1Password X 1.17 browser extension[14] contains a WebAssembly component for password generation; Doom 3 as an example of a large game engine ported to WebAssembly[15]; and finally a set of codecs (webp, mozjpeg, optipng, hqx) for in-browser image conversion.[16] The binaries span different application domains (document editing, games, codecs), deployment scenarios (web application, browser extension), and source languages (C, C++, Rust), so we believe they are a good first approximation of realistic WebAssembly binaries. We collected their most recent versions at the time of our evaluation in March 2020. Since our tool is open source, we welcome others to replicate our results and extend them by analyzing other WebAssembly binaries.

The second group of binaries in our corpus are 17 C and C++ programs from the SPEC CPU 2017 benchmark suite, compiled to WebAssembly. The SPEC CPU suite has been used before to study the performance of WebAssembly [Jangda et al. 2019]. It has also been used to evaluate the security of CFI techniques for native code [Burow et al. 2017; X. Xu et al. 2019], enabling us to address RQ3. Those programs are from compute-heavy domains (programming language implementations, simulations, video codecs, compression), matching the original use cases for WebAssembly.[17]

Our combined program corpus consists of 26 WebAssembly binaries, which contain 19.2 million instructions across 98,924 functions in total. Table 3.3 gives a more detailed overview.

12 https://www.adobe.io/apis/documentcloud/dcsdk/viewsdk.html
13 https://www.figma.com/
14 https://1password.com/
15 http://www.continuation-labs.com/projects/d3wasm/
16 https://squoosh.app/
17 https://webassembly.org/docs/use-cases/

TOOLCHAIN AND CONFIGURATION We compiled the SPEC CPU programs with Emscripten 1.39.7, i.e., the most recent version at the time of the evaluation, with its *upstream* backend. Since this backend is shared by all LLVM-based WebAssembly compilers (Clang, Rust), our results should translate also to them. For completeness, we also compiled all SPEC CPU programs with the now deprecated *fastcomp* backend of Emscripten. Since fastcomp was the default backend of Emscripten until October 2019, its results are relevant for large amounts of code previously compiled to WebAssembly. The results for fastcomp and upstream are very similar, so for brevity we only present the upstream results in the following.

To obtain optimized binaries without symbols or debug information, we compile with the `-O3` option. GCC, x264, Blender, and Xalan-C++ required several preprocessor flags for compatability, e.g., to set correct integer bit-widths and platforms. Some programs also had to be manually linked because Emscripten's libc (based on `musl`) causes errors due to duplicate symbol definitions.

STATIC ANALYSIS To address our research questions, we develop a lightweight static analysis tool. To the best of our knowledge, it is the first security analysis tool for WebAssembly binaries. The analysis is written in about 600 lines of Rust and does the following.

- It extracts general information about the program, e.g., instruction counts, the number of functions, and their types.
- It analyzes the unmanaged stack by inferring which *global* is the stack pointer, which functions access it, and how the stack pointer is incremented and decremented.
- It analyzes the *table* section and its static initialization, to find out which functions are present in it. It also determines the function type for each initialized table entry.
- Finally, it analyzes indirect call edges. For each `call_indirect` instruction, it collects the statically encoded type of allowed targets, it determines all type-compatible functions that are present in the table, and presents this information as CFI equivalence classes and computes the class size.

We explain the analyses in more detail in the following.

3.5.2 *RQ1: Measuring Unmanaged Stack Usage*

Measuring how much data a program stores on the unmanaged stack (RQ1) is important for two reasons. First, such data could potentially suffer from a stack-based overflow. Second, such data may become subject to overwrites once an attacker has a write primitive. So, how much data ends up on the unmanaged and unprotected stack?

STATIC ANALYSIS Our static analysis measures the size of the stack frame on the unmanaged stack for each non-imported function. The analysis operates on optimized, stripped binaries without debug information, as a realistic attacker would, and thus has to infer the unmanaged stack usage directly from the bytecode.

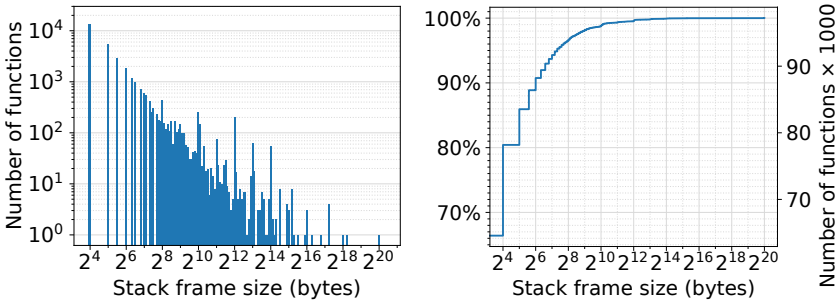
First, the analysis needs to identify the stack pointer. Unlike in native binaries, there is no convention to use a fixed register (such as `rsp` on x86, which does not exist in WebAssembly) or global variable for the stack pointer. Instead, the analysis extracts all instructions that modify globals and selects the one that is most frequently read and written. A manual analysis confirms that this heuristic reliably finds the stack pointer. From the identified global's static initialization, we also know the base address of the unmanaged stack in linear memory.

Second, for each function, the analysis infers the size of the stack frame on the unmanaged stack. In all analyzed binaries, the previously identified stack pointer is modified in a protocol similar to function prologues and epilogues in native binaries. Specifically, our analysis pattern matches against the following sequence of instructions and extracts the *delta* value, which gives us the stack frame size:

```
1  global.get $i
2  i32.const <delta>
3  i32.add or i32.sub
4  local.tee $j (optional)
5  global.set $i
```

This sequence first reads the current stack pointer from global `$i` (identified earlier), then increments or decrements it (depending on whether the stack grows upwards or downwards, see Section 3.2.2), optionally saves it to a `local` (akin to a base pointer), and finally writes the modified value back to the global stack pointer.

RESULTS Figure 3.6 shows the distribution of stack frame sizes across all analyzed binaries, both as a histogram (Figure 3.6a) and the cumulative distribution (Figure 3.6b). One third (32,651) of all functions in the



(a) Histogram (double logarithmic). **(b)** Cumulative distribution.

Figure 3.6: Distribution of frame sizes on the unmanaged stack for all functions in the program corpus.

program corpus store some data on the unmanaged stack. The smallest frame size of 16 (2^4) bytes is allocated by 13,620 functions (14% of all functions). Stack frame sizes span the whole range from 16 bytes to 1 MiB, which is the largest static stack allocation. The distribution has a long tail towards large stack frames. From the cumulative distribution in Figure 3.6b, we see that 6% (6,127) of all functions allocate 128 (2^7) bytes or more on the unmanaged stack, and 1.3% (1,232) of all functions allocate at least 1 KiB.

Overall, we see that many functions use the unmanaged stack, which is susceptible not only to arbitrary memory writes but also to inter-frame buffer overflows (see Section 3.3). This implies that with increasing call depth, the chance for an attacker to find at least some data to overwrite increases quickly. For example, with ten nested calls (assuming a uniform distribution of functions), there would be some data on the unmanaged stack with $1 - ((1 - 0.33)^{10}) \approx 98.2\%$ probability. We conclude that (1) a lot of stack data is prone to being overwritten by buffer overflows and arbitrary write primitives, and (2) it is important to isolate stack frames on the unmanaged stack, e.g., using stack canaries.

3.5.3 *RQ2: Measuring Indirect Calls and Targets*

To better understand the risk for control-flow attacks (RQ2), we analyze indirect calls and their call targets in the binaries.

Table 3.3: Overview of programs and static analysis results on indirect calls, the function table, and CFI equivalence classes.

	Binary	Source	Instruct.	Indirect Calls		Functions					CFI Equivalence Classes			
				Count	of All	Count	Indirectly Callable		Idx. from Mem.		Count	Min	Max	Avg
Collected, real-world prog.	Adobe View SDK	C++	1.1M	2803	6.2%	12566	3076	24.5%	3054	24.3%	87	1	848	32.2
	1Password X exten.	Rust	730.2k	283	1.4%	1941	596	30.7%	586	30.2%	19	1	91	14.9
	Doom 3	C++	1.7M	17903	31.3%	8239	4449	54.0%	4408	53.5%	642	1	3889	27.9
	Figma	C++	3.2M	10469	8.1%	13619	3657	26.9%	3635	26.7%	68	1	4519	154.0
	WebP encoder	C	73.1k	87	3.6%	889	165	18.6%	69	7.8%	22	1	15	4.0
	WebP decoder	C	43.4k	69	5.4%	563	160	28.4%	107	19.0%	20	1	9	3.5
	mozjpeg	C	77.7k	298	22.0%	388	135	34.8%	116	29.9%	28	1	169	10.6
	optipng	C	119.2k	169	5.4%	735	152	20.7%	124	16.9%	28	1	34	6.0
	hqx	Rust	111.4k	34	0.6%	73	17	23.3%	15	20.5%	4	1	16	8.5
Compiled from SPEC CPU 2017	500.perlbench	C	837.8k	425	1.6%	2128	980	46.1%	956	44.9%	31	1	93	13.7
	502.gcc	C	2.9M	3642	2.5%	9541	3394	35.6%	3375	35.4%	78	1	982	46.7
	505.mcf	C	27.4k	44	8.8%	136	12	8.8%	8	5.9%	7	1	28	6.3
	508.namd	C++	323.0k	41	1.1%	296	124	41.9%	107	36.1%	15	1	12	2.7
	510.parest	C++	1.0M	1229	2.6%	3762	2864	76.1%	2714	72.1%	97	1	199	12.7
	511.povray	C++	385.4k	228	1.9%	1421	521	36.7%	510	35.9%	29	1	57	7.9
	519.lbm	C	13.4k	12	6.2%	80	7	8.8%	6	7.5%	5	1	8	2.4
	520.omnetpp	C++	619.3k	4536	10.6%	4615	3569	77.3%	3505	75.9%	79	1	1631	57.4
	523.xalancbmk	C++	1.5M	13567	16.2%	8050	6225	77.3%	6072	75.4%	77	1	3893	176.2
	525.ldecod	C	233.0k	354	8.6%	551	129	23.4%	68	12.3%	24	1	135	14.8
	525.x264	C	283.6k	773	14.2%	636	253	39.8%	177	27.8%	31	1	105	24.9
	526.blender	C++	3.2M	17198	14.9%	25901	17387	67.1%	17263	66.6%	128	1	5360	134.4
	531.deepsjeng	C	53.0k	10	1.1%	174	10	5.7%	8	4.6%	5	1	6	2.0
	538.imagick	C	517.5k	1901	9.9%	1068	91	8.5%	74	6.9%	22	1	1592	86.4
	541.leela	C++	118.8k	263	5.0%	1101	600	54.5%	520	47.2%	41	1	74	6.4
	544.nab	C	55.6k	17	1.7%	201	10	5.0%	8	4.0%	6	1	7	2.8
	557.xz	C	53.3k	71	11.0%	250	98	39.2%	86	34.4%	19	1	11	3.7
	Average per binary		738.1k	2939.5		3804.8	1872.3		1829.7		62.0	1	914.7	33.2
	Total		19.2M	76426	9.8%	98924	48681	49.2%	47571	48.1%				

INDIRECT CALLS First, we want to know how many indirect calls are present in a binary, since each such call could be a source of an unintended control-flow edge. Our analysis disassembles all binaries in Table 3.3 and counts the number of `call_indirect` instructions (column "Indirect calls: Count"). The percentage of indirect calls relative to all calls varies considerably between programs (column "of All"), from 0.6% up to 31.3%. We also observe that the proportion of indirect calls is independent of whether the source language is C or C++. Averaged over all 26 programs, 9.8% of all call instructions are indirect, i.e., almost every tenth call can be potentially diverted to other functions.

INDIRECTLY CALLABLE FUNCTIONS To successfully redirect a control-flow edge, an attacker not only needs to find an indirect call instruction as the source, but also a compatible function as the target. Two conditions must hold for a function to be a valid indirect call target (Section 2.2). First, the function's type must be compatible with the type statically encoded in the indirect call instruction. WebAssembly function types are very low-level, however. Many distinct source types are lowered to the same WebAssembly type. E.g., the WebAssembly function type `[i32]` → `[]` is compatible with all C functions that return `void` and take any of the following C types as argument: a pointer (regardless of pointee type or `const`-ness), an array, a plain `int`, or anything else that is represented as a 32-bit integer, e.g., enums.

Second, the function must be present in the *table* section of the binary, because the index passed to `call_indirect` is resolved to a function via this table. Our static analysis tool finds which functions are initialized in the table at program startup. Entries in the table cannot be manipulated by the WebAssembly program itself. In principle, the host environment, e.g., JavaScript in the browser, could add or remove entries at runtime. We manually verified that the JavaScript code generated by Emscripten does not modify the table, and thus assume our analysis precisely measures the potential targets of indirect calls.

The columns "Indirectly Callable" in Table 3.3 show how many functions are type-compatible with at least one `call_indirect` instruction *and* present in the *table* section. The percentage of indirectly callable functions ranges from 5% to 77.3%, with on average 49.2% of all functions in the program corpus.

FUNCTION POINTERS IN MEMORY The above results give an upper bound of potential targets for control-flow divergence. In practice, if the table index passed to `call_indirect` comes from a local variable, a global variable, or is the result of a sequence of instructions, then the indices an attacker can choose from are likely more restricted. We also measure how many table indices are read directly from memory. We obtain this number through a static analysis of the instructions preceding indirect calls. Columns "Idx. from mem." show the number of type-compatible and in-table functions, for which at least one indirect call exists that takes its table index *directly from linear memory*. For each such function, given an arbitrary write primitive into linear memory, an indirect call could be diverted to reach the function. Perhaps surprisingly, this number is very close to the upper bound: On average, 48.1% of all functions can be reached by a `call_indirect` that takes its argument directly from linear memory, which is quite unprotected.

Overall, our analysis of indirect calls and targets shows a large potential for effective control-flow divergence. Many functions are indirectly callable (49.2%, on average) and most of them could be reached by simply overwriting an index stored in linear memory (48.1%). We conclude that diverging indirect function calls poses a serious threat to the integrity of control flow in WebAssembly.

3.5.4 *RQ3: Comparing with Existing CFI Policies*

WebAssembly's restrictions on control-flow (Section 2.2) can be seen as a way to enforce control-flow integrity (CFI). In general, CFI has the goal of mitigating runtime attacks by aborting execution when control-flow diverges from the permitted control-flow defined by some policy [Abadi et al. 2005]. For example, a coarse-grained policy can restrict function calls to target only the beginning of functions, but not arbitrary code addresses [M. Zhang and Sekar 2013]. More elaborate policies ensure that virtual calls in a C++ program only call methods in the class hierarchy of the static type of the receiver object, but not other functions. In practice, there are several CFI approaches for native code with different performance-security trade-offs [Burow et al. 2017; Niu and Tan 2014, 2015; X. Xu et al. 2019] and implementations in open-source (GCC and LLVM[18]) and commercial compilers (MSVC[19]).

18 [Tice et al. 2014] and https://clang.llvm.org/docs/ControlFlowIntegrity.html
19 https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

Local control-flow with `br_table` is already more restricted in WebAssembly than in native code, so we assume it is secure here. However, indirect calls can still target potentially many type-compatible function in the table. Similar to coarse-grained CFI for native code, the function types divide the possible targets of an indirect call into equivalence classes, namely all functions with the same type. This is enforcing control-flow integrity (CFI) for forward edges.[20] We thus compare WebAssembly's type checking of indirect calls with CFI defenses for native binaries (RQ3).

EQUIVALENCE CLASSES Following prior work on CFI [Burow et al. 2017], we measure its effectiveness by analyzing the sets of control-flow targets an indirect transfer may be diverted to according to the CFI mechanism. Each such a set is called a *CFI equivalence class*. To assess the effectiveness of a CFI defense, we use two measures: The class count, i.e., how many different classes exist, and the sizes of the classes, i.e., how many targets are in each class. A small class count means the CFI defense distinguishes little between targets, giving attackers more options for control-flow divergence. A large class size is also insecure, as it means a large number of control-flow targets can all be reached from a single source instruction.

For WebAssembly, we measure CFI equivalence classes by analyzing the type signatures of indirectly callable functions, assigning all functions with the same type signature into an equivalence class. Additionally, we analyze the preceding instructions before an indirect call to determine whether they restrict the table index, e.g., via bitmasking, to a smaller range. The last block in Table 3.3 shows the results. On average, there are 62 equivalence classes per program, which each contain 33.2 functions. The largest equivalence class, in the Blender program, contains over 5,300 functions. Overall, this shows that an attacker has plenty of call targets to choose from.

COMPARING WITH NATIVE CFI DEFENSES To put the results on equivalence classes in perspective, we compare them with results reported for native CFI defenses [Burow et al. 2017]. The tables in Table 3.4a and Table 3.4b compare the counts and sizes of equivalence classes, respectively. For example, MCFI [Niu and Tan 2014] and πCFI

20 Backward control-flow edges, i.e., returns, are protected in WebAssembly by design because return addresses are managed by the VM. This offers security that is conceptually similar to shadow stacks for native code.

Table 3.4: Comparing WebAssembly type-checking of indirect calls with native CFI solutions. The data of columns two to four is taken from [Burow et al. 2017]; MCFI corresponds to [Niu and Tan 2014], πCFI to [Niu and Tan 2015]. The programs are the ones in the intersection of SPEC CPU 2006 (theirs) and SPEC CPU 2017 (ours).

(a) Number of equivalence classes (higher means more secure).

Program	Number of CFI Equivalence Classes			
	MCFI	πCFI	LLVM-CFI 3.9	WebAssembly
`perlbench`	38	30	36	31
`mcf`	12	8	N/A	7
`omnetpp`	357	321	35	79
`xalancbmk`	1534	1200	260	77
`namd`	166	150	4	15
`povray`	218	204	33	29

(b) Sizes of equivalence classes (lower means more secure).

Program	Size of Largest CFI Equivalence Class			
	MCFI	πCFI	LLVM-CFI 3.9	WebAssembly
`perlbench`	348	347	350	93
`mcf`	29	15	N/A	28
`omnetpp`	275	253	170	1631
`xalancbmk`	1141	608	95	3893
`namd`	187	113	30	12
`povray`	187	113	81	57

[Niu and Tan 2015] partition the control-flow targets of `xalancbmk` into 1534 and 1200 classes, respectively, whereas WebAssembly's indirect call target restrictions yield only 77 such classes. Regarding the size of equivalence classes, WebAssembly has especially large classes for `omnetpp` and `xalancbmk`, and similar classes sizes as the native defenses for other programs.

Notably, `omnetpp` and `xalancbmk` are C++ programs that make heavy use of object-oriented programming and virtual functions. Source-level type information, e.g., about class hierarchies, can help compiler-based CFI methods to identify more precise, and thus restrictive, equivalence classes. In contrast, WebAssembly's type checking has only (combinations of) four low-level primitive types to work with, which might explain the stark difference to the native schemes.

Overall, WebAssembly's type checking is often less effective than modern CFI defenses available for native binaries. While type-checked indirect calls certainly are a step forward compared to not having any CFI defense, adapting more sophisticated CFI defenses could significantly harden the currently produced binaries. For example, Clang's CFI scheme, which uses source-level information, can also be employed by passing `-fsanitize=cfi` when compiling to WebAssembly.

3.6 DISCUSSION OF MITIGATIONS

Before we conclude, we also want to discuss several mitigations that could defeat the attacks presented so far, e.g., through amending the language specification, updates to compilers, or by application and library developers. Table 3.5 lists mitigations known for native binaries but missing in WebAssembly, where they have to be implemented, and what components of the ecosystem we currently find to be vulnerable.

3.6.1 *WebAssembly Language*

As we discussed in Section 2.2, there are several proposals for extending the WebAssembly language. While most of them are orthogonal to security, some could address our attack primitives.

The multiple memories proposal [Rossberg 2019a] gives one module the option of having multiple linear memories. Under the proposal, memory operations statically encode which memory they operate on, e.g., an `i32.load $mem2` instruction can only load data from memory 2.

Table 3.5: Mitigations that could be employed by different parts of the WebAssembly ecosystem.

Mitigation	(Not) Implemented by	Affected
Page protections and ASLR	Language specification and compilers	All current binaries
Stricter CFI	Language specification and compilers	`emcc` 1.39.7, `clang` 9
`FORTIFY_SOURCE` (for C code)	Compilers	`emcc` 1.39.7, `clang` 9
Stack canaries	Compilers	`emcc` 1.39.7, `clang` 9, `rustc` 1.41
Safe unlinking	Allocators	`emmalloc`, `wee_alloc`

Multiple memories would enable separating stack, heap, and constant data. Thus, an overflow in one memory section would no longer affect data in another memory. Also, pointers to the heap could no longer be forged to point into the stack and vice versa. Finally, if compilers emit only load instructions for a particular memory section, it becomes effectively read-only, since stores to other memories can never modify it. This would prevent overwriting of constants. A challenge with this proposal is that compiling to multiple memories is not straightforward. Since memory accesses are statically restricted to a certain memory, code that must handle pointers of different regions must either be duplicated or objects explicitly copied between memories.

The reference types proposal [Rossberg 2019b] allows modules to have multiple tables for indirect calls. Our call redirection primitive is powerful only because all indirectly callable functions are currently in the same table. Multiple tables allow for finer-grained defenses. One option is to define multiple protection domains, e.g., one per statically-linked library, and to keep a separate table per protection domain. Another option is to split call targets into equivalence classes, similar to existing CFI techniques for native binaries, and to keep a separate table per equivalence class.

Finally, the MS-Wasm proposal [Disselkoen et al. 2019] explicitly targets memory safety. It proposes to add so called segments to WebAssembly, memory regions with defined size and lifetime. Handles into

those segments are promoted to first class types, with own operations for allocation and slicing. This requires quite some implementation effort by hosts, and unless hardware support for memory safety is provided, will likely incur a performance overhead.

A challenge with all changes to the core language is that they require updating existing virtual machines. Since WebAssembly is implemented not just by one vendor, but in at least four browsers (Chrome, Firefox, Safari, and Edge), Node.js, and several stand-alone VMs (Wasmtime, WAVM), this risks a split of the still young ecosystem. However, the reference types proposal was standardized in the meantime, and the multiple memories proposal is (as of April 2022) in phase three of the four phase standardization process.[21]

3.6.2 *Compilers and Tooling*

The perhaps most simple way of preventing many of our attack primitives is to implement and activate security features that compilers, linkers, and allocators already provide for native compilation targets. Decades of research on binary security [Szekeres et al. 2013] have resulted in several mitigations that could be applied to WebAssembly. Examples that would benefit WebAssembly compilers are `FORTIFY_SOURCE`-like code rewriting, stack canaries, CFI defenses, and safe unlinking in memory allocators. In particular for stack canaries and rewriting commonly exploited C string functions, we believe there are no principled hindrances to deployment. We hope they will be implemented by compilers in the future, since they offer good security benefit for relatively little change to the ecosystem, unlike, e.g., language changes.

A longer-term mitigation in compilers is to use the WebAssembly language extensions discussed above, once they become available. For example, when compiling C/C++ to WebAssembly, multiple memories could mimic some of the security features provided by page protections in native code.

3.6.3 *Application and Library Developers*

Developers of WebAssembly applications can reduce the risk by using as little code in "unsafe" languages, such as C, as possible. To reduce the attack surface, developers should also ensure to import only those

21 https://github.com/WebAssembly/proposals

APIs from the host environment that are strictly necessary. For example, calling critical host functions, such as `eval` or `exec` is impossible unless these functions are imported in the WebAssembly module. One way to minimize the set of imported APIs in WebAssembly modules for the browser, is to perform as little DOM manipulation as possible from the WebAssembly module, and instead perform those tasks in the JavaScript part of the application.

3.7 SUMMARY

This chapter presents the first in-depth security analysis of WebAssembly binaries and compares the level of security against runtime attacks provided by the WebAssembly language and ecosystem with that of native binaries. We find that vulnerable source programs result in binaries that allow various kinds of attacks, including attacks that have not been possible on native platforms since decades. Our findings are based on a set of attack primitives that enable an attacker to gain a write primitive, overwrite sensitive data, and trigger compromising behavior. Several end-to-end examples of attacks, which cover WebAssembly running in the browser, on Node.js, and in stand-alone VMs, demonstrate that these primitives can be combined into effective exploits. Moreover, an empirical evaluation of real-world binaries quantifies the exploitation risk, showing a large attack surface. Overall, our findings are a call to arms for further hardening the WebAssembly language, its compilers, and ecosystem, making the promise of a secure platform a reality.

4 WASMBENCH: A STUDY OF REAL-WORLD BINARIES

Despite its popularity in different domains, little is known about WebAssembly binaries that occur in the wild. Especially after the conclusions of the previous chapter (Chapter 3), this leaves several open questions. What are WebAssembly binaries used for and how prevalent are benign use cases? Given that WebAssembly binaries can be exploited, how many binaries are potentially vulnerable?

This chapter reports on a comprehensive empirical study of 8,461 unique binaries gathered from a wide range of sources, including code repositories, package managers, and live websites. We study the security properties, source languages, and use cases of the binaries through a combination of static analysis, manual inspection, and statistical analysis. We find that memory vulnerabilities potentially affect a wide range of binaries (e.g., two thirds of the binaries are compiled from memory-unsafe languages, such as C and C++) and that 21% of all binaries import potentially dangerous APIs from their host environment. We also show that cryptomining, which once accounted for the majority of all WebAssembly code, has been marginalized (less than 1% of all binaries found on the web) and gives way to a diverse set of use cases. Finally, 29% of all binaries on the web are minified, calling for techniques to decompile and reverse engineer WebAssembly. Overall, our results show that WebAssembly has left its infancy and is growing up into a language that powers a diverse ecosystem, with new challenges and opportunities for security researchers and practitioners. Besides these insights, we also share the dataset underlying our study, which is 58 times larger than the largest previously reported benchmark.

This chapter shares large parts of its material with the corresponding publication [Hilbig et al. 2021]. The author of this dissertation has proposed the initial research project and supervised Aaron Hilbig as a bachelor student. The dissertation author then significantly extended the project, including collecting all non-Web binaries, re-implementing analyses, evaluating, and writing the majority of the paper.

4.1 MOTIVATION AND CONTRIBUTIONS

Despite its growing popularity, the WebAssembly ecosystem is severely understudied. To date, little is know about how the language is used, for what purposes, and how this affects the security of WebAssembly-based applications. In particular, we are interested in the following research questions:

RQ1: SOURCE LANGUAGES AND TOOLS WebAssembly is a compilation target, and in principle any programming language can be compiled to it. What languages are actually compiled to WebAssembly, how much do they contribute to the overall population, and what tools are used to produce the binaries? Answering these questions is relevant for understanding the impact of issues that specific source languages may have and for guiding future work toward source languages and toolchains prevalent in practice.

RQ2: ATTACK SURFACE In Chapter 3 we show that memory vulnerabilities in unsafe source languages, such as C and C++, can be exploited in WebAssembly binaries, sometimes even more easily than in native code. How large is the attack surface offered by real-world WebAssembly binaries, e.g., in terms of dangerous APIs these binaries import from JavaScript or in terms of vulnerable memory allocators they ship? Answering this question will increase our understanding of the threat posed by vulnerabilities compiled to the Web.

RQ3: CRYPTOMINING Previous results show [Musch et al. 2019a], and recent work assumes [Konoth et al. 2018; Musch et al. 2019b; Rüth et al. 2018; W. Wang et al. 2018], that WebAssembly is frequently used for cryptojacking, i.e., cryptomining performed in the browser of an unsuspecting client. Is cryptomining still an important threat today?

RQ4: USE CASES As a general purpose language, WebAssembly can serve many purposes in web applications and beyond. What are the typical use cases of WebAssembly? Given that the language is becoming more widely adopted, it is important to understand what its use cases are and how this affects the security of the Web.

RQ5: MINIFICATION AND NAMES The ability to understand WebAssembly binaries, e.g., for auditing third-party code or for reverse engineering malware, depends on whether binaries contain meaningful

names for program elements, e.g., functions. Do real-world WebAssembly binaries contain meaningful names or are they obfuscated?

Answering these and other questions requires a set of WebAssembly binaries that is (i) representative for how WebAssembly is used in the wild and (ii) large enough to cover the diversity of real-world WebAssembly usage. Currently, no such set of binaries exists.

The closest existing work is by Musch et al. [2019a], who report on a study of WebAssembly usage in the top one million websites. While inspiring, their study falls short in two respects. First, it has been performed at a point in time when WebAssembly was still in its infancy, with usage biased to early adopters, e.g., cryptominers, and a single toolchain dominating the ecosystem. Since then, many changes have happened, including higher browser adoption, alternative compilers that have become available, the shutdown of Coinhive (a common cryptomining platform) [Varlioglu et al. 2020], and the realization that vulnerabilities in insecure source languages can also be exploited in WebAssembly. Second, the methodology proposed by Musch et al. [2019a] focuses only on binaries found on the Web, and only on those that are executed when just visiting a website. By only looking into client-side web applications, WebAssembly on other platforms is disregarded, e.g., on Node.js, WebAssembly in browser extensions, and binaries for stand-alone WebAssembly runtimes.

This chapter presents a comprehensive empirical study of real-world WebAssembly binaries. The core of our work is WasmBench, a diverse set of 8,461 unique binaries gathered from a variety of sources, including querying source code repositories and package managers, searching the *HTTP Archive*, and crawling the Web. The binaries found with our methodology show that considering only a single one of these data sources would miss a significant fraction of the WebAssembly ecosystem. While we obviously cannot guarantee to cover all real-world WebAssembly usages, WasmBench provides not only a 58 times larger benchmark, but also a more diverse set of WebAssembly binaries, than the largest previously reported benchmark [Musch et al. 2019a].

With WasmBench, we address the above research questions through a combination of manual inspection, custom static analysis tools, and statistical analyses. Our findings include:

- Real-world WebAssembly binaries are compiled from a variety of source languages, including systems programming languages, such

as C, C++, Rust, and Go, higher level languages, such as AssemblyScript (a variant of TypeScript), and some rather unexpected languages, such as COBOL and Kotlin.

- The majority of WebAssembly binaries are compiled from memory-unsafe languages, from which vulnerabilities may propagate.
- 65% of all binaries and 44% of all functions in them use the *unmanaged stack* as described in Section 3.2, namely a portion of linear memory that is unprotected by the virtual machine and that can be exploited by attackers.
- 21% of all binaries import potentially dangerous APIs from their host environment, e.g., the infamous `eval`, APIs to modify the DOM from JavaScript, or system call-like APIs to interact with the network and file system on platforms outside the browser.
- Contrary to earlier findings [Musch et al. 2019a], cryptomining has dropped significantly in relevance, comprising only 1% of all binaries. Instead, we find applications with up to many millions of instructions that cover diverse use cases, including visualization, interactive shells for programming languages, media players, game engines, data compression, and natural language processing.
- 28.8% of all binaries on the Web are minified, calling for future work on decompiling and reverse engineering WebAssembly, to ensure that security analysts can understand web applications despite the presence of low level components.

Overall, our findings show that WebAssembly is "growing up", which leads to a larger and much more diverse ecosystem than in its early days. From a security perspective, this diversity has several implications. First, the fact that there are now many legitimate applications, and proportionally much fewer malicious ones, shifts the focus from detecting *malicious* code to handling *vulnerable* code. Second, the large fraction of binaries that originate from "insecure" source languages, in particular C and C++, shows the risk that their problems, e.g., memory vulnerabilities, will now propagate to the Web. Mitigations against such memory vulnerabilities are becoming an important goal to keep the Web safe. Third, the many different compilation toolchains and their variants, e.g., in terms of memory allocators compiled into the binaries, create a potentially large attack surface. Automated tools to analyze and improve the security of WebAssembly binaries are needed.

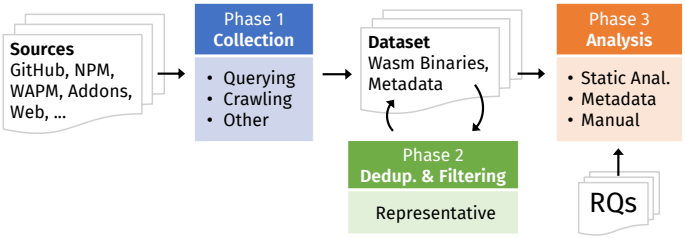


Figure 4.1: Overview of the phases of our methodology.

CONTRIBUTIONS In summary, this chapter contributes:

- The first comprehensive study of WebAssembly binaries gathered from multiple sources, including client-side web applications, package managers, and source code repositories;
- A combination of automated program analyses, manual inspection, and statistical analysis to answer research questions about the security, source languages, and use cases of WebAssembly;
- Empirical evidence and insights about security-related properties of real-world WebAssembly, some of which update earlier findings and many of which call for future work on mitigation techniques and analysis tools;
- By far the largest benchmark of WebAssembly binaries, which we make available as a basis for other studies and as a benchmark for future tools: `https://github.com/sola-st/WasmBench`.

4.2 METHODOLOGY

Our methodology is split into three phases (Figure 4.1). In the collection phase, we obtain a large set of WebAssembly binaries from a variety of sources. We select sources to cover WebAssembly in different contexts and at different stages of deployment. Sections 4.2.1 to 4.2.4 present how we collect WebAssembly binaries from source code repositories, package managers that distribute deployed software, archived and live websites, and through manual search, respectively. Figure 4.2 gives an overview of the different sources we collect binaries from. Alongside each binary, we also collect metadata, e.g., on which website a binary was found. All activities related to collecting binaries were done be-

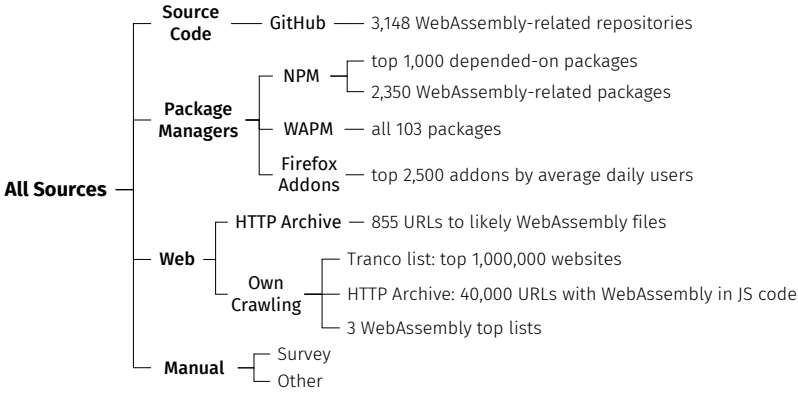


Figure 4.2: Sources from which we collect binaries.

tween April and September 2020. Overall, the collection phase results in 51,148 binaries, including duplicates and binaries that are not representative for real-world usages of WebAssembly. Section 4.2.5 presents the filtering phase, where we filter and deduplicate these binaries into a set of 8,461 unique binaries that serve as the basis for our study. Finally, the third phase analyzes the set of binaries through a combination of static code analysis, analysis of metadata associated with the binaries, and manual inspection. We present the analysis phase along with its results in Section 4.3.

To the best of our knowledge, no prior work has gathered WebAssembly binaries from such a diverse set of sources. As a result, the number of binaries we obtain is 58 times larger than the largest set studied so far (147 unique binaries) [Musch et al. 2019a]. Our experimental results (Section 4.3.2) show that the sources we consider WebAssembly binaries from complement each other, i.e., considering all of them is crucial to obtain a representative dataset.

4.2.1 *Collecting Binaries from Repositories*

Our first method for collecting binaries looks into source code repositories. Even though WebAssembly is a binary format, developers often store binaries into source code repositories, e.g., to ease the installation of a project or to include third-party libraries. To gather such binaries,

we clone all public repositories that are in the top 1,000 results of four queries to the GitHub search API:

- Repositories where "wasm" or "WebAssembly" is in the repository name or description (i.e., two queries).
- Repositories that are tagged with "WebAssembly" as one of the used programming languages.
- Repositories tagged with the topic "WebAssembly".

Overall, the queries result in 3,148 repositories, which we clone, and then search for files ending in `.wasm`.

4.2.2 *Collecting Binaries from Package Managers*

Once developers deploy a WebAssembly-based application, it is often made available through a package manager. We consider three software ecosystems that use WebAssembly.

NPM PACKAGES The Node Package Manager (npm)[1] distributes JavaScript packages, some of which internally use WebAssembly code. Packages distributed via npm are typically used on the client side or in server-side applications with Node.js. To find npm packages that contain WebAssembly binaries, we gather two sets of packages. First, from the full registry file of npm we compute the top 1,000 most depended-upon packages. Second, we query npm for all packages that match at least one of the keywords "wasm" and "WebAssembly", which yields 2,350 packages. We install these packages and their transitive dependencies, and then search the resulting 7,198 packages for `.wasm` files.

WAPM PACKAGES The WebAssembly Package Manager (wapm)[2] specializes on distributing WebAssembly code. Most of the wapm packages are intended to run on stand-alone WebAssembly runtimes. Unlike for npm, we can afford to analyze all 103 available packages. We install all packages and again extract all `.wasm` files.

FIREFOX BROWSER ADD-ONS Browser extensions, traditionally implemented in JavaScript, nowadays can also make use of WebAssembly code. To gather binaries used in browser extensions, we download the top 2,500 Firefox add-ons from `https://addons.mozilla.org`, as

1 https://www.npmjs.com/
2 https://wapm.io/

measured by average daily users. We then unpack the extensions' XPI archives, and search again for `.wasm` files.

4.2.3 *Collecting Binaries from Websites*

Collecting WebAssembly binaries from the Web involves several challenges. First, as the Web is too big to be crawled in its entirety, finding suitable starting points for exploring it is crucial. Second, even when visiting a WebAssembly-powered website, it is non-trivial to identify and collect WebAssembly binaries from it. Some sites embed WebAssembly modules into JavaScript source code, e.g., as Base64-encoded strings that are decoded and instantiated at runtime. For such sites, we must detect WebAssembly modules when they are executed. Other websites spawn requests for WebAssembly modules but never execute them during our collection process, e.g., because execution relies on specific user inputs. A purely dynamic methodology would miss such binaries that are loaded but not executed initially.

We address the first challenge, finding good websites as starting points, through two techniques. On the one hand, we can build on results from the *HTTP Archive* for finding sites known to contain WebAssembly binaries (Section 4.2.3.1). On the other hand, for our own crawling, we systematically start from potentially WebAssembly-related seed URLs (Section 4.2.3.2). To address the second challenge of detecting WebAssembly binaries during crawling, we analyze all websites with a combination of static and dynamic detection techniques.

4.2.3.1 *Direct Downloads Guided by HTTP Archive*

The *HTTP Archive* project[3] regularly crawls the Web and makes the requests and responses available. Starting from URLs obtained from the Chrome User Experience Report , the project currently covers over 5 million top-level domains, monthly. We here focus on websites crawled using the desktop version of Google Chrome, which we access via Google's BigQuery[4] database system.

We search in the HTTP Archive tables for responses that are likely WebAssembly binaries and then directly download the corresponding files. To this end, we query two tables, from months May and June

3 https://httparchive.org/
4 https://cloud.google.com/bigquery/

2020, which contain information about all requests made while crawling the websites, and the corresponding responses. These tables, called `summary_requests` are 434.4 GB and 476.7 GB in size. We then filter all requests in the tables as follows. We keep only those MIME types that are commonly used to serve WebAssembly, such as `application/wasm` and `application/octet-stream`, and all requests where `.wasm` appears in the URL. These queries result in a set of 855 URLs. We download files from each of these URLs using `wget` and keep all that start with `\0asm`, WebAssembly's magic number.

4.2.3.2 *Web Crawling*

The HTTP Archive-guided search covers a wide range of top-level domains, but it may miss WebAssembly binaries on websites not covered by crawling a generic list of websites and binaries that one cannot identify based on their MIME type. To collect additional binaries, we also perform our own web crawling. There are three components to our crawling: the seed list, the crawling algorithm, and methods for detecting WebAssembly.

SEED LISTS Any kind of web crawling requires a *seed list* of URLs to start from. We consider three seed lists, one generic list of popular websites and two lists targeted specifically at WebAssembly:

- *Top one million websites.* As a generic set of websites to explore, we start crawling from the one million most popular websites on the *Tranco* list [Pochat et al. 2019], a top list more resilient to manipulation than the commonly used Alexa list.
- *"WebAssembly" in JavaScript files.* WebAssembly binaries on websites must be executed by some surrounding JavaScript code, e.g., by calling `WebAssembly.instantiate`. To identify websites with such JavaScript code, we query a table provided by the HTTP Archive that stores the full bodies of all HTTP responses up to some size. We search this table, which has a total size of 9.32 TB, with Google BigQuery for all JavaScript responses that contain `WebAssembly` and add the URLs of the corresponding websites to our seed list, which results in about 40,000 URLs.
- *WebAssembly top lists.* As the most targeted seed list, we start crawling from three hand-curated lists of WebAssembly-related websites.

These websites cover projects using WebAssembly[5], tools and demos[6], and WebAssembly-based games[7].

CRAWLING ALGORITHM Given a seed list, our crawler visits each URL on the list and recursively follows links on the visited websites. The crawler visits each URL, with up to one retry. If the website is loading successfully, the crawler waits until either the "DOM content loaded" event is fired and all network connections have become idle, or until a 30-second timeout occurs. The crawler collects all WebAssembly binaries loaded or executed in this time (details below). For each visited website, the crawler extracts more URLs to explore from the `href` attribute of all `<a>`-tags on the site.

To control the amount of sites to visit, the crawler is configured with two parameters: the recursion depth d, which bounds how many links away from the seed URLs to explore, and the exploration breadth b, which bounds how many links to follow on each explored site. If a site has more than b links, the crawler picks b of them at random. For the first two seed lists, we set $d = b = 2$, i.e., the crawler visits at most seven sites per URL in the seed list. Because the third seed list is the most focused one, we explore it more thoroughly with $d = 7$ and $b = 3$, and repeat the exploration with 16 separate crawler instances. We chose those parameters based on preliminary experiments, to find most binaries in a given time budget. We also set a proper user agent to improve chances of not being detected as a bot.

IDENTIFYING WEBASSEMBLY BINARIES For each website visited by the crawler, we use a combination of two techniques to identify WebAssembly binaries on the site. Our first detection mechanism intercepts the network traffic between the crawler and the website using a local proxy[8] that inspects the headers and contents of all requests and responses. To identify WebAssembly modules, we check if the content-type header matches `application/wasm` or `application/octet-stream`, or if the URL contains `.wasm`, and then ensure that the response payload starts with the proper magic number. If this is the case, we store the loaded file as a WebAssembly binary. The key advantage of this detection mechanism is that it detects WebAssembly modules even if

5 https://madewithwebassembly.com/
6 https://github.com/mbasso/awesome-wasm
7 https://www.webassemblygames.com/
8 https://mitmproxy.org/

they are not executed during the crawler's visit of the website. The second detection mechanism tracks calls to APIs used for instantiating WebAssembly modules, as proposed in prior work [Musch et al. 2019a]. We transparently overwrite built-in JavaScript functions, such as `WebAssembly.instantiate`, and analyze its invocations. In contrast to the first detection mechanism, this mechanism can detect WebAssembly binaries that occur inline in JavaScript code, if executed.

4.2.4 *Collecting Binaries Manually*

In addition to automatically collecting WebAssembly binaries, we also gather a small number of binaries manually. On the one hand, we collect binaries through manual interaction with the Web in daily browsing between April and September 2020. On the other hand, we asked WebAssembly developers on `https://reddit.com/r/WebAssembly` in June 2020 for binaries they are willing to share. As discussed in the results, these two manual collection methods complement our automatically collected binaries with otherwise missed examples.

4.2.5 *Deduplication and Filtering*

After collecting binaries and associated metadata from the aforementioned sources, we remove duplicates and filter binaries that are not representative of real-world applications. To deduplicate binaries, we compare files based on their SHA256 hash and remove identical files. Unless mentioned otherwise, our study focuses in the deduplicated dataset. In addition to deduplication, we remove binaries that are non-representative of real-world applications, because they fall into at least one of the following categories. Binaries that occur multiple times, e.g., across different sources, are only removed if all occurrences of it were filtered out.

- *Automatically generated binary variants:* Some GitHub repositories contain binaries generated by research tools, e.g., to fuzz-test WebAssembly implementations, to superoptimize WebAssembly binaries [Arteaga, Donde, et al. 2020], or to perform automatic code diversification [Arteaga, Malivitsis, et al. 2021]. Since these tools turn a single binary into many, only slightly different variants, we remove the generated variants. We identify those variants by filename (e.g., `*.opt.wasm`) and path (e.g., binaries in `afl_out/`).

- *Test suites:* On GitHub and in some npm packages, we find binaries that are used as test inputs for WebAssembly-related tools, e.g., parsers, compilers, and virtual machines. One large portion are binaries from the official specification test suite, which often test only a single instruction or language construct. We identify them by typical repositories and paths (e.g., files in `spectest/`).
- *Tutorial projects:* Many npm projects and some GitHub repositories are instances of users following WebAssembly tutorials for particular tool chains.[9] Those binaries are small and all very similar. We identify them based on common binary (e.g., `hello_world_bg.wasm`) and project names (e.g., `test-wasm@0.0.1`).
- *Small and invalid binaries:* Finally, we remove binaries that contain ten or fewer instructions, and binaries that cannot be validated by the reference WebAssembly Binary Toolkit (WABT), even with all language extensions enabled.

4.3 RESULTS

Based on the collected WASMBENCH dataset of real-world WebAssembly binaries, we address the research questions (RQs) described in the introduction. For each research question, we detail the analyses performed on the dataset, the direct results, and then interpret those to obtain *insights*, i.e., take-away points, often with a focus on security. Before that, Sections 4.3.1 and 4.3.2 also explain our experimental setup and give an overview of the dataset.

4.3.1 *Implementation and Experimental Setup*

The crawler is implemented based on Puppeteer and Puppeteer Cluster, two Node.js libraries for controlling instances of the Chromium browser, here version 83.0.4103.0. The static analyses described in the following are implemented in several Rust programs to statically extract relevant features, such as instructions, names, etc. from the binaries, complemented by Python scripts that perform the final analyses. For parsing binaries, we use the *wasmparser* library, a project by the Bytecode Alliance. All experiments were run on an Ubuntu 18.04 machine with two Intel Xeon CPUs at 2.2 GHz running 48 threads,

9 For example, the *rustwasm* book: `https://rustwasm.github.io/docs/book/`.

Table 4.1: Contribution of different sources to the dataset.

Source (see Figure 4.2)	Found Binaries			
	Total	Unique	Filtered	Only
GitHub, search: wasm	44,218	21,117	6,830	6,641
NPM, top dependend-upon	14	8	8	3
NPM, search: wasm	3,488	2,452	1,163	1,036
WAPM, all	122	113	108	81
Firefox add-ons, top by users	29	17	17	15
Web, HTTP Archive	261	141	141	54
Web, crawling, with seed list:	2,923	432	412	298
HTTP Archive	2,046	268	254	128
Tranco top websites	769	167	164	86
WebAssembly top lists	108	89	76	43
Manual	93	89	88	57
All Sources	51,148	23,413	8,461	

equipped with 256 GB of memory. The crawling was run in chunks of 50,000 websites, where each chunk took about 7 hours to finish, with a total of about 10 days for all crawling. The static analyses usually finish within several minutes for the entire dataset. Our entire dataset and the implementation are available for others to build on at `https://github.com/sola-st/WasmBench`.

4.3.2 *Overview of Dataset*

Table 4.1 gives an overview of our dataset. For each source, the table shows how many binaries we found, and how many remain after deduplication and filtering. The last column shows how many binaries are found only via a single source, illustrating the importance of particular collection methods for obtaining a diverse dataset.

SOURCES The largest contribution to the dataset are the GitHub repositories and packages from npm. Given that WebAssembly binaries on websites or in arbitrary packages are still relatively scarce, selecting repositories and packages related to WebAssembly is an effective way of finding binaries. At the same time, the sources where we do not

Table 4.2: Binaries filtered out due to different criteria.

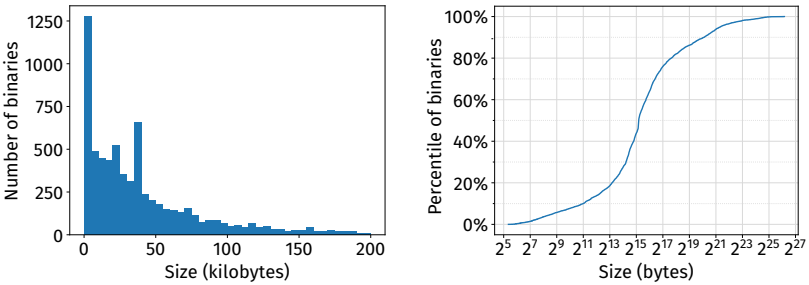
Filter	Removed Binaries	
	Total	Unique
Generated binary variants, of those:	9,279	8,048
in CROW [Arteaga, Malivitsis, et al. 2021] repository	8,025	7,987
Test suites and files, of those:	26,138	5,283
variants of WebAssembly spec suite	25,133	4,931
Invalid WebAssembly binaries	13,593	3,513
Small binaries: <10 instructions	10,146	1,881
Tutorial projects, of those:	848	633
`hello-wasm` projects	695	506
All Filters	37,808	14,952

query for WebAssembly specifically (i.e., most of the web crawling, the top packages from npm, and Firefox add-ons) still show that WebAssembly binaries are found in the wild in popular projects used by millions of users.

For the Web, we also see that all our four sources are essential for finding a diverse set of WebAssembly binaries. Only crawling the top one million websites would miss at least 225 unique binaries. The fact that the seed lists from HTTP Archive and the WebAssembly top lists are much smaller than the list of the top one million websites, yet the number of found binaries are similar or even higher, shows that a targeted seed list for crawling is key to finding otherwise missed binaries.

Insight 1. All methods we use to collect binaries contribute in a non-negligible way. Combining different sources and collection techniques is crucial for obtaining a diverse dataset of WebAssembly binaries.

FILTERING Deduplication and filtering non-representative binaries (Section 4.2.5) significantly reduces the dataset (Table 4.2). In particular, one repository that contains generated variants of input binaries is important to filter, as it otherwise accounts for almost 8,000 unique binaries. From the second category, test suites, we also see that test binaries are commonly reused across many projects (26,138 occurrences, but only 5,283 unique binaries), and that most of them are from the offi-



(a) Histogram of the lower 80% of the sizes of the binaries.

(b) Cumulative distribution of the binary sizes (in $\log_2$ scale).

Figure 4.3: Distribution of binary sizes.

cial specification test suite. The last filter removes binaries from a few, very similar tutorials, including more than 500 binaries from projects called `hello-wasm`.

BINARY SIZES As a first proxy for the diversity of the collected binaries, we look into their sizes and instruction count. Figure 4.3 shows the histogram and cumulative distribution of binary sizes in bytes. The distribution of the number of instructions is similar in shape and omitted for brevity. While there are many small binaries, there is a long and heavy tail towards larger sizes. Two thirds of the binaries are larger than 20 KB and have more than 8,700 instructions. The median binary is 37.1 KB large and has 14,885 instructions. The largest binaries are a WebAssembly port of TiDB[10], a distributed SQL database written in Go, with 75.1 MB and 16.9M instructions, respectively, found as a wapm package; opencascade.js[11] (65.8 MB, 22.2M instructions), a WebAssembly port of an open source C++ CAD library, found on npm and GitHub; and finally the Clang compiler, itself compiled to WebAssembly (46.7 MB, 12.6M instructions), found on wapm and GitHub.

Insight 2. Complex, real-world applications are compiled to WebAssembly. Two thirds of all binaries contain more than 10,000 instructions, and some binaries are from large, well-known projects, compiled from millions of lines of code.

10 https://github.com/pingcap/tidb
11 https://github.com/donalffons/opencascade.js

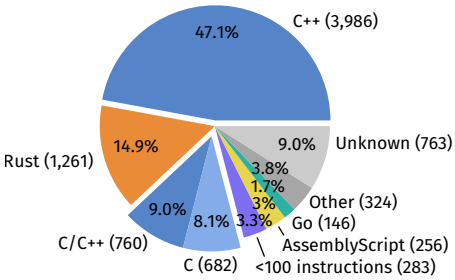
4.3.3 *RQ1: Source Languages and Tools*

Given WebAssembly's goal of being a universal bytecode, we study which languages are compiled to it in practice, and which toolchains are used in the process.

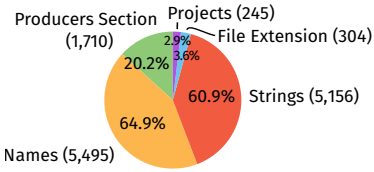
ANALYSIS It is non-trivial to infer from a binary which source language and compiler has produced it. We rely on several complementary methods. First, we check the optional *producers* section, where some toolchains explicitly encode the source language(s) a program is compiled from.[12] Second, our analysis searches for *characteristic function names* that appear in the *import* section, the *export* section, or the optional *name* section. For example, `_ZdaPv` is the name-mangled `delete` operator of C++; or `runtime.gostring` is a Go runtime library function. Overall, we identify characteristic function names for C++, C, Rust, Go, AssemblyScript, Kotlin, and FStar. Third, the analysis searches for *characteristic strings* among all sequences of more than three ASCII characters in the *data* section. For example, being core types, `Result::unwrap` and `Option::unwrap` frequently appear in error messages of the Rust standard library. We identify characteristic strings for C++, Rust, Matlab, and COBOL. Fourth, if none of the above work, we analyze *sibling files* of binaries collected from code repositories and package managers. Sibling file here means a file in the same directory that shares the file name except for the extension. We take into account extensions for C, C++, Rust, Go, AssemblyScript/TypeScript, the WebAssembly text format (`.wat`/`.wast`), and several smaller languages. Finally, for some source code repositories and packages with multiple unidentified binaries, we *manually inspect* source code, build scripts, and binaries. For each of the automated methods above, we manually inspect binaries and the predictions to confirm that our heuristics are precise. For binaries where multiple methods identify the source language, we confirm that the predictions are consistent.

RESULTS Figure 4.4a shows the inferred source languages. We see that almost two thirds (64.2%) of the binaries are compiled from C, C++, or a combination of both. Given that these are memory-unsafe languages, plagued with decades of vulnerabilities [Szekeres et al. 2013] and that WebAssembly binaries are not automatically safe from exploitation (Chapter 3), this result is highly worrying.

12 https://github.com/WebAssembly/tool-conventions/blob/main/ProducersSection.md



(a) Source languages.



(b) Analysis methods (multiple can apply per binary).

Figure 4.4: Source languages and methods for inferring them.

Insight 3. Almost two-thirds of all collected binaries are compiled from memory-unsafe source languages. These results and those in the next section call for techniques to analyze and ensure WebAssembly's binary security.

Rust comes in second place with 14.9% of all binaries, followed by AssemblyScript (3%) and Go (1.7%) as source languages with official WebAssembly support. Finally, there is a longer tail of other languages, often used in single projects: Matlab[13] (0.69%), FStar[14] (0.33%), CHIP-8[15] (0.26%), several binaries compiled from toy languages, and even a single instance of COBOL. A small portion of binaries (1.1%) is translated directly from the WebAssembly text format, i.e., likely to be written by hand. Finally, for 3.3% of all binaries we could not assign a source language, but since they contain less than 100 instructions, they are also likely to be written manually.

Insight 4. In addition to C and C++, various other languages are compiled to WebAssembly, including languages with garbage collection and heavier runtimes (Go, Matlab). This result matches WebAssembly's goal of serving as a universal bytecode. It also means binary analysis will become more important, since source code is not always available, and even if it is, implementing separate analyses for many languages is impractical.

13 https://github.com/Sable/matwably
14 https://github.com/FStarLang/kremlin
15 An 8-bit VM language from 1970s, https://github.com/pepyakin/emchipten

We also analyze the tools used to produce the binaries. For 20.2% of the binaries, the producers section explicitly mentions them in the `processed-by` field. 10.8% of all binaries explicitly mention being produced by Clang. No binaries mention Emscripten because it does not emit a producer section, unlike newer versions of Clang. These results show that Emscripten is no longer the only way to compile C and C++ to WebAssembly. Other tools that appear in the `producer` section are `rustc` (9.5%), `wasm-bindgen`[16] (7.9%), a JavaScript host-code generator, and `walrus` (7.5%)[17], a binary transformation library, and the official Go compiler (0.4%). Since all compilers from Rust, C, and C++ to WebAssembly are based on LLVM, we can also derive that 79.1% of the binaries are produced with the help of LLVM.

Insight 5. Almost 80% of all binaries are compiled with the help of the LLVM toolchain. This implies that security mitigations, such as stack canaries, would have a large effect on the ecosystem if implemented in this toolchain.

Figure 4.4b shows which of our methods are most effective at inferring the source language. 20.2% of the binaries contain a producers section, from which the source code language can be directly obtained. Characteristic names and strings are also important inference methods, since they apply to 64.9% and 60.9% of binaries, respectively. Overall, our methods infer the source language for 91% (7,698) of the 8,461 unique binaries.

4.3.4 *RQ2: Vulnerabilities Propagated from Source Languages*

In Chapter 3 we have shown that memory vulnerabilities in unsafe source languages can propagate to WebAssembly binaries, and may sometimes be exploited even more easily than for native binaries. In this previous work, we have evaluated the risks of such attacks on a small set of 26 binaries, most of which are compiled C/C++ benchmarks. It remains unclear to what extent propagated vulnerabilities may affect larger sets of WebAssembly binaries.

We address this question by studying three important characteristics of binaries that attackers can abuse: (i) uses of the unmanaged stack, i.e., the unprotected portion of linear memory set aside for function-scoped

16 https://github.com/rustwasm/wasm-bindgen
17 https://github.com/rustwasm/walrus

data (Section 4.3.4.1); (ii) unsafe memory allocators compiled into a binary, which attackers can abuse as a memory write primitive (Section 4.3.4.2); and (iii) accesses to potentially dangerous APIs imported from the host environment (Section 4.3.4.3). For (i) we refine the static analysis from Section 3.5 and consider a 325 times larger dataset. For characteristics (ii) and (iii), this work is the first to systematically evaluate their prevalence in real-world WebAssembly binaries. We study all binaries, irrespective of the source language, because the problem of propagated vulnerabilities may affect all languages with memory-unsafe behavior, in particular C, C++, but also, e.g., Rust, as its `unsafe` keyword is commonly used [Evans et al. 2020] and can cause memory-safety related vulnerabilities [H. Xu et al. 2021].

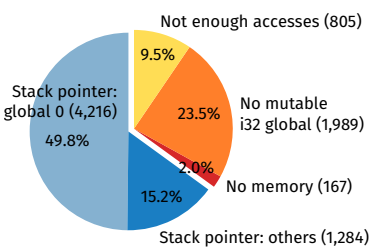
4.3.4.1 *Usage of the Unmanaged Stack*

The so-called *unmanaged stack* is a region within linear memory of a WebAssembly program that holds, e.g., non-primitive data with function lifetime. This design is motivated by the fact that all non-scalar data and all data of which an address is taken cannot be put in WebAssembly's locals or globals, but must instead reside in linear memory (Section 3.2). Given that buffer overflows on the unmanaged stack can overwrite across stack frames and even into supposedly "constant" data, this makes the unmanaged stack a more dangerous exploitation target than even in native programs. For this reason, we evaluate how many binaries use it in practice.

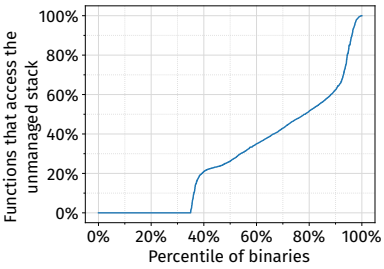
ANALYSIS To analyze the usage of the unmanaged stack, our static analysis performs two steps. First, it tries to identify the stack pointer to determine whether a binary uses an unmanaged stack at all. Out of all global variables, the analysis selects the one that

- has type `i32`, i.e., the type of all pointers,
- is declared mutable, to exclude constants like `STACK_MAX`,
- is the most read and written global, as determined by the number of `global.get` × `global.set` instructions[18], and
- has at least three reads and at least three writes, to avoid false positives in small binaries.

18 The product of the counts prefers globals which are similarly often read and written. This is true for the stack pointer, but not for other frequently accessed pointers.



(a) Proportion of binaries with an identified stack pointer, and if not why.



(b) Cumulative distribution of the percentage of functions in a binary that use the unmanaged stack.

Figure 4.5: Usage of the unmanaged stack in binaries.

We manually validate that these heuristics identify the stack pointer reliably on randomly sampled binaries. If the analysis cannot identify a stack pointer, it conservatively assumes that the binary does not use an unmanaged stack. Second, once the stack pointer is identified, the analysis counts the number of functions in the binary that access the stack pointer somewhere in their body. Our implementation builds upon the analysis described in Section 3.5.2, but uses a newer WebAssembly parser to handle language extensions and make the analysis robust enough to run on thousands of real-world binaries.

RESULTS Figure 4.5a shows that almost two thirds (65%) of all binaries use the unmanaged stack. While most of them use the global with index 0 as their stack pointer, our heuristics to identify the stack pointer are important, since 15.2% of all binaries use another global variable. For 2% of the binaries, the analysis can clearly determine that they have no unmanaged stack in linear memory, simply because there is no linear memory at all. For 23.5% of the binaries, there is a linear memory section, but no mutable `i32` global that could be a stack pointer. Finally, 9.5% of all binaries have at least one candidate mutable global pointer, but it is not accessed often enough for our analysis to consider it the stack pointer. Interestingly, AssemblyScript programs are in the last category, since its runtime seems to not support stack allocation.

To better understand how much a binary uses the unmanaged stack, Figure 4.5b shows how many of the functions in a binary access the stack pointer at least once. Consistent with Figure 4.5a, in 35% of all

binaries no function uses the stack pointer because none is present. In the median binary, already 33% of all functions use the stack pointer, and in some binaries almost every function uses the unmanaged stack. On average across all binaries *with* an unmanaged stack, 44% of their functions make use of it.

Insight 6. Many binaries (65%) and functions in those binaries (44%) use the unmanaged stack, which attackers may abuse for runtime exploitation. This result extends our findings from Section 3.5 to a much larger and more diverse set of real-world binaries.

4.3.4.2 *Statically Linked Allocators*

WebAssembly's memory organization is very low-level. Besides the single linear memory section, which can be expanded at runtime with the `memory.grow` instruction, no help with allocating memory is provided by the language (Section 2.2). Subdividing the linear memory, e.g., to avoid fragmentation and to reuse space of deallocated objects, needs to be handled by an allocator that is statically linked into the binary. Especially for binaries on the Web and for smart contract platforms, code size is an important consideration, so developers can choose a lightweight allocator instead of the default allocator provided by the compiler. Our attacks in Chapter 3 have shown that those smaller allocators can lack important mitigations against heap metadata corruption and yield powerful arbitrary write primitives for an attacker. However, it remains unclear what allocators developers use in practice.

ANALYSIS To identify allocators in a binary, we rely on similar heuristics as for source language detection (Section 4.3.3). That is, we first identify allocators by characteristic *function names* in binaries, if those are available. Then, we inspect frequent *strings* in the `data` section of binaries, e.g., for error messages of certain allocators. We have validated our heuristic, e.g., by searching for the source code of allocators online and comparing it with our findings.

RESULTS Figure 4.6 shows our results, grouped into three categories. In blue, we mark default allocators provided by programming languages and compilers, in different shades of red we mark other allocators that we identified, and in gray when we could not identify an allocator. In terms of default allocators, we see that 16.9% of all binaries use *dlmalloc*,

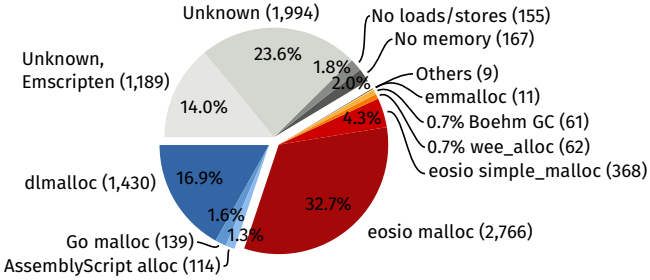


Figure 4.6: Identified memory allocators in binaries (multiple can apply).

the default allocator provided by Emscripten, Clang, and the Rust compiler when targeting WebAssembly. Go and AssemblyScript allocators are present roughly in the proportion of their respective languages.

Among the non-default allocators, two dominate, which are present in 32.7% and 4.3% of our binaries. They are both from EOSIO, a smart contract platform that uses WebAssembly as its bytecode.[19] Those contracts can be written in C++ and compiled with Emscripten. However, most of them are not using Emscripten's default allocator. While we did not perform an in-depth security analysis of EOSIO `malloc` and `simple_malloc`, both are considerably shorter in terms of lines of code and do not feature any assertions that would guard against metadata corruption. In our dataset, we also find `wee_alloc` (62 binaries) and `emmalloc` (11 binaries), two small allocators for Rust and Emscripten respectively, that we already found to be vulnerable against heap metadata corruption attacks (Section 3.3.1.3). Other interesting custom allocators are Boehm GC (a mark-and-sweep garbage collector) and gperftools, in several binaries collected from Google domains.

Insight 7. WebAssembly binaries come with a variety of memory allocators, including many custom allocators (38.6%). This increases the risk of including a vulnerable allocator. If code size is the motivation to use custom allocators, a more secure alternative could be a memory allocation or garbage collection API provided by the host environment [Rossberg 2019b].

19 https://eos.io/

4.3.4.3 *Imports of Security-Critical APIs from Host Environment*

To exploit a WebAssembly binary, an attack proceeds in two steps. The first step is compromising the state or behavior of the WebAssembly binary itself, e.g., by exploiting an unsafe allocator (Section 4.3.4.2) or a buffer overflow on the unmanaged stack (Section 4.3.4.1). The second step is actually performing the malicious action to the underlying system. The only way to do so, assuming VM implementations are bug-free and host security is perfect,[20] is to call functions imported into the WebAssembly binary from the host environment. For example, an attacker could pass an injected string on the unmanaged stack to an imported function, e.g., JavaScript's `eval`. To estimate how often WebAssembly binaries use such security-critical host APIs, we thus analyze their imports.

ANALYSIS We identify imported security-critical APIs based on their import name in the WebAssembly binary. Going by name (instead of implementation) is necessary because, (1) the implementation of an imported function is supplied by the host only at instantiation-time, so it is not available when given only the binary; and (2) there are many host environments, not all of which are using JavaScript. WASI for example, defines imports that can be implemented by different stand-alone WebAssembly VMs in native code. We thus identify import names for which the host implementation is likely a security-critical function. E.g., the import `emscripten_run_script` in WebAssembly binaries is typically bound to Emscripten-generated JavaScript code that calls `eval`. We match imports against 18 patterns in five categories known to be potentially security-critical APIs:

- *Code execution.* Imports like `eval`, `exec`, or `emscripten_run_script`.
- *Network access.* Imports containing `XHR`, `request`, `http`, or `fetch`.
- *File I/O.* Imports containing `file`, `fd`, or `path`.
- *DOM interaction.* Imports containing `document`, `html`, `body`, or `element` could manipulate the DOM, which can lead to cross-site scripting.
- *Dynamic linking.* `dlopen`, `dlsym`, and `dlclose` allow loading additional code at runtime, which can lead to code injection.

20 Obviously, host security is not perfect, looking at past attacks against WebAssembly VM implementations [Plaskett et al. 2018; Silvanovich 2018].

Table 4.3: Imports matching potentially security-critical APIs.

Category	Patterns	Matching		
		Imports	Binaries	%
Code execution	`eval`, `exec`, `execve`, `emscripten_run_script`	383	160	1.9%
Network access	`xhr`, `request`, `http`, `fetch`	944	172	2.0%
File I/O	`file`, `fd`, `path`	7,532	1,610	19.0%
DOM interaction	`document`, `html`, `body`, `element`	1,720	212	2.5%
Dynamic linking	`dlopen`, `dlsym`, `dlclose`	352	138	1.6%
At least one		10,468	1,797	21.2%

To avoid spurious matches, especially for short patterns like `fd`, we tokenize import names based on camel-case and non-alphabet characters, and then check for a pattern to occur verbatim in the token sequence. E.g., `fd_write` matches our file I/O category, but `bufdelete` does not. We manually inspect matches to ensure they are plausible and remove benign matches otherwise.

RESULTS Table 4.3 shows the results of our name-based import analysis. The first two columns show the category and patterns we match import names against. In the third column, we count imported functions that match at least one pattern. The last two columns show the number of binaries with at least one matching import, and which fraction of the filtered dataset this corresponds to. We see that imports related to file I/O, the most common category, are present in almost every fifth binary. Interestingly, even though WebAssembly was originally not meant to replace JavaScript, but rather for compute intensive applications, still 212 binaries likely interact with the DOM from WebAssembly, which attackers could use for cross-site scripting. In the last row, we see that overall 21.2% of all binaries import at least one potentially security-critical API.

Insight 8. Many binaries (21.2%) import potentially dangerous APIs from their host environment, which may allow compromised binaries, e.g., to inject arbitrary code or to write to the file system.

4.3.5 *RQ3: Cryptomining*

A study of real-world uses of WebAssembly performed in early 2019 [Musch et al. 2019a] reports cryptomining to be one of the most common use cases of WebAssembly on the Web. That study found 55.7% of the analyzed websites to use WebAssembly for cryptojacking, i.e., the practice of using a website visitor's hardware resources for mining cryptocurrencies without their consent. Identifying and controlling cryptominers on the Web has been the focus of several recent pieces of work [Konoth et al. 2018; Musch et al. 2019b; Rüth et al. 2018; W. Wang et al. 2018]. In this research question, we study whether cryptomining is still an important threat today. We address this question in two ways. First, we analyze those binaries we collected from the Web for signs of being cryptominers. Second, we directly compare the binaries gathered in earlier work with our dataset.

4.3.5.1 *Analyzing Binaries Found on the Web*

To understand the prevalence of cryptomining today, we analyze all binaries collected from the Web, i.e., direct downloads guided by HTTP Archive and the results of our own crawling, using VirusTotal. The VirusTotal API allows to upload and scan files with up to 70 independent third-party antivirus scanners and malware detectors, and reports back the number of positive results. Among the 352 analyzed binaries, VirusTotal reports four files to contain malicious content. Three of them are likely to be the same program, as they have similar sizes (68.8 ± 0.7 kB) and the same distribution of instructions. These three files are detected by 26 or more scanning tools employed by VirusTotal. One of the files is collected from `http://monero.cit.net`, which further supports the presumption that the binary is a cryptominer, as "Monero" is the name of a common cryptocurreny. Moreover, one of three binaries is identical to a binary we collect also from a GitHub repository called "CryptoNoter"[21], which is an open-source Monero cryptominer. The fourth file reported by VirusTotal is tested positive by only one scanner. Our manual analysis shows that the report for this binary is likely to be a false positive.

21 `https://github.com/JayWalker512/CryptoNoter`

4.3.5.2 *Comparison with Dataset by Musch et al. [2019a]*

The previous study is based on 147 unique WebAssembly binaries, which the authors kindly shared with us. The intersection of their dataset with ours contains 23 binaries, i.e., 16% of their dataset and 0.2% of our dataset. To better understand these binaries, we manually examine them and visit the corresponding websites. We find four of the 23 binaries to be suspicious. Two of them are among the files flagged by VirusTotal, as discussed above. For one file from a website that declares itself to be a "blockchain explorer", we could not observe any suspicious activity when visiting the source website, but also could not identify its functionality, and thus declare it to be suspicious. For the last suspicious binary, visiting the corresponding website increases CPU load to 70%. The website offers a service to mine cryptocurrency and openly advertises the fact that one can start mining immediately in the browser. That is, the binary is an example of crypto*mining* but not illicit crypto*jacking*.

In summary, we identify only four binaries from the web portion of our dataset as possible cryptominers (about 1% of the dataset), three of which appeared to be inactive when visiting the website. While our analysis may miss cryptomining binaries, a risk one could reduce through additional analysis techniques beyond those provided via VirusTotal [Musch et al. 2019a; W. Wang et al. 2018], the prevalence of cryptomining seems to have dropped significantly over the past one to two years. This result is also confirmed by a manual analysis of WebAssembly binaries found on the Web (Section 4.3.6.2) and is in line with other reports [Varlioglu et al. 2020] that cryptomining became less appealing after one of the major cryptomining script providers, Coinhive, shut down. Varlioglu et al. find a 99% decrease in sites using cryptomining among sites that had made use of it before. The low numbers of cryptominers found by our analysis confirms this trend and shows its declining influence on the WebAssembly ecosystem.

Insight 9. We find WebAssembly-based cryptominers to have significantly dropped in importance compared to the results of an earlier study [Musch et al. 2019a]. This finding motivates security research to shift the focus from malicious WebAssembly to vulnerabilities in WebAssembly binaries.

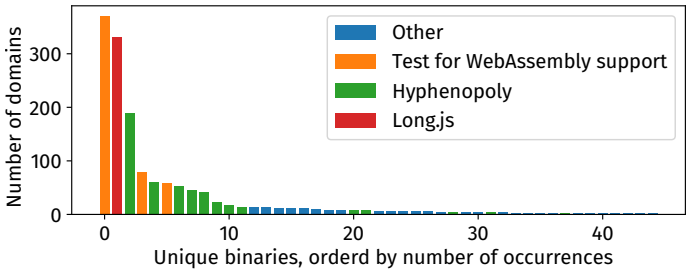


Figure 4.7: Binaries found on multiple websites.

4.3.6 *RQ4: Use Cases on the Web*

Given the decreased prevalence of cryptominers, what other use cases does WebAssembly have? The following focuses on the Web because it is the most prominent target platform of WebAssembly and because websites are complete applications that we can manually analyze with reasonable effort. We address the question in two ways. First, we study binaries that occur across multiple websites, to understand libraries and other widely reused components (Section 4.3.6.1). Second, we inspect a random sample of 100 unique binaries, to understand the application domains of WebAssembly (Section 4.3.6.2). To capture the full picture of use cases, the results are on unfiltered binaries.

4.3.6.1 *Binaries Found on Multiple Websites*

Out of all 476 unique binaries found on the Web, 70 are reused across at least two different top-level domains. Figure 4.7 shows a histogram of how often binaries occur on multiple domains. The data follows a long-tail distribution, i.e., a few binaries occur on many websites, while many other binaries recur a few times or only once. The top-most widely distributed binary occurs 371 times, i.e., in 28% of all domains where we detect WebAssembly binaries.

To better understand the most recurring binaries, we analyze them through a combination of automated clustering and manual inspection. The automated clustering represents each binary as a set of byte n-grams [Manning et al. 2008], summarizes the number of n-gram occurrences in a binary into a characteristic vector, and then clusters binaries based on the pairwise cosine similarity of their vectors. We then

inspect the top-most binaries in Figure 4.7, using the clusters to quickly identify variants of the same binary. Our analysis shows the following to be the most widely occurring WebAssembly binaries.

TESTING FOR WEBASSEMBLY SUPPORT At least 509 domains (38.5% of all domains that use WebAssembly) are serving a WebAssembly binary that tests whether the browser supports WebAssembly at all. We found two variants of such binaries, both of which are rather small: a six instruction binary with a single function called `test` and an eight byte binary that only contains the WebAssembly magic number followed by the language version. Websites serving these test binaries often also serve larger binaries, i.e., they first test whether WebAssembly is supported, and if it is, load a more complex binary. For example, at least 397 domains that serve a test binary also serve the Hyphenopoly library discussed next.

HYPHENOPOLY At least 462 domains (34.9% of all domains that use WebAssembly) serve binaries that are part of the *Hyphenopoly.js* JavaScript library, which uses WebAssembly to implement its core functionality. Hyphenopoly is a polyfill that "hyphenates text if the user agent does not support CSS-hyphenation".[22] Our clustering identifies 24 variants of this binary, which are all generated from the same underlying library to support different natural languages.

64-BIT INTEGER ARITHMETIC IN LONG.JS At least 331 domains (25% of all domains that use WebAssembly) serve a binary that is part of *long.js*, a JavaScript library for 64-bit integer computations.[23] The library is commonly used in video players.

DRACO LIBRARY FOR 3D DATA COMPRESSION At least 25 domains (1.8% of all domains that use WebAssembly) serve a binary that belongs to the *Draco* library, which support compressing and decompressing 3D data.[24] These binaries commonly occur on websites with 3D demos or integrated 3D assets.

Insight 10. The most widely occurring binaries on the Web are dynamic tests for WebAssembly support and JavaScript-WebAssembly libraries that perform computation-heavy tasks.

22 https://github.com/mnater/Hyphenopoly
23 https://github.com/dcodeIO/long.js
24 https://github.com/google/draco

Table 4.4: Application domains of 100 randomly sampled, unique WebAssembly binaries found on the Web.

Application Domain	# Binaries
Games	25
Text processing	11
Visualization / Animation	11
Media processing / Player	9
Demo, e.g., of a programming language	7
WebAssembly tutorial or test	5
Chat	3
Online gambling	2
Barcodes and QR codes	2
Room planning / Furniture	2
Blogging	2
Cryptocurrency wallet	2
Regular expressions	1
Hashing	1
PDF viewer	1

4.3.6.2 *Manual Inspection of a Random Sample*

To better understand the long-tail of binaries found on a few or only a single website, we also inspect a random sample of 100 unique binaries found on the Web. We exclude binaries detected only via the WebAssembly top lists (Section 4.2.3.2) to avoid biasing the results toward pre-selected application domains. By inspecting the binaries, the corresponding websites, and how the websites uses the binaries, we identify the purpose of 84 out of the 100 binaries. Table 4.4 summarizes the application domains that the binaries are used in. The most common domains are games, accounting for a quarter of all binaries. Text processing and applications in visualization and animation are also relatively common, with 11 of 100 binaries each. The remaining list shows the diversity of application domains WebAssembly is used in, ranging from online demos of programming languages, over support for creating and scanning barcodes, to document viewers.

Insight 11. The application domains of WebAssembly binaries on the Web reflect the diversity of the Web itself, showing that WebAssembly is used in a wide range of applications.

4.3.7 *RQ5: Minification and Names*

As a binary format with only a low-level textual representation, WebAssembly binaries cannot be as easily inspected and understood as source

code, e.g., in JavaScript. The ability to understand a WebAssembly binary is relevant for auditing third-party code and reverse engineering malware. For example, when a frequently depended-upon npm package contains a WebAssembly binary, the package distribution platform may want to check that it does not perform malicious actions, such as stealing cryptocurrency.[25] Because meaningful names, e.g., for functions, are helpful for understanding code [Lawrie et al. 2006], especially in binaries, we study to what extent WebAssembly binaries provide meaningful names.

ANALYSIS We perform a static analysis to assess two name-related characteristics of binaries. First, the analysis checks whether a binary contains a *name* section. While by default binaries contain names only for imported and exported program elements, the optional *name* section maps *all* function indices to identifiers. Second, the analysis checks whether the names of imported and exported names are minified. Both to save space and to obfuscate the code, compilers may shorten names down to single or two-letter names devoid of information. The analysis considers a WebAssembly binary as minified if it contains more than ten import or export names (to exclude small, potentially handwritten modules), where the average length of those names is ≤ 4 characters (to account for some imports that are never minified by Emscripten, thus increasing the average).

RESULTS Our results show an interesting difference between the full dataset and binaries found on the Web. In the full data set, many binaries contain a *name* section (19.6%) but only 4.1% of the binaries are minified. In contrast, among all binaries found on the Web, only 13.3% contain a *name* section but 28.8% are minified. These results show that for a significant fraction of websites, not only minified JavaScript code [Skolka et al. 2019], but also minified WebAssembly binaries make it harder to understand what code is running on the client side.

Insight 12. Many WebAssembly binaries on the Web (28.8%) are minified and do not contain useful names. To help security analysts understand third-party code, work on decompiling and reverse engineering WebAssembly is needed. Our work in Chapter 7 is a first step in that direction.

25 https://blog.npmjs.org/post/185397814280/plot-to-steal-cryptocurrency-foiled-by-the-npm

4.4 SUMMARY

This chapter presents a comprehensive empirical study of security properties, languages, and use cases of a diverse set of real-world WebAssembly binaries. After gathering binaries from several sources, ranging from source code repositories over packages managers to live websites, we analyze them through a combination of static code analysis, manual inspection, and statistical analysis. Our study shows that WebAssembly has grown into a diverse ecosystem with new challenges and opportunities for security researchers and practitioners, e.g., in analyzing vulnerabilities in WebAssembly binaries, in hardening binaries against exploitation, and in helping security analysts reverse engineer binaries. We make the binaries underlying our study, which yields by far the largest benchmark of WebAssembly binaries to date, available to support future work.

5 WASABI: A DYNAMIC ANALYSIS FRAMEWORK

In the previous Chapters 2, 3, and 4, we have been concerned with understanding and studying the WebAssembly language and ecosystem to gather new insights, in particular with a focus on security. In the following, we aim to also provide concrete and constructive help for developers. This chapter is the first of three (Chapter 5, 6, and 7) to detail practical tools we built for analyzing WebAssembly binaries. As discussed in the introduction, program analysis techniques are vital to ensure the reliability, security, and performance of applications. However, manually building such tools from scratch requires knowledge of low-level details of the language and its runtime environment.

This chapter presents WASABI, the first general-purpose framework for dynamically analyzing WebAssembly. It abstracts over low-level details and automates the tedious parts of dynamic analysis. WASABI provides an easy-to-use, high-level API that allows to observe all WebAssembly instructions with their inputs and outputs. It is based on binary instrumentation, which inserts calls to analysis functions written in JavaScript into a WebAssembly binary. Dynamically analyzing WebAssembly comes with several unique challenges, such as the problem of tracing type-polymorphic instructions with analysis functions that have a fixed type, which we address through on-demand monomorphization. We evaluate WASABI on compute-intensive benchmarks and real-world web applications and show that it (i) faithfully preserves the original program behavior, (ii) imposes an overhead that is reasonable for heavyweight dynamic analysis, and (iii) makes it straightforward to implement various dynamic analyses, including instruction counting, call graph extraction, memory access tracing, and taint analysis.

This chapter shares large parts of its material with the corresponding publication [Lehmann and Pradel 2019]. The author of this dissertation is also the main author of that paper and did all of the implementation, evaluation, and the majority of the writing.

5.1 MOTIVATION AND CONTRIBUTIONS

As we introduce in Section 1.2, dynamic analysis tools observe programs at runtime to help developers with improving the reliability, security, and performance of their applications. Unsurprisingly, dynamic analysis tools have a long history of success for languages other than WebAssembly, with household names such as *Memcheck* and *Valgrind* [Seward and Nethercote 2005] for native programs, coverage tools like *gcov* in GCC or *JaCoCo* for Java,[1] or *TaintDroid* [Enck et al. 2014] for security and privacy monitoring of Android applications. The need for dynamic analysis is particularly strong for highly dynamic languages, such as JavaScript [Andreasen et al. 2017], and for languages with a lot of low-level control, such as C and C++. As a compilation target of systems languages (Chapter 4) and with JavaScript as the language in the common browser host environment, WebAssembly programs sit exactly at the intersection of these two spheres, making them a prime target for dynamic analysis.

Authors of a dynamic analysis can usually choose between several implementation options. One option is to implement an individual analysis from scratch, e.g., by manually adding code to the program. This is both tedious and error-prone. If done at the binary level, it also requires an in-depth understanding of the instruction set and tools to manipulate it. A second option is to modify the runtime environment of the program, e.g., a virtual machine. Again, such modifications require detailed knowledge of the virtual machine implementation, and they also tie the analysis to a particular runtime or often even a specific version of it. Since WebAssembly serves as a compilation target of other languages, source-level instrumentation of these languages might appear to be another possible option. However, web applications often rely on third-party code, for which source code is unavailable at the client-side.

Instead of implementing a dynamic analysis from scratch, or modifying a runtime environment, another robust option is to build upon general-purpose dynamic analysis frameworks. Table 5.1 lists some existing popular frameworks: *Pin* [Luk et al. 2005] and *Valgrind* [Nethercote and Seward 2007] for native programs, *DiSL* [Marek et al. 2012] and *RoadRunner* [Flanagan and Freund 2010] for JVM byte code, and *Jalangi* [Sen et al. 2013] for analyzing JavaScript programs. Building

1 https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, https://www.jacoco.org

Table 5.1: Overview of existing dynamic analysis frameworks for other platforms and comparison with WASABI.

Framework	(Primary) Platform	Instruments…	Analysis Language	API Style
Pin	x86-64	native binaries	C/C++	instrumentation or callbacks/hooks
Valgrind	x86-64	native binaries	C	low-level instrumentation
DiSL	JVM	bytecode	Java	aspect-oriented
RoadRunner	JVM	bytecode	Java	event stream
Jalangi	JavaScript	source code	JavaScript	callbacks/hooks
WASABI	WebAssembly	bytecode	JavaScript	callbacks/hooks

on an existing framework reduces the overall effort required to build an analysis and enables the analysis author to focus on the design of the analysis itself. Unfortunately, there currently is no general-purpose dynamic analysis framework for WebAssembly.

This chapter presents WASABI, the first general-purpose framework for dynamic analysis of WebAssembly.[2] WASABI provides an easy-to-use, high-level API to implement heavyweight analyses that can monitor all low-level behavior. The framework is based on binary instrumentation, which inserts WebAssembly code that calls into analysis functions in between the program's original instructions. The analyses themselves are written in JavaScript and implement analysis functions, called *hooks*, to perform arbitrary operations whenever a particular instruction is executed. To limit the overhead that a dynamic analysis imposes, WASABI supports *selective instrumentation*, i.e., it instruments only those instructions that are relevant for a particular analysis.

As a simple example of a WASABI-based analysis, Listing 5.1 shows our re-implementation of the profiling part of a cryptomining detector [W. Wang et al. 2018]. Unauthorized use of computing resources is detected by monitoring the WebAssembly program and gathering an instruction signature that is unique for typical mining algorithms. Implementing the dynamic part of this analysis in WASABI takes ten lines

2 "Wasabi" stands for WebAssembly dynamic analysis using binary instrumentation.

```
const signature = {};
Wasabi.analysis.binary = function(loc, op) {
  switch (op) {
    case "i32.add":
    case "i32.and":
    case "i32.shl":
    case "i32.shr_u":
    case "i32.xor":
      signature[op] = (signature[op] || 0) + 1;
}};
```

Listing 5.1: An example WASABI analysis for detecting cryptominers through instruction profiling, implemented after a description in prior work [W. Wang et al. 2018].

of JavaScript, which use the framework's `binary` hook to keep track of all executed binary operations. In contrast, the original implementation is based on a special-purpose instrumentation of WebAssembly that W. Wang et al. [2018] implemented from scratch. This and more sophisticated analyses (Section 5.6.2) show that WASABI allows implementing analyses with little effort.

Apart from being the first dynamic analysis framework for WebAssembly, WASABI addresses several unique technical challenges. First, to provide a high-level API for tracking low-level behavior, the approach abstracts away various details of the WebAssembly instruction set. For example, WASABI bundles groups of related instructions into a single analysis hook, resolves relative target labels of branch instructions into absolute instruction locations, and resolves indirect call targets to actual functions. Second, WASABI transparently handles the interaction of the WebAssembly code to analyze and the JavaScript code that implements the analysis. A particular challenge is that WebAssembly functions must statically declare fixed parameter types, while some WebAssembly instructions are polymorphic, i.e., they can be executed with different numbers and types of arguments. To insert hook calls for polymorphic instructions, a different monomorphic variant of the hook must be generated for every concrete combination of argument types. WASABI uses *on-demand monomorphization* to automatically create such monomorphic hooks, but only for those type variants that are actually present in the given WebAssembly code. Third, WASABI faithfully executes the original program and even preserves

its memory behavior, which is useful to implement memory profilers. To this end, none of the inserted instructions requires access or modification of the program's original memory. Instead, analyses can track memory operations in JavaScript, i.e., in a separate heap that does not interfere with the WebAssembly heap.

For our evaluation, we implement eight analyses on top of WASABI, including basic block profiling, memory access tracing, call graph analysis, and taint analysis. We find that writing a new analysis is straightforward and typically takes a few dozens of lines of code. As expected by design, WASABI faithfully preserves the original program behavior. The framework instruments large binaries quickly (e.g., 40 MB in about 15 seconds). The increase in binary size and the runtime overhead imposed by WASABI depend greatly on the program and which instructions in it shall be analyzed. For most instructions, thanks to selective instrumentation, code size increases by less than 1%, but in the worst case, when every single instruction in PolyBench/C is instrumented, code size increases by 742%. The runtime overhead can be below 1.02x for analyzing instructions such as `drop` and `select`, 2.8x for analyzing `call`s, and up to 163x when analyzing every single instruction of `heat-3d` from the PolyBench/C benchmark suite. These results are in line with existing frameworks for heavyweight dynamic analysis.

CONTRIBUTIONS In summary, the work in this chapter has the following contributions:

- We present the first general-purpose framework for dynamically analyzing WebAssembly code, an instruction format that is becoming a cornerstone of future web applications.
- We present techniques to address unique technical challenges not present in existing dynamic analysis frameworks, including static resolution of relative branch targets and on-demand monomorphization of analysis hooks.
- We show that WASABI is useful as the basis for a diverse set of analyses, that implementing an analysis takes very little effort, and that the framework imposes an overhead that is reasonable for heavyweight dynamic analysis.
- We make WASABI available as open-source, enabling others to build on it: `http://wasabi.software-lab.org`.

5.2 OVERVIEW

This section gives on overview of the design of WASABI and the decisions that have led to it. Figure 5.1 shows the main components and their relation. The inputs to our framework, which are provided by an analysis author, are a WebAssembly binary to analyze (top-left) and a dynamic analysis written in JavaScript (bottom-left). Available source code for the program to analyze is not required.

The rationale for choosing JavaScript as the analysis language is threefold. First, it is widely used on the Web and hence well-known to web developers and other potential users of WASABI. Second, JavaScript is a high-level language, which makes it a convenient choice for writing dynamic analyses. Third, browsers on the client side and Node.js on the server side are two common host environments for WebAssembly. Both use JavaScript as their programming language. A consequence of this choice is that WASABI is currently limited to host environments with JavaScript support. An interesting direction for future work is to also compile the dynamic analysis itself to WebAssembly and merge it with the program code, which would resolve this limitation.

WASABI operates in two phases: First, *static instrumentation* of the binary (top half of Figure 5.1), and second, the actual *dynamic analysis* of the application (bottom-half). The first phase augments the given WebAssembly binary with instructions that call into the analysis implementation. To this end, the instructions of the original program are interleaved with calls to *low-level analysis hooks* (middle gray box). Those low-level hooks are generated by WASABI also during the static instrumentation phase. They are implemented in JavaScript. Additionally, WASABI statically extracts information from the program which is used in the second phase. In the second phase at runtime, the low-level hooks then call *high-level analysis hooks*, which the analysis author implements. The WASABI runtime provides further information, e.g., the types of functions in the program, to the analysis.

Ahead-of-time binary instrumentation offers three important advantages over alternative designs. First, it is independent of the execution platform that WebAssembly runs on and robust to changes in current platforms. Suppose we would instead modify a specific runtime implementation, e.g., the WebAssembly engine in Firefox, then WASABI could not analyze programs executed elsewhere. Moreover, WASABI would risk becoming outdated when the execution platform evolves.

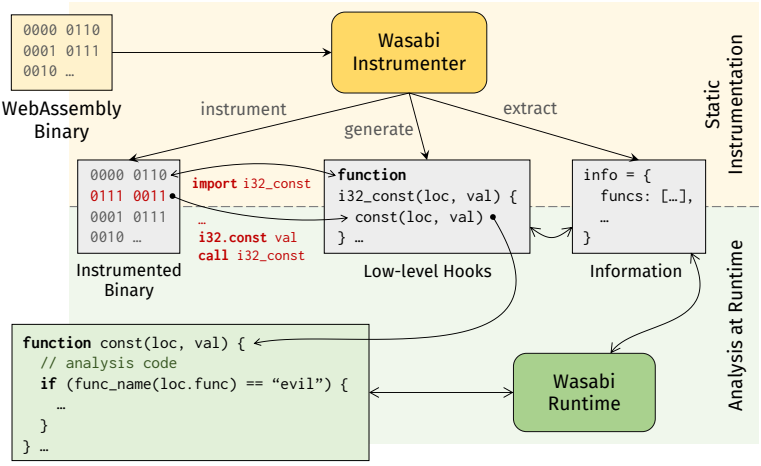


Figure 5.1: Overview of WASABI, its instrumentation and analysis phases, and the inputs and outputs.

Second, binary instrumentation enables WASABI to support binaries produced by a variety of compilers and makes it independent of the source language. There are several source languages compiling to WebAssembly, e.g., C and C++,[3] Rust,[4] Go,[5] and TypeScript.[6] Source-level instrumentation for all these languages would not scale, and the source code of WebAssembly running on websites is often unavailable. So this is not an option if we want to support, e.g., security applications like reverse engineering. Third, ahead-of-time instrumentation avoids runtime overhead compared to instrumenting code during the execution [Bruening et al. 2003; Luk et al. 2005; Nethercote and Seward 2007]. Since WebAssembly, in contrast to other binary formats, does not suffer from language features that make ahead-of-time instrumentation inherently difficult, such as self-modifying code or mixed code and data (Chapter 2), WASABI can reliably instrument WebAssembly binaries ahead-of-time.

3 https://emscripten.org, [Zakai 2011]
4 https://www.rust-lang.org/what/wasm, https://github.com/rustwasm
5 https://github.com/golang/go/wiki/WebAssembly, [Musiol 2018], https://tinygo.org/docs/guides/webassembly/
6 https://www.typescriptlang.org, [Reiser and Bläser 2017]

5.3 ANALYSIS API

WASABI offers analysis authors a high-level analysis API, consisting of 23 hooks to be implemented by the analysis. The API is both powerful enough to enable arbitrary dynamic analyses and high-level enough to spare the analysis author dealing with irrelevant details. Table 5.2 shows the hooks,[7] along with their arguments and the types of the arguments. The hooks can be roughly clustered into six groups: Numerical operations (`const`, `unary`, `binary`) instructions related to variables and stack manipulation (`local`, `global`, `drop`, `select`), memory accesses and management instructions (`load`, `store`, `memory.size`, `memory.grow`), function calls (`call_pre`, `call_post`, `return`), unconditional and conditional branches (`br`, `br_if`, `br_table`), and blocks (`begin`, `end`). When invoked, each hook receives details about the respective instruction, e.g., a string representation of the operation (`op`), its inputs and outputs, and the code location (first row of the table).

The API is designed to ensure four important properties:

- *Full instruction coverage.* It covers all WebAssembly instructions and provides all their inputs and results to the analysis. This property is crucial to implement arbitrary dynamic analyses that can observe all runtime behavior. We describe in Section 5.4 how selective instrumentation limits the costs to be paid for this flexibility.
- *Grouping of instructions.* The API groups related WebAssembly instructions into a single hook, which significantly reduces the number of hooks analyses must implement. Providing one hook per instruction to the analysis would require a huge number of hooks (e.g., there are 123 numeric instructions alone), whereas WASABI's API provides 23 hooks only. To distinguish between instructions, if necessary, the hooks receive detailed information as arguments. For example, the `binary` hook receives an `op` argument that specifies which binary operation is executed. To hide the various variants of polymorphic instructions from analyses authors, WASABI also maps all variants of the same kind of instruction into a single hook. For example, the `call` instruction can take different numbers and types of arguments, depending on the called function, which are represented in the hook as an array of varying length.

7 For brevity, the table leaves out four additional hooks that WASABI supports – `start`, `nop`, `unreachable`, and `if` – for a total of 23 hooks.

Table 5.2: API of the high-level analysis hooks.

Hook Name	Hook Arguments and Their Types
Every hook	`location: {func: number, instr: number}`, *other arguments…*
Simple, numerical operations:	
`const`	`op, value:` $type_{val}$
`unary`	`op, input:` $type_{val}$, `result:` $type_{val}$
`binary`	`op, first:` $type_{val}$, `second:` $type_{val}$, `result:` $type_{val}$
where	`op: string` of the instruction, e.g., `"i32.add"` or `"local.get"`
Variables and stack management:	
`local`/**`global`**	`op, index: number, value:` $type_{val}$
`drop`	`value:` $type_{val}$
`select`	`condition: boolean, first:` $type_{val}$, `second:` $type_{val}$
Memory instructions:	
`load`/**`store`**	`op, memarg, value:` $type_{val}$
where	`memarg: {addr: number, offset: number}`
`memory.size`	`currentSizePages: number`
`memory.grow`	`byPages: number, previousSizePages: number`
Function calls and returns:	
`call_pre`	`func: number, args: [`$type_{val}$`], tableIndex: (number | null)`
where	`tableIndex == null` iff it was a direct call
`call_post`	`results: [`$type_{val}$`]`
`return`	`results: [`$type_{val}$`]`
Branches:	
`br`	`target`
`br_if`	`target, condition: boolean`
`br_table`	`table: [target], tableIndex: number`
where	`target: {label: number, location: location}`
Block entry and exit hooks:	
`begin`	`type`
`end`	`type, begin: location, if: (location | null)`
where	`type: ("function" | "block" | "loop" | "if" | "else")`, `if != null` iff the `end` terminates an `else` block

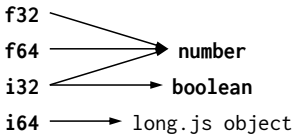


Figure 5.2: Mapping of WebAssembly types to JavaScript.

- *Pre-computed information.* WASABI provides pre-computed information along with the runtime values to some hooks because the values from WebAssembly execution on their own are not informative enough for an analysis. For example, a relative branch label would not be helpful to a dynamic analysis without additional context of the surrounding blocks (cf. Section 2.2). For this reason, the three branch-related hooks receive `target` objects that contain the statically resolved, absolute location of the next instruction that will be executed if the branch is taken (`target.location`), alongside the low-level relative branch label (`target.label`). Similarly, for indirect calls, WASABI resolves the table index to the actually called function.
- *Faithful type mappings.* Finally, the API faithfully maps typed values from WebAssembly to JavaScript. Figure 5.2 shows the four primitive types in WebAssembly and how they are represented without loss of precision in a WASABI analysis. `i32`, `f32`, and `f64` are represented with the JavaScript `number` type. However, because JavaScript `numbers` are floating-point values, they can only represent integers up to $2^{53} - 1$ without loss of precision. To pass 64-bit numbers to JavaScript, WASABI thus transparently maps them to `long.js`[8] objects. Finally, conditionals, which are `i32`s with value 0 or 1 in WebAssembly, are mapped to JavaScript `booleans`.

The API gives analysis authors the power to implement sophisticated dynamic analyses with little effort. In particular, it is straightforward to implement memory shadowing [Nethercote and Seward 2007], a feature useful, e.g., for tracking the origin of undesired values or for taint analysis. To associate some meta-information with a memory value, all an analysis must do is to maintain a map of memory locations to meta-information, and to update the meta-information on memory-related instructions. One of our example analyses (Section 5.6.2) is a taint analysis that implements memory shadowing in this way.

8 https://github.com/dcodeIO/long.js

5.4 STATIC BINARY INSTRUMENTATION

The following presents the core of WASABI: its static binary instrumentation component, which inserts code that eventually calls the high-level analysis hooks described in the previous section. We first describe the instrumentation of individual, simple WebAssembly instructions (Section 5.4.1) and how WASABI reduces overhead via selective instrumentation (Section 5.4.2). Then, we highlight four instrumentation challenges that are unique to WebAssembly and describe how WASABI addresses them (Sections 5.4.3 to 5.4.6).

5.4.1 *Instrumentation of Instructions*

To allow an analysis to trace every instruction that is executed, WASABI inserts calls to analysis hooks for each instruction. Table 5.3 illustrates the instrumentation for a subset of all instructions. Row 1 shows the simplest case: a `const` instruction that pushes an immediate value on the stack. The instrumentation adds a call of the corresponding hook. Since the hook also receives the value produced by the `const` instruction as an argument, the value is pushed once more on the stack prior to the call. After the call to the hook, the stack will be the same as in the original, uninstrumented program.

Row 2 of Table 5.3 shows an instruction that takes inputs and produces results. To pass both to the inserted hook call, we need to duplicate values on the stack. For this purpose, WASABI generates a fresh local of the appropriate type and writes the current stack top to this local with `local.tee`. Before the hook call, the inserted code retrieves the stored input and its result with `local.get`. Row 4 illustrates how WASABI instruments `call` instructions. In contrast to other instructions, we surround the original `call` instruction with two hook calls, so that an analysis author can execute analysis behavior before and after the function call.

All inserted calls go to JavaScript functions that are imported into the WebAssembly binary. These imported functions are not yet the high-level hooks from Section 5.3, but low-level hooks that are automatically generated by WASABI. There are several reasons for this indirection. First, it allows WASABI to map typed WebAssembly instructions to untyped JavaScript hooks in a seamless way (Section 5.4.3). Second, it helps providing information that is useful in high-level hooks but not

Table 5.3: Instrumentation of select WebAssembly instructions. Every hook also receives two `i32` arguments that encode the original instruction's location. For brevity, we omit the corresponding `i32.const` instructions preceding the hook call here.

#	Instructions: Original ⇒ Instrumented		Explanations and other changes made to the module
	Instructions that only produce a value, here a simple constant instruction:		
1	**`i32.const`** *value*	**`i32.const`** *value*	Original instruction
		`i32.const` *value*	Duplicate the value, to pass as an argument for the hook
		`call` $idx_{\texttt{hooks.i32.const}}$	Call low-level analysis hook, added as an imported function to module
	General instructions that take inputs (e.g., unary, binary, load, store):		
2		**`local.tee`** $idx_{\textit{temp input local}}$	Save instruction input(s) into freshly generated local(s)
	`f32.abs`	**`f32.abs`**	Original instruction
		`local.tee` $idx_{\textit{temp result local}}$	Save instruction result into freshly generated local
		`local.get` $idx_{\textit{temp input local}}$	} Push instruction input(s) and result as hook arguments on the stack
		`local.get` $idx_{\textit{temp result local}}$	
		`call` $idx_{\texttt{hooks.f32.abs}}$	Call low-level analysis hook, stack is now as it was after the instruction
	Type-polymorphic `drop` and `select` instructions:		
3	...	...	Type check all instructions to keep track of abstract stack
	(preceding code)	(preceding code)	Here: assume at the top of the stack is a value of type $type_{val}$
	...	...	Thus, the following `drop` has type $[type_{val}] \rightarrow []$
	`drop`	**`call`** $idx_{\texttt{hooks.drop\_}type_{val}}$	Monomorphized hook (here: consumes the value in place of `drop`)

#	Instructions: Original ⇒ Instrumented	Explanations and other changes made to the module
4	Calls (`call` and `call_indirect`) and similarly also `return`s:	
	`call` idx_{func}	`local.tee` $idx_{\text{temp input local}}$ — Save function argument(s) in freshly generated local(s)
		`i32.const` idx_{func} — Pass index of called function to hook
		`local.get` $idx_{\text{temp input local}}$ } Pass arguments to monomorphized `call_pre` hook
		`call` $idx_{\texttt{hooks.call\_pre\_}(type_{local})^*}$
		`call` idx_{func} — Original function call
		`local.tee` $idx_{\text{temp result local}}$ — Save call result(s) in freshly generated local(s)
		`local.get` $idx_{\text{temp input local}}$ } Pass results to monomorphized `call_post` hook
		`call` $idx_{\texttt{hooks.call\_post\_}(type_{local})^*}$
5	Instructions with `i64` inputs or results (here: `i64` constant); the `i64` values are split into two `i32`s for passing to the hook:	
	`i64.const` *value*	`i64.const` *value* — If instruction has side-effects, the values are duplicated via a local instead
		`i64.const` *value* } Push lower 32 bits of the value as an `i32` onto the stack
		`i32.wrap/i64`
		`i64.const` *value* } Shift upper 32 bits to the right, then push as `i32` onto the stack
		`i64.const` 32
		`i64.shr_s`
		`i32.wrap/i64`
		`call` $idx_{\texttt{hooks.i64.const}}$ — Call hook with pair of (`i32`, `i32`) instead of the original `i64` value

#	Instructions: Original	⇒	Instructions: Instrumented	Explanations and other changes made to the module
	Blocks/structured control-flow (`block`, `loop`, `if`, `else`, `end`) and branches (`br`, `br_if`, `br_table`):			
	label: **block**		*label*: **block**	*label* is implicit and not encoded in the WebAssembly binary
			call $idx_{\texttt{hooks.begin\_block}}$	Every block type (`block`/`loop`/`if`/`else`) has its own low-level `begin` hook
	otherlabel: **loop**		*otherlabel*: **loop**	Nested block (here: a `loop`)
			call $idx_{\texttt{hooks.begin\_loop}}$	The loop's `begin` hook is called once per iteration
	…		…	Block body (instrumented recursively)
6			**i32.const** 1 (*label*)	"Raw" (unresolved, relative) target label is passed to hook as an integer
			i32.const resolve(*label*)	Resolve label during instrumentation to absolute location, also pass to hook
			call $idx_{\texttt{hooks.br}}$	Branch hook is invoked before taking the branch
			call $idx_{\texttt{hooks.end\_loop}}$	} Call `end` hooks of blocks "traversed" during the branch
			call $idx_{\texttt{hooks.end\_block}}$	} Call `end` hooks of blocks "traversed" during the branch
	br 1 (*label*)		**br** 1 (*label*)	
	…		…	(Not shown:) `end` hooks receive location of matching block begin
			call $idx_{\texttt{hooks.end\_loop}}$	Every block type (`block`/`loop`/`if`/`else`) has its own `end` hook (cf. `begin_*`)
	end		**end**	
			call $idx_{\texttt{hooks.end\_block}}$	
	end		**end**	

available at the current instruction (Section 5.4.4). Third, Section 5.4.5 shows that WASABI sometimes also calls other hooks at runtime, because the necessary information which hooks to call is available only then. Finally, the low-level hooks can convert values before passing them to the high-level hooks (Section 5.4.6). All of these issues can be solved by automatically generated low-level hooks that are hidden from analysis authors.

5.4.2 *Selective Instrumentation*

Not every analysis uses all of the hooks provided by the API from Section 5.3. To reduce both the code size and the runtime overhead of the instrumented binary, WASABI supports *selective instrumentation*. That is, only those kinds of instructions are instrumented that have a matching high-level hook in a given analysis. WASABI ensures that the instrumentation for different kinds of instructions are independent of each other, so that instrumenting only some instructions still correctly reflects their behavior. Sections 5.6.5 and 5.6.6 show that selective instrumentation significantly reduces code size and runtime overhead.

5.4.3 *On-Demand Monomorphization*

An interesting challenge for the instrumentation comes from static typing in WebAssembly. While there are polymorphic instructions, WebAssembly functions, including our hooks, must always be declared with a fixed, monomorphic type. For polymorphic instructions, WASABI cannot simply generate one hook per kind of instruction: Consider `drop` with the polymorphic instruction type $[\tau] \rightarrow [\,]$. (That is, it pops an argument of any type τ from the stack and pushes no value.) Inserting a call to the *same* hook function after each `drop` is not possible, because the hook's function type would then be polymorphic. Instead, WASABI generates multiple monomorphic variants of a polymorphic hook and inserts a call to the appropriate monomorphic low-level hook.[9]

For many polymorphic instructions, determining which monomorphic hook variant to call is straightforward. For example, the instruction type of `global.set` depends only on the type of the referenced variable. The types of `drop` and `select`, however, cannot be simply looked

9 This strategy is similar to the compilation of generic functions in Rust or instantiation of function templates in C++ [Klabnik and Nichols 2018; Vandevoorde and Josuttis 2002].

up. Instead, as shown in row 3 of Table 5.3, their type depends on all preceding instructions. WASABI thus performs full type checking during instrumentation, that is, it keeps track of the types of all values on the stack [Haas et al. 2017; Watt 2018]. When the `drop` in the last line of the example is encountered, its input type is equal to the top of the abstract stack and WASABI can insert the call to the matching monomorphic low-level hook.

While creating monomorphic variants of hooks yields type-correct WebAssembly code, doing so eagerly leads to an explosion of the required number of monomorphic hooks. Since functions can have an arbitrary number of arguments and results[10], the number of monomorphic hooks for `calls` and `returns` is even unbounded. One way to address this problem would be to set a heuristic limit, e.g., by generating hooks for calls with up to ten arguments. However, the resulting 4^{10} = 1,048,576 call-related hooks would cause unnecessary binary bloat and may still fail to support all calls.

Instead, WASABI generates monomorphic hooks on-demand only for instructions and type combinations that are actually present in the given binary. We call this approach *on-demand monomorphization* of hooks. During instrumentation, WASABI maintains a map of already generated low-level hooks. If a required hook, e.g., for a `call` instruction with type `[i32]` → `[f32]`, is present in the map, the function index of the hook is returned. Otherwise, WASABI generates the hook and updates the map. Our evaluation shows that on-demand monomorphization significantly reduces the number of low-level hooks, and hence the code size, compared to the eager approach described above.

5.4.4 *Resolving Branch Labels*

As described in Section 2.2, WebAssembly relies on structured control-flow, a feature not present in other low-level instruction sets. An interesting challenge that arises from structured control-flow is how and when to resolve the destination of branches. Row 6 of Table 5.3 illustrates the problem with a few control-flow-related instructions. The `br` instruction jumps to a destination referenced by a relative integer label 1. However, passing this label to the high-level dynamic analysis API would be of limited use, because without additional static infor-

10 Strictly speaking, functions in the binary format 1.0 have at most one result, but the formal semantics already support multiple return values [Haas et al. 2017].

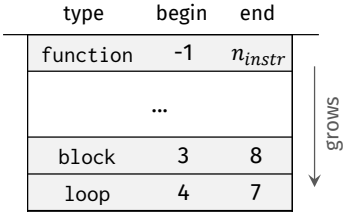


Figure 5.3: Abstract control stack at the `br` instruction in row 6 of Table 5.3 (assuming the `block` is preceded by three other instructions).

mation (namely the surrounding blocks), the dynamic analysis cannot resolve the label to a code location.

To enable analysis authors to reason about branch destinations without implementing their own static analysis, WASABI resolves branch labels during the instrumentation and passes the resulting absolute instruction locations to the high-level API. To resolve branch labels, WASABI keeps track of an *abstract control stack* while instrumenting WebAssembly code. Whenever the instrumentation enters a new block, an element is pushed to the control stack, consisting of the block type (`function`, `block`, `loop`, `if`, or `else`), the location of the block begin, and the location of the matching `end` instruction. Whenever the instrumentation encounters the end of a block, the topmost entry is popped of the control stack. As an example, Figure 5.3 shows the control stack for the code in row 6 of Table 5.3.

Given the abstract control stack, WASABI can determine during instrumentation what code location a branch, if taken, will lead to. At every branch to a label n, WASABI queries the control stack for its $n + 1$-th entry from the top, to determine the targeted block, and then computes the location of the next instruction from the block type and the locations of the `begin` and `end` instructions. This absolute instruction location is then given as an argument to the branch hook, as shown in the example in Table 5.3 and in the high-level API in Table 5.2.

5.4.5 *Dynamic Block Nesting*

Another control-flow-related challenge is about observing the end of the execution of a block. Some analyses may want to observe the block nesting at runtime, i.e., to perform some action when a block is entered

and left. For this purpose, WASABI offers the high-level `begin` and `end` hooks (Section 5.3). The example in row 6 of Table 5.3 shows that our instrumentation adds the respective hook calls (`call` $idx_{\texttt{hooks.begin\_block}}$ and `call` $idx_{\texttt{hooks.end\_block}}$) at the beginning of a block and before the matching `end`.

Unfortunately, `branching` or `returning` will jump out of a block and over the inserted `end` hook calls. Consider the last two hook calls of `hooks.end_loop` and `hooks.end_block` in Table 5.3. They are not executed because the earlier `br 1` directly transfers control to after the enclosing block. To account for that, WASABI adds additional calls before each `branch` and `return` that invoke every end hook of the blocks that will be "traversed" during the jump. That is, as the example shows, WASABI inserts calls to the `end` hooks for the two enclosing blocks prior to the `br 1` instruction. Again, the *control stack* can tell us which end hooks need to be called, namely all between the current block (stack top, inclusive) and the branch target block (also inclusive). For example, in Figure 5.3, the instrumented code calls the `loop` and `block` end hooks. For a `return` it would be all blocks on the block stack up to and including the `function` block.

For conditional branches (`br_if`), we call the `end` hooks for traversed blocks only if the branch is actually taken. Similarly, for multi-way branches (`br_table`), which branch is taken (and thus which blocks are left) is known only at runtime. Thus, the instrumentation statically extracts the list of ended blocks for every branch table entry and stores this information. Inside the low-level hook for `br_table`, one of the stored branch table entries will then be selected, before calling the corresponding `end` hooks at runtime.

5.4.6 *Handling* `i64` *Values*

As mentioned in Section 5.3, `i64` values cannot be represented precisely with the JavaScript `number` type, because it is a double precision floating-point number. To nevertheless enable dynamic analyses to faithfully observe all runtime values, including `i64` values, WASABI splits a 64-bit integer into two 32-bit integers to pass them to JavaScript. For every `i64` stack value (either produced by a `const` or by any other instruction), we thus insert instrumentation as shown in row 5 of Table 5.3. The inserted code duplicates the `i64` value twice. From the first copy only the lower 32 bits are extracted, and the second copy is

shifted to the right to result in the upper 32 bits. Both `i32` values can then be passed to the hook in question. On the JavaScript side, the low-level hook joins the two 32-bit values into a `long.js` object, enabling an analysis to faithfully reason about 64-bit integers.[11]

5.5 IMPLEMENTATION

We have implemented the static instrumentation component of WASABI in about 5000 lines of Rust code. This includes a parser for the WebAssembly binary format, data type definitions of an AST, a type checker, and the other static analyses WASABI performs during instrumentation, e.g., statically resolving relative branch labels to absolute instruction locations. Rust programs themselves can be compiled to WebAssembly, which gives us the option to run WASABI in the browser and instrument WebAssembly programs at load time in the future.

To reduce the time required for instrumenting large binaries (large WebAssembly binaries can have more than 100,000 functions), WASABI can parse and instrument multiple functions in parallel. For parallel parsing, we exploit that the WebAssembly binary format contains section sizes that allow to jump ahead to the next section (as described Section 2.1). Once parsed, individual functions can be instrumented almost fully independently of each other. The only synchronization point is the map of low-level hooks created during on-demand monomorphization (Section 5.4.3), which is guarded by an upgradeable multiple readers/single writer lock.

Our implementation is available for others to build on under the permissive MIT license at `http://wasabi.software-lab.org`. We have also reused the binary parser and AST components of WASABI ourselves in the FUZZM project, described in Chapter 6.

5.6 EVALUATION

To evaluate WASABI, we focus on five research questions:

RQ1 How easy is it to write dynamic analyses with WASABI?

11 An alternative would be to use the `BigInt` type in JavaScript (`https://github.com/tc39/proposal-bigint`). However, this was only standardized in late 2019 and became widely available in browsers in 2020, after we had already developed WASABI.

RQ2 Do the instrumented WebAssembly programs remain faithful to the original execution?

RQ3 How long does it take to instrument programs?

RQ4 How much does the code size increase?

RQ5 What is the runtime overhead due to instrumentation?

5.6.1 *Experimental Setup*

For the evaluation, we apply WASABI to 32 programs in total. 30 of them are from the PolyBench/C benchmark suite,[12] which has also been used in the evaluation of the paper introducing WebAssembly [Haas et al. 2017]. In total, the PolyBench benchmark suite comprises 5,163 non-empty, non-comment lines of C code. We compile the PolyBench programs to WebAssembly with Emscripten version 1.38.8, resulting in 790 KB of WebAssembly binaries. Moreover, we use two complex, real-world WebAssembly binaries without access to their source code: the Unreal Engine 4 Zen Garden demo,[13] as an example of a major game engine running in the browser, and the PSPDFKit benchmark,[14] which exercises a commercial library for in-browser rendering and annotation of PDFs. Their WebAssembly binaries are much larger than the PolyBench binaries, with sizes of 39.5 MB (Unreal demo) and 9.5 MB (PSPDFKit), respectively.

All experiments are performed on a laptop computer with an Intel Core i7-7500U CPU (2 cores, hyper-threading, 2.7 to 3.5 GHz, 4 MB L3 cache) and 16 GB of RAM. The operating system is Ubuntu 17.10 64-bit. To execute the WebAssembly programs, which are embedded into websites, we use a nightly version of Firefox 63.0a1 (2018-08-02).

5.6.2 *RQ1: Ease of Implementing Analyses*

We have implemented eight dynamic analyses on top of WASABI. Table 5.4 lists them, along with the hooks they implement and their total lines of JavaScript code.

12 http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

13 The original link has since been removed, but an alternative sourceh is https://www.unrealengine.com/marketplace/en-US/product/epic-zen-garden

14 https://pspdfkit.com/webassembly-benchmark/

Table 5.4: Dynamic analyses we built on top of WASABI.

Analysis	Used Analysis Hooks	LoC
Instruction mix analysis	all	42
Basic block profiling	`begin`	9
Instruction coverage	all	11
Branch coverage	`if`, `br_if`, `br_table`, `select`	14
Call graph analysis	`call_pre`	18
Dynamic taint analysis	all	208
Memory access tracing	`load`, `store`	11
Cryptominer detection	`binary`	10

INSTRUCTION MIX ANALYSIS This analysis counts how often each kind of instruction is executed, which can serve as a basis for performance and security analyses.

BASIC BLOCK PROFILING A classic dynamic analysis that counts how often each function, block, and loop is executed, which is useful, e.g., for finding "hot" code.

INSTRUCTION AND BRANCH COVERAGE These analyses record for each instruction and branch, respectively, whether it is executed, which is useful to assess the quality of tests.

CALL GRAPH ANALYSIS This analysis creates a dynamic call graph, including indirect calls and calls between functions that are neither imported nor exported. Call graphs are the basis of various other analyses, e.g., to find dynamically dead code or to reverse-engineer malware.

TAINT ANALYSIS The analysis associates a taint with every value and tracks how taints propagate through instructions, function calls, and memory accesses, to detect illegal flows from sources to sinks.

MEMORY ACCESS TRACING The analysis tracks all memory accesses and stores them for later off-line analysis, e.g., to detect cache-unfriendly access patterns.

CRYPTOMINER DETECTION As discussed in the introduction, this analysis gathers a signature based on the frequency of binary instructions to identify mining of cryptocurrencies [W. Wang et al. 2018].

```
// A sparse array from function index and instruction index
// to a list of taken branches/conditionals.
const coverage = [];
function addBranch({func, instr}, branch) {
    // Make sure the list of branches is initialized.
    coverage[func] = coverage[func] || [];
    coverage[func][instr] = coverage[func][instr] || [];
    // Add the branch to the list.
    if (!coverage[func][instr].includes(branch)) {
        coverage[func][instr].push(branch);
    }
}
// High-level hooks implementation.
Wasabi.analysis = {
  // These are all conditional branching instructions:
    if_(location, condition) { addBranch(location, condition) },
    br_if(location, target, condition) { addBranch(location, condition) },
    br_table(location, tbl, df, tableIdx) { addBranch(location, tableIdx) },
    select(location, condition) { addBranch(location, condition) },
};
```

Listing 5.2: Simple branch coverage analysis with WASABI.

As illustrated by the low numbers of lines of code in Table 5.4, each of these analyses can be implemented with little effort. For further illustration, Listing 5.2 shows the implementation of the branch coverage analysis. It implements four hooks, `if`, `br_if`, `br_table`, and `select` to keep track of all branches.

5.6.3 *RQ2: Faithfulness of Execution*

To validate that WASABI's instrumentation does not modify the semantics of the original program, we compare the behavior of each unmodified binary with the behavior of the fully instrumented binary. For the PolyBench programs, we compile each program with an option to output intermediate results of every calculation on the console. Similarly, the Unreal Engine demo has a mode to check that the pixel values of rendered frames are the same as pre-defined reference frames. For all these programs, the behavior remains unchanged after instrumentation. The PSPDFKit benchmark does not provide any built-in correctness check; based on our manual observations the behavior of the original and instrumented code appear to be the same.

Table 5.5: Time taken to instrument WebAssembly binaries, averaged across 20 runs (and across 30 programs for the PolyBench suite).

Program	Binary Size (B)	Runtime (ms)	$\frac{\text{MB}}{\text{s}}$
PolyBench (30 programs)	26 332 ± 299	23 ± 1.4	1.15
PSPDFKit	9 615 389	5 129 ± 65	1.87
Unreal Engine 4	39 510 398	15 481 ± 293	2.55

As another way to validate the instrumented WebAssembly binaries, we use the static WebAssembly validator, which checks that the instrumented binaries are well-formed and type-correct [WebAssembly Specification]. Running `wasm-validate` from the WebAssembly Binary Toolkit[15] on all 32 fully instrumented programs shows that all the instrumented code passes the validator. We also instrument and successfully validate Wasabi's output on all programs of the official WebAssembly specification test suite,[16] which consists of 63 additional programs on top of the 32 programs of our benchmark suite.

5.6.4 *RQ3: Time to Instrument*

Table 5.5 shows how long Wasabi takes to instrument the programs. The $x \pm y$ notation means a mean value of x and a standard deviation of y after 20 repetitions. For readability, we have summarized the results for all 30 PolyBench programs in one row. While the PolyBench programs are of similar, small size (26.3 KB ± 299 B), the PSPDFKit and Unreal Engine binaries are considerably larger (9.6 MB and 39.5 MB, respectively). Instrumentation takes 23ms, on average, for the PolyBench programs, i.e., it is almost instantaneous, and still quick for the larger PSPDFKit (5s) and Unreal Engine binaries (15.5s). Wasabi's instrumentation is parallelized (Section 5.5), and these numbers are obtained with four threads running on two physical cores. The single-threaded instrumentation time on the large Unreal Engine binary is on average 26.5s, showing that the parallelization reduces the execution time to $15.5/26.5 \approx 0.58$ of the single-threaded time. The last column of Table 5.5 reports the throughput, i.e., binary code processed per second, showing that the throughput increases with larger binary sizes.

15 https://github.com/WebAssembly/wabt
16 https://github.com/WebAssembly/spec/tree/master/test

5.6.5 *RQ4: Code Size Overhead*

Figure 5.4 presents the increase in binary code size after instrumenting a program. Since many analyses need only a small subset of all hooks (e.g., block profiling needs only `begin`), we evaluate code size increase per required hook, as provided by selective instrumentation (Section 5.4.2). For each hook on the x-axis, the figure shows on the y-axis the increase in binary size as a percentage of the original program size. That is, 0% means the instrumented binary has the same size as the original one and 100% means the program doubled in size due to instrumentation.

With selective instrumentation, more than half of the hooks increase the binary size only by a negligible amount or not at all (less than 1% increase for `nop`, `unreachable`, `memory.size`, `memory.grow`, `select`, and `br_table`; less than 10% for `drop`, `return`, `unary`, `global`, `if`, `br`, and `br_if`, on average). In fact, in several cases the Unreal Engine binary size decreased by 1% because WASABI encodes indices more compactly than the original binary. This is because of the LEB128 variable-length format used to encode integers in WebAssembly (see Section 2.1). It allows for multiple equivalent encodings of the same number with different lengths.

Naturally, hooks for instructions that appear very often in the program have the largest influence on the code size, e.g., memory `load` and `store` (between 39% and 58% increase), `begin` and `end` of blocks (11% – 84%), pushing to the stack with `const` (59 – 71%), operations on `locals` (128 – 180%), and finally `binary` instructions (83 – 190%). The difference for the `binary` hook between PolyBench and the other programs can be explained by the former being mostly numerical computation (thus having more binary instructions such as `i32.mul`), whereas PSPDFKit and the Unreal Engine have more diverse code. When instrumenting for all hooks together, which is not required for many analyses, the size increases between 495% (Unreal Engine 4) and 743% (mean across the 30 PolyBench/C programs). This result shows that selective instrumentation is very effective in reducing the binary size, compared to blindly instrumenting all instructions.

To evaluate WASABI's on-demand monomorphization of hooks, we count how many low-level hook functions are inserted during full instrumentation. For PolyBench, between 110 (`floyd-warshall` program) and 122 (`deriche`) hooks are inserted, 302 hooks for PSPDFKit, and

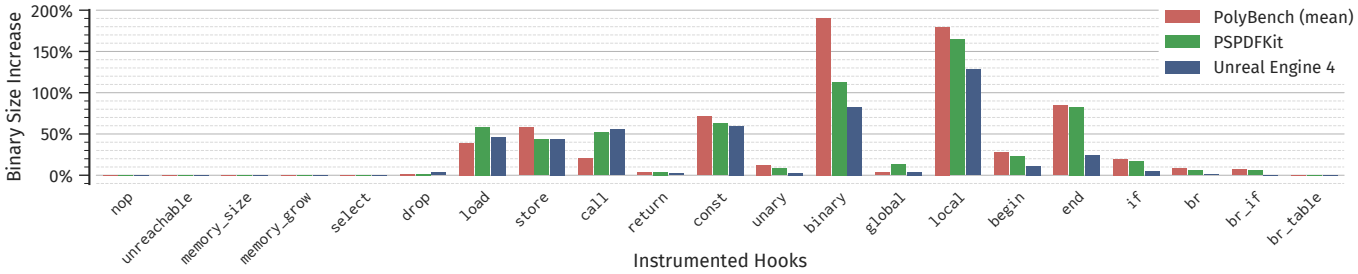


Figure 5.4: Binary size increase in percent of the original size, when instrumenting the test programs for different analysis hooks. For readability, binary sizes for the 30 PolyBench programs are shown averaged.

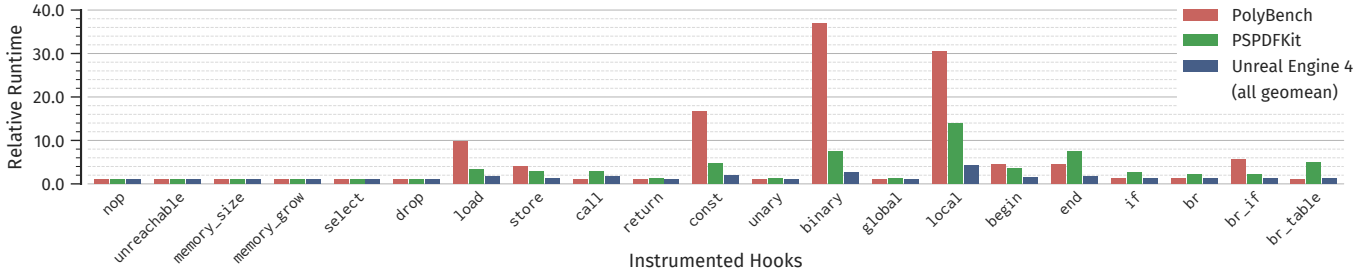


Figure 5.5: Runtime of the instrumented programs relative to the uninstrumented runtime, per analysis hook. Results are averaged over 20 runs (and again for readability, over the 30 PolyBench programs).

783 hooks for the Unreal Engine. In the original Unreal Engine binary, i.e., a real-world WebAssembly program, the call with the largest number of arguments passes 22 `i32` values (!), which clearly shows that eagerly generating all possible monomorphic combinations of call hooks ($4^{22} \approx 1.7 \times 10^{13}$) is simply not possible. Even in the small PolyBench programs, calls to functions with 6 arguments are common. For these programs, generating no more than 122 hooks on-demand is much better than generating all $4^6 = 4{,}096$ hooks for `call` instructions plus some more for other instructions.

5.6.6 *RQ5: Runtime Overhead*

Finally, Figure 5.5 shows how much runtime overhead the instrumentation imposes. On the y-axis, we show the runtime of the instrumented program relative to the original program, that is, a value of 1.0x means the runtime does not increase due to instrumentation. While we did run 20 repetitions of the measurements, the variance was too small for whiskers to be visible in the plot.

Similar to the results for code size, most of the hooks contribute only a small runtime overhead: `nop`, `unreachable`, `memory.size`, `memory.grow`, `select`, `drop`, and `unary` each impose less than 1.02x overhead, on average. Instrumenting for `return` or `call` hooks, which are sufficient for many interesting analyses at the function level, incurs a reasonable overhead of up to 1.3x and 2.8x, respectively. More expensive hooks are `begin` and `end` for observing blocks, which incur between 1.5x and 9.9x runtime overhead (depending on the program), 1.8x – 20x for `load`, up to 6.5x for `store`, 2x – 32x for `const`, 4x – 48.5x for operations on `locals`, and 2.6x – 77.5x for `binary` operations. When instrumenting for all hooks, the runtime overhead is between 49x and 163x. Note that the overheads for the PolyBench programs, which perform only numerical computations, are much higher than for the real-world workloads in PSPDFKit and the Unreal Engine. Typical WebAssembly programs call out to the host environment, e.g., to perform shading in WebGL, modify the DOM, or interact with some other Web API, so any overhead imposed by WASABI contributes only to parts of the total execution time.

When comparing the overhead results to existing heavyweight dynamic analysis frameworks for other languages, we find WASABI's overhead to be reasonable. The widely-used JavaScript analysis framework *Jalangi* reports overheads with the *empty* analysis in the same order

of magnitude, namely 26x during record plus 30x during replay, on average [Sen et al. 2013]. Similarly, the RoadRunner analysis framework for JVM byte code reports an average slowdown of 52x without any analysis [Flanagan and Freund 2010].

5.7 SUMMARY

This chapter presents WASABI, a general-purpose dynamic analysis framework for WebAssembly. The framework instruments WebAssembly binaries ahead-of-time and inserts code into the binary that calls into analysis hooks implemented in JavaScript. Besides being the first dynamic analysis framework for WebAssembly, WASABI addresses several unique challenges that did not occur in dynamic analysis tools for other platforms. In particular, we handle the problem of tracing polymorphic instructions with analysis functions that have a fixed type via an on-demand monomorphization of analysis hooks, and we statically resolve relative branch labels in control-flow constructs during the instrumentation. The high-level API provided to analysis authors allows for implementing otherwise complex analyses with a few dozens of lines of code, while still providing a complete view of the execution. Our evaluation with both compute-intensive benchmark programs and real-world web applications shows 1.02x to 163x runtime overhead, depending on the program and which instructions are analyzed, which is reasonable for heavyweight dynamic analysis.

We believe that WASABI provides a solid basis for various analyses to be implemented in the future. As an interesting challenge for future work, we envision cross-language dynamic analysis, in particular, to analyze applications that run both JavaScript and WebAssembly code.

6 FUZZM: FINDING AND MITIGATING MEMORY ERRORS

We demonstrate in Chapter 3 that vulnerabilities in memory-unsafe source languages can translate into exploitable WebAssembly binaries. Our results in Chapter 4 further show that many real-world binaries are potentially susceptible to such attacks. Unfortunately, remedies such as changing the WebAssembly language or rewriting programs in safer source languages are not easily implemented, so we must tackle memory errors in WebAssembly binaries with additional techniques.

This chapter presents FUZZM, the first binary-only fuzzer for WebAssembly. It can be applied to WebAssembly binaries and does not require source code access. We combine static binary instrumentation to detect memory errors on the stack and heap, instrumentation for collecting coverage, and efficient interaction between native input generation and the program running in a WebAssembly VM, into a fast and effective WebAssembly fuzzer. Besides as an oracle for fuzzing, our instrumentation also serves as a stand-alone hardening technique to prevent the exploitation of vulnerable binaries in production. We evaluate FUZZM with 28 real-world WebAssembly binaries, both well-known software projects and binaries found in the wild without access to their source code. FUZZM's performance is close to native AFL, despite WebAssembly being a bytecode and running in a VM. On average, FUZZM generates 1,232 inputs and triggers 40 crashes per program in 24 hours, at 321 program executions per second. When used for binary hardening, our instrumentation effectively prevents the exploits from Chapter 3 while imposing only 2% to 35% runtime overhead.

This chapter shares large parts of its material with a publication that is currently under submission [Lehmann, Torp, et al. 2021]. The project is joint work with the second author of the publication. The dissertation author developed the underlying instrumentation framework for WebAssembly, contributed to the canary and coverage implementations and the evaluation, and wrote major parts of the paper.

6.1 MOTIVATION AND CONTRIBUTIONS

While WebAssembly prevents some security issues by design (see Section 2.2), source-level vulnerabilities may still propagate to WebAssembly binaries, as discussed in Chapter 3. Surprisingly, memory vulnerabilities in WebAssembly binaries can sometimes be even more easily exploited than when the same source code is compiled to native code.

To find vulnerabilities, *coverage-guided greybox fuzzing* has proven to be extremely effective [Böhme et al. 2020, 2019; Hazimeh et al. 2020; Zeller et al. 2022]. For example, Google's OSS-Fuzz project has found tens of thousands of vulnerabilities in widely used software.[1] Among others, OSS-Fuzz employs *AFL*, widely regarded has a very practical fuzzer.[2] A greybox fuzzer automatically generates inputs that gradually explore the target program in the hope of triggering a vulnerability. For that, it requires (i) lightweight feedback from the execution, e.g., coverage information, to guide the input generation, and (ii) runtime oracles that make a vulnerability apparent, e.g., by crashing the program.

A greybox fuzzer for WebAssembly would be highly desirable, but several characteristics of WebAssembly must be taken into account. First, WebAssembly is a compilation target for multiple source languages, including C, C++, Rust, Go, and several others (Chapter 4). A fuzzer aimed at a specific source language could thus analyze only a subset of all real-world binaries. Second, the source code of a WebAssembly binary may not be available, e.g., when analyzing third-party websites, third-party libraries, or in-house legacy applications. Even if the source code is available, adopting a fuzzer into the development workflow is more difficult if it requires changes to the build system or a particular compiler. Third, even when compiling from the same source code, the security-relevant behavior of a program compiled to WebAssembly may differ from the same program compiled to native code. We illustrate this with an example in Section 6.2. As a result, fuzzing a binary compiled for another platform, e.g., x86 [Choi et al. 2019; Dinesh et al. 2020], is insufficient to expose memory errors in WebAssembly binaries. Taken together, these characteristics motivate a fuzzer explicitly targeting WebAssembly binaries. However, despite the overall success of greybox fuzzing and the increasing importance of WebAssembly, such a fuzzer currently does not exist.

1 [Serebryany 2017], https://google.github.io/oss-fuzz/
2 https://lcamtuf.coredump.cx/afl/

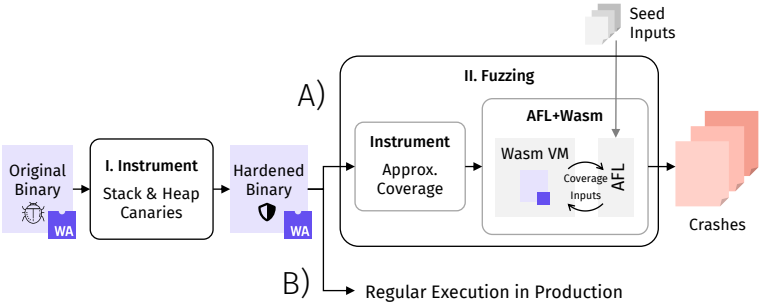


Figure 6.1: Overview of the main components of FUZZM.

This chapter presents FUZZM,[3] the first binary-only greybox fuzzer for WebAssembly. Its main components, shown in Figure 6.1, address the following three technical challenges. First, unlike native programs, WebAssembly lacks several built-in oracles that native fuzzers can rely on for finding suspicious program behavior. E.g., none of the current compilers targeting WebAssembly add stack canaries [Cowan et al. 1998; Prasad and Chiueh 2003], and due to WebAssembly's linear memory, overflows from the stack to the heap remain unnoticed, unlike in native code, where they would crash the program (Chapter 3). While tools like *AddressSanitizer* [Serebryany et al. 2012] can instrument source code to detect memory-related misbehavior, they do not apply to binaries. Instead, FUZZM employs static instrumentation, adding stack and heap canaries to the binaries in order to detect over- and underflows on the stack and heap at runtime. Besides fuzzing, the canaries are also useful for retroactively hardening existing WebAssembly binaries.

Second, a binary-only fuzzer cannot rely on compiler-inserted code to track coverage, which is what AFL and other fuzzers do [Böhme et al. 2019; Mathis, Gopinath, Mera, et al. 2019]. Even though there are dynamic instrumentation approaches for binaries, e.g., via *QEMU* or *DynInst*, they are architecture-dependent, not applicable to WebAssembly, and often suffer from high overheads. Our coverage instrumentation instead applies to unmodified WebAssembly binaries and tracks coverage efficiently.

The final challenge is efficiency, especially considering WebAssembly programs are executed on a virtual machine. A naïve approach may

3 "FUZZM" is a portmanteau word of "fuzzing" and "Wasm".

suffer from a prohibitively high program start-up time and expensive communication between the VM and the fuzzer. Instead, we integrate the WebAssembly VM that executes the target program with AFL's tried-and-tested input generation that runs natively. Here, WebAssembly's sandboxing can be an opportunity rather than a drawback: The target application and AFL can reside in a single address space, without the need for different processes separating the two.

The result of addressing the above challenges is a practical, effective, and efficient fuzzer for WebAssembly binaries. Our evaluation applies FUZZM to 28 programs, of which ten are well-known programs compiled from source code to WebAssembly, and 18 are WebAssembly binaries without any source code from our WASMBENCH dataset (Chapter 4). We find our approach to be effective, generating 1,232 inputs and triggering 40 unique crashes on average during 24 hours of fuzzing. The majority of the triggered crashes are due to our canary instrumentation. In terms of efficiency, FUZZM performs 321 program executions per second, despite requiring only a binary as input and running the program in a VM. Finally, we show that the canaries inserted by our instrumentation effectively prevent all three exploits against vulnerable WebAssembly binaries we described in Section 3.4. Due to the canaries' low runtime overhead (1.05x and 1.06x, for stack and heap canaries, respectively) the instrumentation serves, beyond fuzzing, as a stand-alone hardening for existing, vulnerable WebAssembly binaries.

CONTRIBUTIONS In summary, this chapter contributes:

- The first *binary-only fuzzer for WebAssembly* programs.
- A static binary instrumentation that inserts stack and heap canaries, which can be used to *harden existing WebAssembly programs* and as an *oracle in our fuzzer* (Section 6.3).
- Integrating AFL's input generation, a binary-only instrumentation that provides compatible coverage information, and a WebAssembly VM, into *efficient end-to-end fuzzing* (Section 6.4).
- Empirical evidence that FUZZM *effectively generates inputs and finds crashes* in real-world programs (Section 6.5).
- Empirical evidence that binaries hardened with our canary instrumentation run with *low runtime overhead* and *effectively thwart previously published exploits* (Section 6.5).

We also make our source code, data, and all results publicly available at https://github.com/fuzzm/fuzzm-project.

6.2 OVERVIEW

Our approach consists of two main components, as shown in Figure 6.1. First, a *binary-only canary instrumentation* (Section 6.3) that hardens WebAssembly applications by adding stack and heap canaries. Second, a *binary-only fuzzer for WebAssembly* (Section 6.4). The fuzzer integrates several components into an effective and efficient end-to-end approach: a second instrumentation, which gathers coverage information from WebAssembly binaries, a WebAssembly VM, and the input generation abilities of the proven AFL tool. The remainder of this section illustrates our approach with a motivating example; subsequent sections fill in the details.

EXAMPLE The program in Listing 6.1 suffers from a textbook buffer overflow on the stack (line 3) that can be exploited given the right inputs (lines 15 and 3). Because of differences in compilers, system libraries, and protection features, the vulnerability is not exploitable when compiled to a native architecture, e.g., x86-64, but it can be exploited when compiled to WebAssembly (compare with Chapter 3). Figure 6.2a and b show the stack layout of the program when executed as a native binary (compiled with GCC) and as a WebAssembly binary (compiled with Emscripten). Because the arrays `input1` and `input2` are stored in a different order on the stack, an attacker overflowing `input1` cannot change the program behavior in the native binary, but *can* do so in WebAssembly. It is thus important to fuzz the WebAssembly binary, and not only a native binary compiled from the same source.

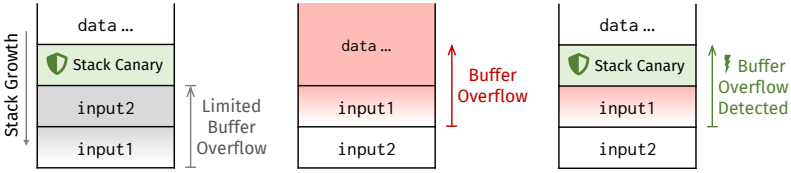
CANARY INSTRUMENTATION To detect executions that exploit vulnerabilities like in the above example, we present a binary-only instrumentation technique that adds protections in the form of stack and heap canaries. The approach instruments every function in the binary with code that inserts a canary onto the stack frame upon entry and checks it upon function exit. Beyond overflows in the stack, the instrumentation also detects memory violations on the heap by surrounding heap chunks with canaries. Figure 6.2c shows the inserted canary on the stack of the example program. An attack writing beyond the buffer

```c
void vulnerable() {
    char input1[8];
    scanf("%16s", input1);  // Buffer overflow!
    char input2[8];
    scanf("%8s", input2);
    // More code...
}
int read_int() {
    int i;
    scanf("%d", &i);
    return i;
}
int main(int argc, char** argv) {
    char data[10] = "some data";
    if (read_int() == 42) {  // Input, figured out by fuzzer.
        vulnerable();
    }
    if (strcmp(data, "some data") == 0) {
        puts("equal");
    } else {
        puts("not equal");
    }
}
```

Listing 6.1: Example program with a memory vulnerability (simplified).

will overwrite the canary, which the instrumented binary will detect and abort execution.

The canary instrumentation serves two purposes, marked with A) and B) in Figure 6.1. A) The primary purpose explored in this chapter is as an oracle during fuzz-testing. If a fuzzer successfully generates an input that causes an overflow (e.g., of `input1` in the example), it might remain unnoticed, unless the overflow causes a crash. Analogous to dynamic checks for memory corruptions in native programs [Dinesh et al. 2020; Prasad and Chiueh 2003; Robertson et al. 2003], our stack and heap canary instrumentation provides a precise test oracle that warns about memory corruptions observed during execution. B) Beyond fuzzing, our instrumentation also serves as a hardening technique for binaries running in production. The inserted code mitigates exploits by detecting overflows at runtime, terminating the program, and hence preventing exploitation. As we show in our evaluation, this protection comes with a low runtime overhead and be can be applied to large, real-world binaries compiled from C, C++, and Rust.



(a) Compiled to native code, e.g., x86. Overflow is not detected but also not exploitable.

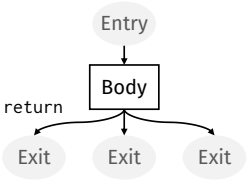
(b) Compiled to WebAssembly. Overflow is undetected and thus exploitable!

(c) Instrumented WebAssembly from b. Overflow is now detected and no longer exploitable.

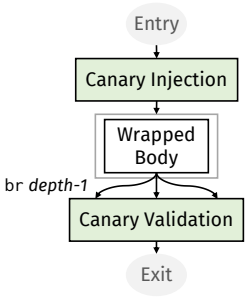
Figure 6.2: Stack layouts of the example program from Listing 6.1.

FUZZING WEBASSEMBLY The second main component of our approach is the actual fuzzer. We use a greybox fuzzing approach based on the widely used AFL fuzzer and its proven input generation [Böhme et al. 2019; Klees et al. 2018]. Starting from some seed input(s), our fuzzer repeatedly executes the program, and the program gathers coverage feedback, which is used to mutate inputs until triggering a crash. In the example program, the fuzzer's input generation eventually figures out to start the input with `"42"` (line 15) to explore more behavior in function `vulnerable`. However, in said function, native AFL fails to detect a vulnerability due to the different stack layouts between the native and WebAssembly binaries, whereas FUZZM finds a crashing input after few minutes of fuzzing.

Applying AFL-style greybox fuzzing to WebAssembly is non-trivial for two reasons. First, the fuzzer requires coverage information, which native AFL obtains by inserting code when compiling the program from source. However, we want to fuzz WebAssembly binaries without requiring access to the source code. We hence present a binary-only instrumentation technique to gather AFL-compatible coverage information from WebAssembly binaries. Second, to be practical, fuzzers must execute the program-under-test hundreds of times per second. We present several technical contributions that efficiently integrate a WebAssembly VM that runs the program with native input generation of AFL in the same process. More details of both points are given in Section 6.4.



(a) Original function, `returning` from multiple locations in the body.



(b) Instrumented with canaries.

Figure 6.3: Control-flow graphs of a WebAssembly function before and after the instrumentation. The rectangles represent subgraphs.

6.3 CANARY INSTRUMENTATION

To detect memory violations, we present a reliable static instrumentation that inserts stack and heap canaries into WebAssembly binaries. Similar to native programs [Ha et al. 2018; Nikiforakis et al. 2013; Prasad and Chiueh 2003; Robertson et al. 2003], the basic idea is to surround memory regions with a special value, called the *canary*, and to check whether this value gets overwritten. Our approach differs from prior work in four ways. First, we are the first to present a canary-based protection against buffer overflows in WebAssembly at all. Current compilers, e.g., Clang and Emscripten based on LLVM, do not implement canaries, so many already existing WebAssembly binaries are potentially vulnerable. Second, in contrast to, e.g., compiler-inserted canaries or the popular AddressSanitizer [Serebryany et al. 2012], our approach does not require source code but instruments WebAssembly binaries directly. This allows us to retroactively harden existing binaries. Finally, instrumenting WebAssembly provides novel challenges, e.g., due to structured control-flow, multiple returns, and relative branch target labels. The implementation of the static instrumentation builds on the WebAssembly binary parser and AST libraries we have developed for our WASABI project (Chapter 5).

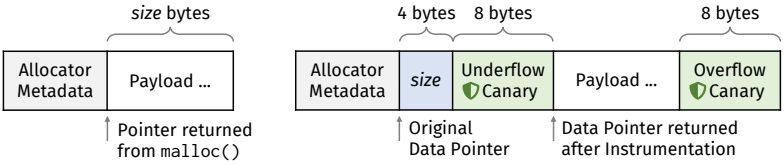
Algorithm 6.1: Stack canary instrumentation. The ++ operator denotes concatenation.

```
1:  procedure InstrumentFunction(body)
2:      canary ← eight randomly generated bytes
3:      body ← InjectCanary(canary) ++ body
4:      body ← block ++ body ++ end                    ▷ Wrap original body
5:      depth ← 0
6:      for instr in body do
7:          if OpensBlock(instr) then                  ▷ Track block depth
8:              depth ← depth + 1
9:          else if ClosesBlock(instr) then
10:             depth ← depth − 1
11:         if instr = return then                     ▷ Redirect and replace returns
12:             instr ← br (depth − 1)
13:     body ← body ++ ValidateCanary(canary)
```

6.3.1 *Stack Canaries*

To detect buffer overflows that write beyond the current stack frame, Fuzzm performs three transformations on each function in a binary, as illustrated in Figure 6.3. First, we insert a preamble that injects a random canary value onto the stack in linear memory (first green box in Figure 6.3b). Second, we need to make sure that the canary is checked when returning from the function. For that, we wrap the function body in a new block and rewrite all original `return` instructions to branches to the end of the wrapped block (middle box), giving the code a single unique exit point. Finally, we append a canary validation postamble to the function (second green box).

Algorithm 6.1 presents the instrumentation of a given function in more detail. Line 2 generates a random 8-byte canary value. Line 3 prepends the injection code to the function, which writes the canary at runtime into the stack in linear memory. For brevity, the code of InjectCanary is omitted here, but it decrements the stack pointer by 16 bytes (due to stack alignment) and then writes the random canary to that address with an `i64.store` instruction. Because WebAssembly does not have registers, unlike native code, the stack pointer is stored in a global variable, which needs to be statically identified. For WASI



(a) Without canaries. **(b)** With canaries.

Figure 6.4: Heap chunks, before and after instrumentation.

applications, it is the first global variable; otherwise, heuristics to identify the stack pointer can be used (Chapter 4).

WebAssembly functions frequently return from multiple locations in their body, which raises the question where to insert code to validate the canary. Unlike in native code, there is no single function epilogue that clears the stack and returns to the caller. One possible approach is to separately instrument every `return` instruction with a copy of the validation code, but this would increase code size considerably. Instead, we rewrite all original return instructions to branch to a single location and then insert the validation code there. Thus, the entire function is wrapped into a new WebAssembly `block` (line 4 of Algorithm 6.1). Then, each `return` instruction in the function body is replaced with a branch to the end of that new block (lines 6–12), keeping the semantics of the original code. Because WebAssembly has relative branch labels instead of absolute code addresses, the depth variable (line 5) is needed to keep track of the number of nested blocks around the current instruction, to emit the correct branch target.

Finally, line 13 appends the postamble ValidateCanary for validating the canary before the function returns. It loads the canary from memory and compares it against the known, correct canary value. If they differ, an `unreachable` instruction is executed, which aborts the program and thwarts potential exploits. If the canary was intact, the stack pointer is adjusted by adding 16 bytes, restoring the stack layout of the uninstrumented binary.

6.3.2 *Heap Canaries*

Fuzzm also detects and prevents memory violations on the heap. To illustrate the problem, Figure 6.4a shows the typical layout of a *heap chunk*, i.e., a region of dynamically allocated memory returned by func-

Algorithm 6.2: Instrumentation of heap allocation functions.

```
1:  procedure InstrumentAllocFunction(f)
2:      local_req_size ← AddFreshLocal(f)
3:      f.body ←                                          ▷ Insert preamble
4:          SaveAllocRequestSize(f, local_req_size) ++
5:          IncreaseAllocSize(f, local_req_size) ++ f.body
6:      local_data_ptr ← AddFreshLocal(f)
7:      f.body ← f.body ++                                ▷ Insert postamble
8:          SaveDataPointer(local_data_ptr) ++
9:          WriteSizeAndUnderflowCanary(local_req_size, local_data_ptr) ++
10:         WriteOverflowCanary(local_req_size, local_data_ptr) ++
11:         AdjustDataPointer(local_data_ptr)
```

tions like `malloc`. The *payload* is where the user will read and write data to. The *metadata* precedes or follows the payload and is used for bookkeeping by the allocator. An attack that over- or underflows a buffer in the payload and that writes into adjacent metadata can yield a dangerous arbitrary write primitive [Anonymous 2001; Kaempf 2001].

To detect such violations on the heap, FUZZM instruments heap allocation and deallocation functions in the binary. Our instrumentation inserts canary values before and after the payload, as illustrated by Figure 6.4b, enabling us to detect both overflows and underflows. The canaries are inserted into the heap chunk by instrumented versions of allocation functions (Section 6.3.2.1) and checked by instrumented versions of deallocation functions (Section 6.3.2.2).

6.3.2.1 *Insert Canaries on Heap Allocation*

Our instrumentation adds code to allocation functions from the C standard library, i.e., `malloc`, `calloc`, and `realloc`. Other functions that allocate by calling the low-level libc functions in turn thus profit from our protection as well. The instrumentation inserts code into allocation functions in two places: a *preamble* in the beginning and a *postamble* at the end, as outlined in Algorithm 6.2. The added code has three high-level goals. First, the allocation size needs to be increased to fit the canaries (line 5). Second, the canary values must be written to memory (lines 9 and 10). Third, the data pointer returned by the allocator needs to be adjusted before passing it to the user, such that it points to the

now shifted payload (line 11). Additionally, we add two new locals to the function (lines 2 and 6) to save data (lines 4 and 8) used later by the inserted code. Our canary code is interposed between the original allocator and the client code requesting the allocation, and it is transparent to both. From the allocator's point of view, the payload is the whole region after the metadata in Figure 6.4b, including the inserted canaries. For the client code requesting the allocation, the payload is only the region between the canaries, matching the originally requested size.

In the preamble, we retrieve the originally requested allocation size, save it to a local for later (line 4), and increase it by 20 bytes (line 5). The additional 20 bytes make space for two 8-byte canaries and a 4-byte size field. The latter is required for the checking code (Section 6.3.2.2). After the preamble, the original allocator code performs the regular memory allocation routine and produces a *data pointer* to the beginning of the freshly allocated memory. Then follows our inserted postamble. It saves the data pointer to a local (line 8). Starting at this pointer, the inserted code writes the chunk size and underflow canary (line 9), i.e., before the payload. Then, line 10 writes the overflow canary after the payload, i.e., to *data_ptr* + *size* + 12. Finally, the data pointer is adjusted to point past the underflow canary (line 11). This pointer is finally returned to the calling code. The result of this instrumentation is that memory allocation functions create chunks as shown in Figure 6.4b.

6.3.2.2 *Check Canaries on Heap Deallocation*

FUZZM checks whether the heap canaries are valid whenever a heap chunk gets deallocated. Similar to allocation functions, we currently instrument libc's `free` and `realloc`, which are also at the basis, e.g., of C++ `delete`. The validation code is inserted in the beginning of the deallocation functions, as corrupted allocator metadata could lead to an exploit if not checked at this stage. The validation code loads and checks both canaries around the allocation payload, aborting the program with an `unreachable` instruction if they have been altered.

When to check the heap canaries is a trade-off between performance, complexity of the instrumentation, and the likelihood of catching buffer overflows. FUZZM performs this check during deallocation, which is inexpensive, as every canary is checked at most once, but has the disadvantage of not catching overflows in chunks that are never deallocated. Others have proposed more aggressive techniques that check canaries

at every memory read or write [Serebryany et al. 2012] or validate canaries at every syscall [Nikiforakis et al. 2013]. While these approaches may detect more attacks in production, they also impose a larger runtime overhead, which makes them less suited for binary hardening and efficient fuzzing.

6.4 BINARY-ONLY FUZZER

In this section we show the other components of our first binary-only fuzzer for WebAssembly. We take a greybox fuzzing approach and build upon the popular AFL fuzzer, reusing its highly effective input generation. Because AFL usually targets programs with source code available and does not support WebAssembly, there are two key challenges to address. The first challenge is gathering compatible coverage information during the execution of a WebAssembly program (Section 6.4.1). The second challenge is how to integrate execution of the program in a WebAssembly VM with AFL running natively, in a way that allows for performing hundreds of program executions per second, which is the level of efficiency AFL provides for natively compiled code (Section 6.4.2).

6.4.1 *Coverage Instrumentation*

Greybox fuzzing is effective because it relies on lightweight feedback during program execution to steer the fuzzer. To collect that feedback, native AFL compiles applications from source, inserting code to track an approximate form of path coverage [Böhme et al. 2019], which is stored into a *trace bits array*. Unlike native AFL, we fuzz WebAssembly binaries without access to their source code, and hence, cannot instrument during compilation. AFL also offers a QEMU mode for dynamic binary instrumentation, but it comes with a high performance overhead and naturally is architecture-specific, offering no WebAssembly support. Instead, FUZZM gathers coverage via static binary instrumentation that inserts code at all branches to extract AFL-compatible coverage information.

As a prerequisite for instrumentation, the approach determines all branches. This is non-trivial because of WebAssembly's structured control flow and relative branch target labels. Algorithm 6.3 traverses each function of a binary and marks instructions that transfer control-flow.

Algorithm 6.3: Insertion of AFL coverage instrumentation.

```
1:  procedure InstrumentApproximateCoverage(f)
2:      depth ← 0
3:      targets ← {}
4:      for instr in f.body do
5:          if instr ∈ {block, if, else, loop} then
6:              if instr ∈ {block, if, loop} then
7:                  depth ← depth + 1
8:              if instr ∈ {if, else, loop} then
9:                  mark(instr)
10:         else if instr = br_if n then
11:             targets ← targets ∪ {depth − n}
12:             mark(instr)
13:         else if instr = br_table(jmp_targets) then
14:             targets ← targets ∪ ⋃_{t∈jmp_targets} {t − n}
15:         else if instr = end then
16:             if depth ∈ targets then
17:                 mark(instr)
18:                 targets ← targets \ {depth}
19:             depth ← depth − 1
20:     mark(f.body[0])
```

This includes `br_if` (line 12), but also `if`, `else` and `loop` blocks (line 9) since they are structured control flow constructs. Furthermore, the algorithm keeps track of the block nesting depth, i.e., depth is incremented at `block`, `if` and `loop` instructions (line 7) and decremented at an `end` instruction (line 19). Keeping track of the depth is necessary to resolve relative branch labels to concrete, unambiguous targets. Whenever the algorithm encounters a conditional branch (either `br_if` on line 10 or `br_table` on line 13), it adds the target depth of the branch instruction to the targets set. At every `end` instruction, the algorithm then checks whether the depth of that `end` instruction is in the targets set (line 15). In case it is present, the `end` is a target of some branch, and is therefore also marked for instrumentation (line 17). In addition to the instructions marked by Algorithm 6.3, FUZZM also marks the beginning of every function (line 20) since an indirect function call also represents a branch.

Given the branching points identified by Algorithm 6.3, FUZZM instruments each of them. It adapts the coverage mechanism described in the AFL documentation[4] to WebAssembly. Every branch target is assigned a random identifier. Whenever a branch is taken, the instrumentation computes an index that combines the identifier of the current branch, `<CUR_LOCATION>`, and the identifier of the previous branch, `<PREV_LOCATION>`. The code then increments the element of the trace bits array corresponding to that index. To initialize the trace bits array, FUZZM also injects code into the `_start` function of the binary, which is the program's entry point.

6.4.2 *Integrating a WebAssembly VM and AFL*

AFL for native programs is heavily optimized towards performing as many executions of the target program as possible. In the following we present techniques that allow FUZZM to achieve a similar level of efficiency. Our implementation targets WebAssembly binaries using the WASI syscall interface, i.e., applications running on a compliant VM, such as Wasmer or Wasmtime (Section 2.3)

AVOIDING VM RESTARTS With the fuzzed program running on a VM, one possible approach is to start a new instance of the VM for each run of the target program. However, doing so easily results in more time spent on starting the VM and compiling the WebAssembly bytecode to native code than on running the target program. Instead, FUZZM starts the VM once, lets the VM precompile the target WebAssembly binary, and then reuses both throughout the fuzzing process. FUZZM uses the Wasmtime C API[5] to separately compile and run WebAssembly modules. For every newly generated input, the fuzzer instantiates the WebAssembly module that was already compiled to native code and calls the exported `_start` function.

ACCESSING THE TRACE BITS ARRAY To generate new inputs, the fuzzer needs to access coverage information stored in the trace bits array. The native version of AFL starts the target program as a subprocess and accesses the trace bits array via shared memory. Instead, FUZZM exploits the fact that the VM sandbox allows for the target program and AFL's own code to share a single address space. Our approach in-

4 https://lcamtuf.coredump.cx/afl/technical_details.txt
5 https://docs.wasmtime.dev/c-api/

serts an accessor function into the binary that returns a pointer to the trace bits array in the WebAssembly memory of the target program. After each execution of the program, FUZZM calls the accessor function, extracts the trace bits, and passes them to AFL.

DETECTING CRASHES The native version of AFL detects crashes by looking for fatal signals (SIGSEGV, SIGKILL, SIGABRT) in the target program. However, WASI does not support signals, so FUZZM uses the trap system of WebAssembly to determine when the target program crashes. To this end, the oracles that FUZZM inserts (Section 6.3) trigger an `unreachable` trap when they detect an overflow or underflow. In addition, WebAssembly has other runtime errors that also indicate faulty behavior, e.g., the type checking of indirect calls (2.2). When the `_start` function terminates, FUZZM checks if the termination was triggered by a trap, and in that case, marks it as a crash.

KILLING LONG-RUNNING EXECUTIONS Randomly generated inputs may trigger long-running or even non-terminating executions. To prevent those from slowing down overall fuzzing, native AFL runs the fuzzed program in a separate process, which is killed after a timeout. However, since the WebAssembly VM and the other parts of the fuzzer are running in the same process, FUZZM implements two mechanisms to stop long-running executions. First, it uses a "soft killing" mechanism based on a separate thread that interrupts the thread of the target program after a dynamically set timeout. Second, for programs that do not react to the interrupt, a second "hard killing" mechanism restarts the entire VM after a longer timeout.

6.5 EVALUATION

We evaluate FUZZM along the two use-cases presented in Section 6.2. First, *end-to-end fuzzing* of WebAssembly binaries:

RQ1 *Effectiveness*: How effective is FUZZM at covering paths and finding crashes?

RQ2 *Robustness*: How robust is the instrumentation when applied to real-world binaries?

RQ3 *Efficiency*: How efficient is fuzzing with FUZZM?

Second, *hardening binaries for production* through canaries:

RQ4 *Effectiveness*: How effective are the inserted canaries at preventing previously demonstrated exploits?

RQ5 *Efficiency*: How much overhead do the canaries impose?

For reproducibility, future research, and practicioners, we make our source code, data, and all experimental results available at https://github.com/fuzzm/fuzzm-project.

6.5.1 *Experimental Setup*

BENCHMARKS We use three sets of benchmarks (Table 6.1). Benchmarks 1 to 7 are real-world applications and libraries, compiled to WebAssembly with WASI. The selected versions of those programs suffer from known memory vulnerabilities, which a fuzzer might help to uncover and fix. We select these applications from https://www.exploit-db.com/ and https://www.rapid7.com/db/ and confirmed that each vulnerability could be exploited after compilation to WebAssembly.

Benchmarks 8 to 10 are from the LAVA-M suite [Dolan-Gavitt et al. 2016]. We omit the *who* program since it reads the list of mounted file systems, which is not yet supported by WASI. AFL, and by extension also FUZZM, is known to perform poorly on LAVA-M as the fuzzer does not handle the multi-byte constraints of LAVA-M bugs well [Rawat et al. 2017]. Because LAVA-M has been criticized for not being representative of real bugs [Klees et al. 2018], we would have liked to instead evaluate against the more modern Magma benchmark suite [Hazimeh et al. 2020]. However, all seven Magma programs use features not yet supported by WASI, such as networking, threads, and long jumps.

Benchmarks 11 to 28 are real-world WebAssembly binaries, which we took from our WASMBENCH dataset (Chapter 4). We select 18 binaries that run without error (before instrumentation) in the Wasmtime VM with WASI. Among them are large applications, such as SQLite and Clang compiled to WebAssembly, but also several smaller binaries, such as a formatter (*canonicaljson*), a template engine (*handlebars-cli*), and an interpreter (*bfi*).

COMPILATION We compile the source code from the first and second set using a version of Clang that targets WebAssembly[6] and then instrument the binaries as described in Sections 6.3 and 6.4. For comparing against native AFL, we also compile the benchmarks with AFL's

6 https://github.com/WebAssembly/wasi-sdk

GCC wrapper. Since AFL's instrumentation is applied during compilation, this is not completely true to our scenario where source code is not available and binaries are compiled for production. To level the ground, we do not use AddressSanitizer or any other oracle that requires source code, neither for FUZZM nor AFL. An alternative baseline would be the QEMU mode of AFL, which uses dynamic instrumentation, but since it is much slower than normal AFL, it would give FUZZM an unfair advantage. For the third benchmark set, we do not have any source code, which highlights the need for a binary-only fuzzer such as FUZZM.

REPETITIONS AND SYSTEM CONFIGURATION We fuzz each benchmark five times for 24 hours, both with FUZZM and AFL [Klees et al. 2018], and report the mean across repetitions and the 95% confidence intervals. All experiments are performed on two machines, each with two Intel Xeon 12-core 24-thread CPUs running at 2.2 GHz, using 256 GB of system memory, using Ubuntu 18.04 LTS and AFL 2.57b.

6.5.2 *RQ1: Effectiveness of Fuzzm*

We evaluate the end-to-end effectiveness of fuzzing WebAssembly binaries with FUZZM by measuring how many inputs that cover new paths are generated, how many unique crashes are triggered, and if the canary instrumentation helps in finding those crashes. Table 6.1 gives the results, where the numbers for FUZZM are presented in the left block. What AFL reports as explored "paths" (e.g., in its status screen) is defined as the total number of added inputs to the test case queue. A new input is only added to the queue if it covers a new path as per AFL's coverage metric.[7] To avoid confusion, we report this number in the "Generated Inputs" column, not as paths. Unique crashes are counted using AFL's notion of uniqueness, i.e., two crashes are merged if they are found in executions with the same coverage map. Different crashes as per this metric may sometimes have the same root cause [Klees et al. 2018]. Finally, the "Execs/sec" columns show the average number of executions of the benchmark program per second.

We find that FUZZM successfully generates many hundreds of inputs that cover new paths for the programs, on average 1232 inputs per benchmark after 24 hours of fuzzing. We see that this works even for complex programs such as *flac* (benchmark set 1) or *sqlite* (set 3). For

7 https://github.com/google/AFL/blob/master/README.md

Table 6.1: Benchmark sets and fuzzing results (5 × 24 hours). The reported numbers are mean and 95% confidence intervals.

#	Benchmark	Fuzzm (WebAssembly binaries)					AFL (native, built from source)		
		Generated Inputs	Crashes, of those: caused by Canaries			Execs/sec	Generated Inputs	Crashes	Execs/sec
			Total	Stack Can.	Heap Can.				
Benchmark set 1 – Real-world applications and libraries:									
1	`abc2mtex`	1678.6 ± 22.5	267.9 ± 6.6	224.5 ± 5.5	4.2 ± 1.6	359.4 ± 42.4	2999.2 ± 82.6	820.1 ± 63.2	879.6 ± 212.1
2	`flac`	607.5 ± 11.5	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	497.0 ± 7.1	1617.1 ± 75.1	0.0 ± 0.0	1228.1 ± 391.3
3	`jbig2dec`	2199.1 ± 13.8	0.1 ± 0.2	0.0 ± 0.0	0.0 ± 0.0	66.2 ± 51.6	3330.1 ± 49.8	0.0 ± 0.0	437.7 ± 264.8
4	`libpng`	727.0 ± 10.4	96.1 ± 4.0	0.0 ± 0.0	77.2 ± 3.8	430.9 ± 67.6	1123.4 ± 25.3	176.4 ± 2.8	692.5 ± 481.6
5	`libtiff`	860.3 ± 9.1	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	868.5 ± 64.3	2542.5 ± 33.6	0.0 ± 0.0	953.7 ± 467.8
6	`openjpeg`	5322.2 ± 3611.0	90.3 ± 39.3	7.7 ± 4.5	8.6 ± 10.4	457.3 ± 242.8	1779.7 ± 39.8	90.7 ± 3.4	605.4 ± 435.4
7	`pdfresurrect`	840.1 ± 207.0	54.5 ± 8.4	15.1 ± 3.4	17.2 ± 5.4	228.1 ± 194.1	1011.0 ± 226.3	129.9 ± 29.2	701.5 ± 369.2
Benchmark set 2 – From LAVA-M [Dolan-Gavitt et al. 2016]:									
8	`base64`	200.6 ± 7.2	34.4 ± 1.5	0.0 ± 0.0	0.0 ± 0.0	225.8 ± 83.7	355.8 ± 29.1	0.1 ± 0.2	514.3 ± 276.6
9	`md5sum`	395.2 ± 20.6	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	324.6 ± 202.7	317.9 ± 8.9	0.0 ± 0.0	202.4 ± 60.7
10	`uniq`	213.1 ± 22.8	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	678.2 ± 109.6	113.0 ± 3.7	0.3 ± 0.4	415.0 ± 337.5
Average (sets 1 & 2)		1304.4	54.3	24.7	10.7	413.6	1518.9	121.7	663.0

Table 6.1 continued.

#	Benchmark	FUZZM (WebAssembly binaries)					AFL (native, built from source)		
		Generated Inputs	Crashes, of those: caused by Canaries			Execs/sec	Generated Inputs	Crashes	Execs/sec
			Total	Stack Can.	Heap Can.				
Benchmark set 3 – Real-world WebAssembly binaries from WasmBench:									
11	`bf`	271.4 ± 27.5	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	28.6 ± 7.4			
12	`bfi`	2158.0 ± 108.8	97.8 ± 26.1	0.0 ± 0.0	30.8 ± 12.5	286.4 ± 40.1			
13	`canonicaljson`	357.4 ± 15.8	180.4 ± 6.9	0.0 ± 0.0	0.0 ± 0.0	428.2 ± 193.4			
14	`clang`	6.0 ± 0.6	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	36.8 ± 0.7			
15	`colcrt`	231.0 ± 7.6	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	83.6 ± 50.3			
16	`handlebars-cli`	882.2 ± 69.5	39.4 ± 0.7	0.0 ± 0.0	39.4 ± 0.7	222.0 ± 136.8			
17	`hq9_plus_rs`	227.0 ± 15.5	42.8 ± 0.9	0.0 ± 0.0	42.8 ± 0.9	111.0 ± 42.0	N/A		
18	`jq`	1.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	334.0 ± 7.3			
19	`libxml2`	177.8 ± 5.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	70.0 ± 1.5	(As those samples are binary-only WebAssembly programs, there is no native counterpart to fuzz with AFL.)		
20	`qjs`	9640.0 ± 96.0	140.4 ± 61.4	3.4 ± 3.0	11.6 ± 8.6	387.0 ± 20.0			
21	`qr2text`	1.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	84.6 ± 0.4			
22	`rev`	161.4 ± 6.1	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	140.4 ± 20.1			
23	`save`	23.2 ± 0.7	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	46.4 ± 1.2			
24	`sqlite`	4284.2 ± 738.3	2.4 ± 4.2	0.0 ± 0.0	0.0 ± 0.0	359.4 ± 116.5			
25	`viu`	12.4 ± 1.8	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	707.2 ± 7.6			
26	`wasi-example`	213.4 ± 7.1	50.2 ± 0.9	0.0 ± 0.0	0.0 ± 0.0	764.4 ± 81.8			
27	`wasm-interface`	5.2 ± 0.4	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	622.2 ± 10.3			
28	`zxing_barcode`	2799.2 ± 236.2	32.2 ± 5.6	0.0 ± 0.0	17.4 ± 5.2	131.4 ± 62.2			
Average (all sets)		1232.0	40.3	9.0	8.9	320.7			

benchmark sets one and two, where the fuzzed programs are known, we provide the fuzzer with seed inputs. For set three, we only provided an empty file as seed input, which could explain the lower average number of generated new inputs. In terms of crashes, FUZZM finds 40.3 crashes per benchmark on average. For *libpng* and *pdfresurrect*, FUZZM generates crashing inputs that produce the exact stack trace of the native proof-of-concept exploits, confirming that FUZZM can find real bugs in WebAssembly binaries.

To better understand how FUZZM exercises a program over a 24-hour period of fuzzing, Figure 6.5 shows the number of generated inputs that cover new paths over time, for four programs. As the plots for found crashes strongly correlate with them, we omit the former. As is typical for fuzzers, the majority of behaviors are detected early on, usually within the first couple of hours (6.5a/b/c). Then, the number of new inputs and crashes often saturates, especially for the LAVA-M benchmarks (b). For some benchmarks, e.g., *qjs*, FUZZM still finds new inputs when given more time (d). The confidence intervals are small, except for *openjpeg* (a) and *pdfresurrect*, where the results vary considerably across runs. Overall, these findings are consistent with previous work, and show that running a fuzzer multiple times is important to obtain statistically meaningful results [Klees et al. 2018].

COMPARISON As FUZZM is the first binary-only fuzzer for WebAssembly, we cannot directly compare to any baseline. However, to put the number of generated inputs and crashes into perspective, we also present results for native AFL on the right side of Table 6.1. This is only meant as a rough frame of reference, as a fair comparison is not possible for several reasons. First, FUZZM requires only the binary as input, whereas AFL applies its instrumentation during compilation from source. Second, our notion of branches may differ from branches considered by AFL due to different compilers and their target-dependent optimizations. Third, unlike in native binaries, all libraries (including *libc*) are statically linked in WebAssembly, which increases the amount of code FUZZM has to instrument and fuzz. Fourth, the number of generated inputs naturally depends on the execution speed, which is in principle lower on WebAssembly compared to native (Section 6.5.4). Finally, benchmark set three is only available as WebAssembly binaries without source code, which is why we cannot compare against AFL for these benchmarks.

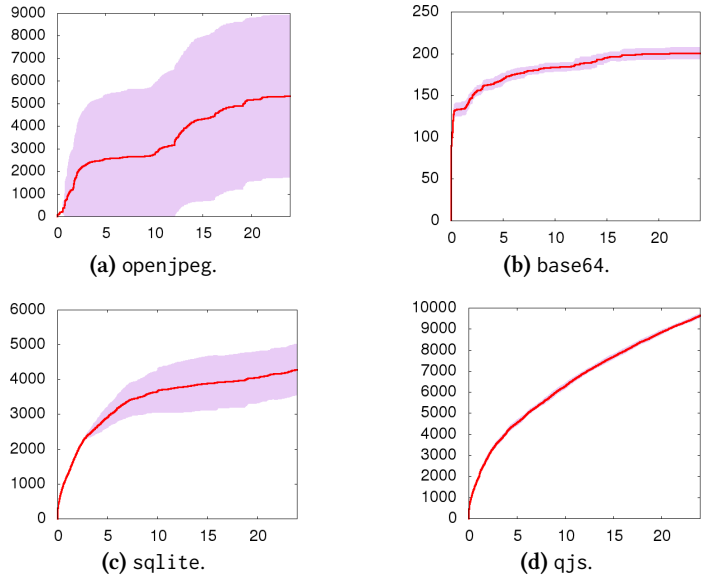


Figure 6.5: Average number of generated inputs covering new paths (y-axis) over 24 hours of fuzzing (x-axis), with 95% confidence intervals.

From the data on benchmark sets one and two, we can see that FUZZM generates, on average, a number of inputs similar to AFL on native programs (1304 vs. 1519). In terms of crashes, AFL triggers 122 on average, which is roughly twice as many as FUZZM's 54. One outlier is *base64* where FUZZM triggers 34 unique crashes but AFL triggers only one crash in one of the performed runs. The 34 WebAssembly crashes are triggered by a built-in sanity check of the executing VM, which is not present in native binaries and explains why these crashes are not detected by AFL. For programs where FUZZM does not find any crashes (e.g., *flac*), AFL does not either.

For the LAVA-M benchmarks, both FUZZM and AFL fail to trigger any of the bugs injected by the LAVA tool. This is surprising since prior work that also compares with AFL report at least some bugs detected for *uniq* and also sometimes for *base64* [P. Chen and H. Chen 2018; B. Zhang et al. 2017]. Manually investigating several LAVA-M bugs shows that a crash depends on the memory allocator allocating a new chunk at the exact location of some previously freed chunk. We attribute the

fact that neither FUZZM nor AFL finds these bugs to differences in (versions of) the used memory allocator (compare with Section 4.3.4.2) and to differences across versions of AFL.

EFFECTIVENESS OF CANARIES Besides being the first approach for fuzzing WebAssembly binaries, FUZZM contributes canary-based oracles to detect stack and heap over- and underflows. We measure how much these oracles contribute to the crashes detected by FUZZM by distinguishing crashes caused by the oracles from other crashes. The three "Crashes" columns of Table 6.1 show the results. The stack and heap canaries are responsible for 22.2% and 22.0% of all detected crashes, respectively, on all benchmarks, and 45.5% and 19.7% on benchmark sets one and two, a significant proportion of all crashes.

Summary: Applied to well-known applications, libraries, and real-world WebAssembly binaries, FUZZM triggers an average of 40 unique crashes and generates 1,232 inputs within 24 hours of fuzzing, which are similar results as AFL applied to native programs. Our canary-based oracles detect about half of all detected crashes, and hence, contribute significantly to the effectiveness of FUZZM.

6.5.3 *RQ2: Robustness of Instrumentation*

Our instrumentation should not affect the semantics of the program, except in the presence of overflows, where the canaries should terminate the program. To validate the robustness of FUZZM's instrumentation, we compare the output of the non-instrumented against the instrumented versions of the benchmarks. For each benchmark in the first two sets, we collect at least ten different inputs, totaling 138 test cases (Table 6.3). We sample these inputs from different websites[8], and for programs where we could not find sufficiently many examples online, e.g., *pal2rgb*, we generate inputs by, e.g., converting images to the pal format. As we do not have source code or documentation for benchmark set three, we do not generate test inputs for those programs. For the binaries and test inputs shown in Table 6.3, we verify that the outputs produced by the instrumented binaries are equivalent to the outputs produced by the uninstrumented binaries for all 138 test cases. As additional evidence for the robustness of our instrumentation, we find

8 E.g., https://filesamples.com/.

that all binaries we instrumented pass built-in WebAssembly validation, which performs, e.g., type-checking of instructions and functions.

Summary: Test runs of the benchmarks and the static validation applied to each WebAssembly module before its execution show that the binary instrumentation applied by FUZZM preserves the semantics of the original program.

6.5.4 *RQ3: Efficiency of End-to-End Fuzzing*

Effective fuzzing requires many repeated executions of the target program in limited time. In FUZZM, fast execution is even more challenging due to WebAssembly being a bytecode language and applying our instrumentation at the binary level.

Table 6.1 lists the average program executions per second during fuzzing in the column "Execs/sec". With an average speed of 321 executions per second, FUZZM is able to quickly explore many paths. As already described in Section 6.5.2, comparisons between FUZZM and AFL are possible in broad strokes only. This is especially true for performance, because even uninstrumented WebAssembly binaries can execute on average up to 50% slower than native code [Jangda et al. 2019]. Despite this, on benchmark sets one and two, FUZZM achieves 414 executions per second on average, which is only 37% slower compared with 663 native executions per second in AFL. We believe this is fast enough for practical fuzzing of WebAssembly binaries and respectable, given execution of the target program in a VM. Finally, our approach will also benefit from potential improvements to the young Wasmtime VM in the future.

Besides the program execution in a VM, other sources of slowdown in FUZZM can come from the applied binary instrumentation. To evaluate the runtime overhead of the added code, we run the benchmark programs with the test inputs used for RQ2, and compare the runtime of the original, uninstrumented binaries against the runtime when the binaries were instrumented. The results are shown in the right part of Table 6.3. On average over 25 program executions, the coverage instrumentation imposes a runtime overhead of 1.46x over the uninstrumented binary. We will detail the overhead of the canaries in Section 6.5.6. The overhead of the coverage instrumentation is generally higher than for the canaries, because for every branch in the program it

Table 6.3: Robustness and runtime overhead of instrumented binaries (mean over 25 repetitions, 95% confidence intervals).

#	Benchmark	Test Inputs	Execution Time, Uninstrumented	Execution Time, Relative to Uninstrumented Binary				
				Coverage	Stack Canaries	Heap Canaries	All Canaries	Cov. + Can.
1	`abc2mtex`	30	815 ms	1.38 ± 0.03	1.02 ± 0.01	1.02 ± 0.01	1.06 ± 0.01	1.38 ± 0.04
2	`flac`	10	2,449 ms	1.42 ± 0.01	1.02 ± 0.01	1.00 ± 0.01	1.02 ± 0.01	1.48 ± 0.02
3	`jbig2dec`	28	4,742 ms	2.05 ± 0.01	1.22 ± 0.01	1.00 ± 0.00	1.22 ± 0.01	2.24 ± 0.01
4	`libpng`	10	3,480 ms	1.57 ± 0.02	1.03 ± 0.01	1.00 ± 0.01	1.02 ± 0.01	1.58 ± 0.02
5	`libtiff`	10	899 ms	1.30 ± 0.03	1.13 ± 0.01	1.11 ± 0.01	1.14 ± 0.01	1.40 ± 0.03
6	`openjpeg`	10	4,750 ms	1.77 ± 0.02	1.05 ± 0.01	1.00 ± 0.01	1.06 ± 0.01	1.84 ± 0.02
7	`pdfresurrect`	10	2,894 ms	1.16 ± 0.02	1.06 ± 0.01	1.31 ± 0.01	1.35 ± 0.01	1.53 ± 0.02
8	`base64`	10	256 ms	1.32 ± 0.10	1.02 ± 0.02	0.99 ± 0.01	1.03 ± 0.02	1.31 ± 0.09
9	`md5sum`	10	272 ms	1.33 ± 0.09	1.03 ± 0.03	1.00 ± 0.01	1.05 ± 0.02	1.30 ± 0.09
10	`uniq`	10	260 ms	1.34 ± 0.09	1.04 ± 0.03	1.02 ± 0.01	1.11 ± 0.09	1.39 ± 0.10
Average			2,082 ms	1.46 ± 0.04	1.06 ± 0.02	1.05 ± 0.01	1.11 ± 0.02	1.54 ± 0.04

adds 13 instructions. More efficient implementations, e.g., by predicting some branches based on static analysis [Ben Khadra et al. 2020], could further reduce the overhead, which we leave for future work. The last column of Table 6.3 shows the combined overhead of both the canary instrumentation and the coverage instrumentation.

Summary: FUZZM performs hundreds of program executions per second, which is only 37% slower than native AFL, despite executing the program in a VM. The coverage instrumentation imposes an average overhead of 1.46x, which dominates the overall overhead imposed by FUZZM's instrumentation.

6.5.5 *RQ4: Effectiveness Against Exploitation*

In the previous research questions, we have analyzed FUZZM as an end-to-end WebAssembly fuzzer. The canary instrumentations from Section 6.3 are, however, also useful in a stand-alone setting, namely for catching memory errors in production binaries to prevent exploitation. To evaluate this scenario, we apply our canary instrumentation to the three previously described, vulnerable WebAssembly applications with proof-of-concept exploits in Section 3.4. The applications use WebAssembly in three different settings: on a website in the browser, on Node.js, and a command-line application for stand-alone WASI VMs. Since the canary instrumentation is platform-independent, we can harden binaries in all three settings. The proof-of-concept inputs exploit two buffer overflow vulnerabilities on the stack, and one buffer overflow on the heap that writes into allocator metadata (Table 3.2). We confirm that the uninstrumented, original WebAssembly binaries can be exploited, which causes cross-site scripting, executes code, and writes to an unintended file, respectively. Then, we successfully instrument all three binaries, without requiring access to the source code or their build process. When given correct and benign inputs, those three instrumented binaries work as before, but when passing the exploit inputs, all three examples are successfully terminated by the inserted canary checks.

Summary: The stack and heap canaries inserted by our binary-only instrumentation effectively hardens existing binaries and protects against previously demonstrated exploits.

6.5.6 *RQ5: Efficiency of the Inserted Canaries*

As demonstrated in the previous section, the inserted canaries can mitigate buffer overflow attacks when applied to existing binaries. For this usage scenario, it is essential that the canaries have only a minimal impact on performance. We evaluate their efficiency by running the benchmark programs with and without instrumentation on the inputs from RQ3. The results are in Table 6.3, which shows the execution times with different combinations of canaries relative to the execution time of the uninstrumented programs.

Both the stack and heap canary instrumentation only slightly impact performance, with an average execution time of 1.06x and 1.05x relative to the uninstrumented binary. The stack canary overhead is similar to efficient implementations of canaries for native binaries [Dang et al. 2015], which are employed by default in common compilers (Clang, GCC, MSVC). Some applications, e.g., *jbig2dec* with an execution time of 1.22x, are impacted more than others, e.g., *flac*, where the overhead is negligible. The relative cost of heap canaries depends on the number of memory allocations, especially small ones. While *pdfresurrect*, an analyzer of PDF files, stands out due to its frequent allocations, the overhead for the other applications is low or even too small to measure. The overhead with both canary instrumentations applied (column "All Canaries") is approximately the combination of both, imposing a moderate overhead of 1.11x.

Summary: The overhead imposed by the canary-based oracles is small (1.06x and 1.05x on average, respectively) and comparable to canary implementations for other binary formats, which is encouraging for their use to harden production binaries.

6.6 SUMMARY

WebAssembly programs are becoming more and more prevalent, which increases the need for techniques that can uncover security problems. This chapter presents FUZZM, the first binary-only greybox fuzzer for WebAssembly. The approach combines canary-based binary instrumentation to detect overflows and underflows on the stack and the heap, an efficient coverage instrumentation, a WebAssembly VM running the program, and the input generation algorithm of native AFL. FUZZM

works directly on production binaries, without requiring access to the source code. We show that FUZZM finds a substantial amount crashes in real-world WebAssembly binaries, while being efficient enough to perform hundreds of executions per second, even though WebAssembly is a slower, non-native language. Besides as oracles for fuzzing, the canaries also serve as a stand-alone binary hardening technique to prevent exploitation of vulnerable binaries in production. In this scenario, the approach prevents our previously discussed exploits while imposing low overhead. Overall, our work is an important step toward securing increasingly popular WebAssembly programs against exploitation of memory vulnerabilities.

7 SNOWWHITE: NEURAL RECOVERY OF HIGH-LEVEL TYPES

As a low-level code format, WebAssembly binaries are hard to understand. This is especially true for web developers, which only had to deal with high-level JavaScript on the client side so far. Chapter 4 has also shown that many WebAssembly binaries on the web lack debug information, which exacerbates the problem. Developers thus need help with reverse engineering WebAssembly binaries.

Recovering high-level function types is an important part of reverse engineering. One method to recover types is data-flow analysis, but it is complex to implement and may require manual heuristics when logical constraints fall short. In contrast, this chapter presents SnowWhite, a machine learning-based approach for recovering precise, high-level parameter and return types for WebAssembly functions. It improves over prior work on learning-based type recovery by representing the types-to-predict in an *expressive type language*, which can describe a large number of complex types, instead of the fixed, and usually small type vocabulary used previously. Thus, recovery of a single type is no longer a classification task but sequence prediction, for which we build on the success of neural sequence-to-sequence models. We evaluate SnowWhite on a new, large-scale dataset of 6.3 million type samples extracted from 300,905 WebAssembly object files. The results show the type language is expressive, precisely describing 1,225 types instead the 7 to 35 types considered in previous learning-based approaches. Despite this expressiveness, our type recovery has high accuracy, exactly matching 44.5% (75.2%) of all parameter types and 57.7% (80.5%) of all return types within the top-1 (top-5) predictions.

This chapter shares large parts of its material with the corresponding publication [Lehmann and Pradel 2022]. The author of this dissertation is also the main author of that paper and did all of the implementation, data collection, evaluation, and the majority of the writing.

7.1 MOTIVATION AND CONTRIBUTIONS

Because WebAssembly programs are in a low-level, binary format, understanding them is all but trivial. At the same time, due to its increasing popularity and the multitude of application domains, there is ample demand for reverse engineering WebAssembly code. For example, a developer integrating a third-party WebAssembly module may want to better understand its exported functions, or audit it to prevent supply-chain attacks [Zimmermann et al. 2019]. Security experts may need to analyze malicious WebAssembly binaries, which, e.g., try to escape from the browser sandbox [Plaskett et al. 2018; Silvanovich 2018], or perform unsolicited cryptocurrency mining [Musch et al. 2019b]. Finally, good reverse engineering tools are even more important when malicious JavaScript code is intentionally hidden inside or compiled to WebAssembly for obfuscation [Romano, Lehmann, et al. 2022].

An important first step toward understanding a WebAssembly binary is to understand the type signatures of its functions. Because types are highly relevant for understanding low-level code, they are similarly targeted by existing reverse engineering tools for native binaries [Caballero and Lin 2016; Chua et al. 2017; Pei et al. 2021]. Developer studies also show that static types help to understand code [Hanenberg et al. 2014; Mayer et al. 2012].

Unfortunately, the types available in a WebAssembly binary are only of very limited help. WebAssembly code *is* statically typed, but there are only four low-level primitive types for numbers. To a reverse engineer, those are not very informative. E.g., an `i32` could be a signed or unsigned integer in the application domain, a size in bytes, an array, a pointer to a struct, or one of many other source types. Thus, in addition to WebAssembly's four low-level types, it would be beneficial to recover precise, high-level types similar to those used in the programming language the binary was compiled from.

One avenue to recover high-level types is based on "classical" data-flow analysis or type inference, which collects constraints based on how values are used in the program [Caballero and Lin 2016]. However, this is complex to implement and often builds on heavy analysis frameworks, such as *BAP* or *CodeSurfer* [Lee et al. 2011; Noonan et al. 2016]. Supporting WebAssembly, especially with its slightly unusual stack machine (Section 2.2), would be a non-trivial undertaking. More fundamentally, not all information can be expressed as logical constraints,

so manual heuristics are often still employed to present simplified, intuitive types in the end [Noonan et al. 2016].

In contrast, we pursue a *data-driven, learning-based approach*, in line with general guidelines on when neural software analysis is beneficial [Pradel and Chandra 2021]. For example, some sequences of instructions may only provide a statistical hint, but no guarantee about the type of a parameter. Similarly, there is often no single correct solution in type recovery, e.g., both `class` and `struct` might be valid in terms of constraints, yet convey different intuitions to a human reverse engineer. Finally, there is, at least in principle, ample data to learn from, as arbitrary code can be compiled to WebAssembly and debug information can provide type labels for supervised training.

Various learning-based approach for predicting types in other languages have been proposed in recent years. They consider either binaries for native architectures [Chua et al. 2017; J. He et al. 2018; A. Maier et al. 2019] or dynamically typed source languages, e.g., Python [Allamanis, Barr, et al. 2020; Pradel, Gousios, et al. 2020] and JavaScript [Hellendoorn, Bird, et al. 2018; Malik et al. 2019; Raychev et al. 2015]. These approaches explore different input representations, e.g., token sequences [Hellendoorn, Bird, et al. 2018], data flow graphs [Allamanis, Barr, et al. 2020], and natural language associated with code [Malik et al. 2019], and different model architectures and ways of training them, e.g., recurrent neural networks [Pradel, Gousios, et al. 2020], transformers [Ahmed et al. 2021], graph neural networks [Allamanis, Barr, et al. 2020], and unsupervised pre-training [Pei et al. 2021].

Unfortunately, current learning-based approaches suffer from two key limitations. First, on the practical side, no existing approach predicts high-level types for WebAssembly. Second and more fundamentally, prior work focuses either on how to represent the code for which types are predicted, or on what machine learning model is most suitable for this task. In contrast, another important aspect of type prediction is currently understudied: *How to represent the to-be-predicted types themselves?* Almost all existing papers, with one noteworthy exception [Allamanis, Barr, et al. 2020], formulate type prediction as a classification problem. As classification scales poorly to a large number of classes, this formulation typically implies a small number of types to choose from. For example, recent binary-level type prediction mod-

els [Chua et al. 2017; J. He et al. 2018; A. Maier et al. 2019; Pei et al. 2021] support only 7, 11, 17, and 35 types, respectively.

This chapter addresses both the lack of a type prediction approach for WebAssembly in particular, and the limitations of previous learning-based approaches to represent types for binary type recovery in general. We present SnowWhite,[1] a learning-based approach for predicting high-level function types for a given WebAssembly binary. The core technical contribution is using an expressive language for describing the types SnowWhite can predict. The language supports primitive types, named types, aggregates, such as pointers, arrays, and enums, as well recursive combinations of the above. Because different source languages compile to WebAssembly (Chapter 4), the type language is derived from the DWARF debugging format [DWARF 5 Standard], which is supported by several compilers for different source languages.

Given the type language, SnowWhite trains a model to predict types as a sequence of tokens. That is, we formulate the type prediction problem as a sequence prediction, and not a classification task. An important advantage of sequence prediction is that we do not have to artificially limit the number of types the model can choose from, but instead support (at least in principle) infinitely many types.

To train and evaluate SnowWhite, we also gather the first large-scale dataset of WebAssembly binaries with debugging information. Based on the DWARF information provided by the compiler, we can associate each WebAssembly function with its return type and parameter types. Our dataset consists of 6.3 million types in 300,905 WebAssembly object files compiled from over 4,000 C and C++ Ubuntu source code packages. The dataset is two orders of magnitude larger than datasets considered in previous work on WebAssembly compiled from source code, which consider only tens of programs [Jangda et al. 2019]. Beyond this work, we envision the dataset to provide a basis for other learning-based work on WebAssembly, e.g., to predict the names of program elements or to decompile WebAssembly code back to source code.

Our evaluation shows that the type language expresses 1,225 different types, i.e., many more than prior work on binary type prediction [Chua et al. 2017; J. He et al. 2018; A. Maier et al. 2019; Pei et al.

1 In the eponymous fairy tale, the protagonist discovers helpful dwarfs. In our approach, we want to recover types from the DWARF debugging information format.

2021], while also offering a more uniform type distribution. Despite this expressiveness, the type prediction model exactly predicts 44.5% (75.2%) of all parameter types and 57.7% (80.5%) of all return types within the top-1 (top-5) predictions, clearly outperforming a statistical baseline approach based on the data distribution.

CONTRIBUTIONS In summary, this chapter contributes:

- Addressing the practical problem of predicting high-level types of WebAssembly functions, which is important for understanding WebAssembly binaries.
- A type language for binary type recovery that is much more expressive than the small number of labels used in prior learning-based approaches (Section 7.3).
- Formulating the type prediction as a sequence prediction task, which facilitates accurate predictions across a large number of types to choose from (Section 7.4).
- Creating and sharing the so-far largest dataset of WebAssembly binaries with debug information (Section 7.5).

The collected dataset, the source code, and our results are publicly available at `https://github.com/sola-st/wasm-type-prediction`.

7.2 OVERVIEW

This section gives an overview of SnowWhite, starting with a motivating example in Listing 7.1, then defines the problem more precisely, and presents the main components of the approach.

MOTIVATING EXAMPLE Listing 7.1a shows the source code of a function from `libglpk`, a linear-programming library written in C, contained in the Ubuntu repositories. The function has one parameter, which is declared as an array of `doubles` (line 1). If it is non-`NULL`, the function reads two values from the array, and otherwise uses defaults (lines 4–10).

Compiling this function to WebAssembly yields the code in Listing 7.1b. The comments are for illustration only and not part of the actual binary. On the right, Listing 7.1c shows the type of the parameter as represented in the DWARF debugging format [DWARF 5 Standard]. DWARF data is embedded in binaries when compiling with debug information (`-g`), but not present in stripped binaries a reverse engineer

```c
void amd_control(double Control[]) {
    double alpha;
    int aggressive;
    if (Control != (double *) NULL) {
        alpha = Control[DENSE];
        aggressive = Control[AGGRESSIVE] != 0;
    } else {
        alpha = DEFAULT_DENSE;
        aggressive = DEFAULT_AGGRESSIVE;
    }
    if (alpha < 0) {
        printf("no rows treated as dense");
    }
    [...]
}
```

(a) Excerpt of the C source code of a function in `libglpk`.

```
0073: ;; Offset of the start of the code section; first function @code+3:
0076: (func $1 (param $0 i32) (result))  ;; Low-level function type.
        [...]
0091:   block
0093:     local.get $0      ;; Push the parameter on the stack.
0095:     br_if 0           ;; Branch if non-zero.
          [...]
00e9:   end                 ;; The branch continues here.
00ea:   local.get $0
00ec:   f64.load offset=8   ;; Load 64-bit float from memory at $0 + 8.
        [...]
00f7:   local.get $0
00f9:   f64.load offset=0   ;; Load 64-bit float from memory at $0.
```

(b) The function from (a) compiled to WebAssembly. Byte offsets into the binary shown on the left. The parameter has only a low-level `i32` type.

```
0026: DW_TAG_pointer_type  // DWARF entry for one particular pointer type.
        DW_AT_type @ 002b  // The pointee type references another entry.
002b: DW_TAG_base_type     // A primitive type, identified by:
        DW_AT_name: "double"         // - Its name in the source code.
        DW_AT_encoding: DW_ATE_float // - An encoding, here IEEE 754.
        DW_AT_byte_size: 8           // - Its size in bytes.
0033: DW_TAG_subprogram    // DWARF entry for a function.
        DW_AT_name: "amd_control"
        DW_AT_low_pc: 0x03 // Offset of function from start of code section.
0047:   DW_TAG_formal_parameter
          DW_AT_name: "Control"
          DW_AT_type @ 0026
```

(c) DWARF debugging information for (a).

```
pointer primitive float 64
```

(d) The high-level type to be predicted for the parameter of (a).

Listing 7.1: Motivating real-world example for recovering high-level types from binaries. The source code, compiled WebAssembly, DWARF information, and parameter type to predict for a function is shown.

typically encounters. It is a hierarchical binary format, with single-tag entries (**blue**) that have multiple attributes (**teal**), and potentially multiple other entries as children (e.g., the parameter in line 10 is a child of the function in line 7). As attributes can refer to other entries (lines 2, 12), the information forms a directed, possibly cyclic graph. In the example, the parameter entry refers to a `pointer` type entry (line 1), which in turn refers to its element type, a primitive 64-bit float type (line 3).

PROBLEM DEFINITION The goal of SNOWWHITE is to predict precise, high-level types from WebAssembly binaries. Because understanding functions and their types are valuable first steps to a reverse engineer understanding the functionality of a binary, we focus on function parameter and return types. More formally:

Definition 1. The *type prediction problem* is to find a mapping

$$f, e, t_{low} \rightarrow t_{high}$$

where

- f is the body of a given WebAssembly function with k parameters and zero or one return value,
- $e \in \{param_1, ..., param_k, return\}$ is the desired element from the function signature to predict a type for,
- $t_{low} \in \{\texttt{i32}, \texttt{i64}, \texttt{f32}, \texttt{f64}\}$ is the low-level WebAssembly type of e, already present in the binary, and
- $t_{high} \in \mathcal{L}_{types}$ is a type defined by a high-level type language $\mathcal{L}_{types}$.

SNOWWHITE predicts the type of each parameter and the return type separately. As WebAssembly is statically typed, the low-level type of each program element is known, which we exploit by providing it as an input to the approach. The main novelty of SNOWWHITE is how to represent the output t_{high} of the type prediction task. One option would be to predict one out of a fixed set of types, as done in prior work on binary type prediction [Chua et al. 2017; J. He et al. 2018; A. Maier et al. 2019; Pei et al. 2021]. For example, we could aim to predict simply that the function parameter in Listing 7.1 is a pointer. While providing a relatively easy prediction task, that approach misses many details relevant for understanding the functions in a binary. Another option would be to predict the full DWARF type (Listing 7.1c). However, full DWARF types contain various details that are irrelevant for a reverse engineer, making the prediction task unnecessarily hard.

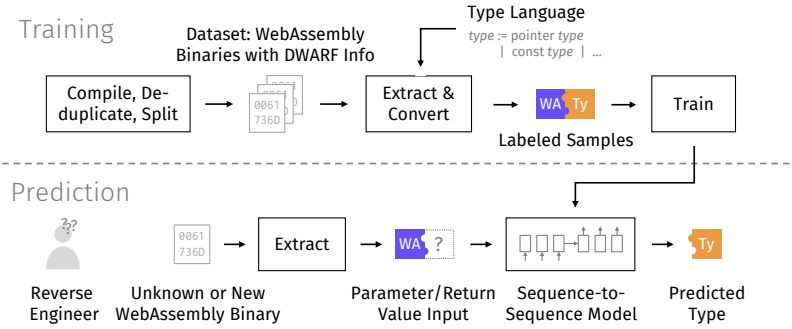


Figure 7.1: Overview of SNOWWHITE's components.

Instead of the above two extremes, SNOWWHITE predicts types that are sentences in a high-level type language. For example, the type in Listing 7.1d expresses the fact that the parameter is a pointer to a memory location that stores a primitive 64-bit float. Our high-level type language is derived from DWARF, and hence can express types across multiple source languages that are commonly compiled to WebAssembly, such as C and C++. In contrast to DWARF, the type language omits details that are not crucial to a reverse engineer but that would make prediction harder.

SNOWWHITE IN A NUTSHELL To address the type prediction problem, SNOWWHITE uses a neural sequence-to-sequence model and a large-scale dataset of WebAssembly binaries with DWARF type information. Figure 7.1 shows the two major phases of the approach: In the *training* phase, the sequence-to-sequence neural network model is trained from labeled data. As there exists no suitable large, real-world dataset of WebAssembly functions with high-level type information, we create a large-scale dataset by compiling 4,081 Ubuntu source packages to WebAssembly, resulting in more than 6.3 million labeled samples. In the *prediction* phase, a reverse engineer can query the trained model with previously unseen WebAssembly functions, to obtain a high-level type prediction for parameters and return values.

The following sections present the type language (Section 7.3), the neural prediction model (Section 7.4), and the collected dataset (Section 7.5) in more detail.

7.3 HIGH-LEVEL TYPE LANGUAGE

Prior work has explored different input representations and model architectures for binary type prediction [Chua et al. 2017; J. He et al. 2018; A. Maier et al. 2019; Pei et al. 2021]. Much less focus has been on the representation of types for the prediction task itself. One core contribution of this work is to describe the types to predict using a type language that includes precise information about primitive types, nested types, const and signed-ness, and type names. We refer to the language as $\mathcal{L}_{\text{SnowWhite}}$, or $\mathcal{L}_{\text{SW}}$ for short. Figure 7.2 gives a BNF grammar of the language. Table 7.1 compares our type language against those used in prior learning-based binary type prediction and against full DWARF information [DWARF 5 Standard].

7.3.1 *Type Language Structure*

There are two extremes in terms of how types can be represented: On the one end of the spectrum, types can be represented as a small, *fixed set* of choices. This is the case for prior work, as shown in the first four rows of Table 7.1. The $|\mathcal{L}|$ row shows the number of unique types as reported in the respective papers. Even though a type grammar is sometimes presented, this is only for illustration and the sets of types these grammars describe are all finite and small.

One virtue of a fixed set of types is simplicity, both in terms of data extraction and the model architecture, as classification tasks are simple to train and evaluate. The downside is that there is a mismatch between the fixed set of types and the infinite types in the source program (e.g., in C and C++), where more complex types can be built up from simple ones by composition. As such, the source types need to be heavily simplified to map to the target set, which loses information and equates many potentially different types.

On the other end of the spectrum stands the full type language of DWARF. Types therein are directed, possibly cyclic graphs, allowing DWARF to capture recursive types. A downside of this representation is that predicting graphs with neural networks is challenging; recent work we know of only encodes graphs, but does not predict them [Allamanis, Brockschmidt, et al. 2018; Z. Chen et al. 2021; Hellendoorn, Sutton, et al. 2020]. Full DWARF types also contain constructs that

Table 7.1: Comparing different type languages of learning-based binary type prediction approaches. ✓ means a feature is supported, ✗ means not. The 'Prim. Size' row shows whether the representation for primitive types captures the size. † in that row means that sizes are captured only via C type names, which are ambiguous, instead of an exact, unambiguous representation.

	Eklavya [Chua et al. 2017]	Debin [J. He et al. 2018]	TypeMiner [A. Maier et al. 2019]	StateFormer [Pei et al. 2021]	SnowWhite	Full DWARF
Type Structure	Fixed set	Fixed set	Fixed set	Fixed set	Sequence	Full graph
$\lvert\mathcal{L}\rvert$	7	17	11	35	∞	∞
Primitives:						
`int` / `char`	✓	✓	✓	✓	✓	✓
`bool`	✗	✓	✓	✗	✓	✓
`float`	✓	✗	✓	✓	✓	✓
`int` Sign	✗	✓	✗	✓	✓	✓
Prim. Size	✗	only `int`†	only `int`†	(✓)†	✓	✓
Aggregates:						
`struct`	✓	✓	✗	✓	✓	✓
`class`	✗	✗	✗	✗	✓	✓
`union`	✓	✓	✗	✓	✓	✓
`enum`	✓	✓	✗	✓	✓	✓
`array`	✗	✓	✓	✓	✓	✓
Pointee Type	✗	✗	`struct`, `char`, function	Single level	Recursive	Recursive
Fields	✗	✗	✗	✗	✗	✓
Other:						
`const`	✗	✗	✗	✗	✓	✓
Names	✗	✗	✗	✗	Top k	✓
`volatile` / `restrict`	✗	✗	✗	✗	✗	✓
Language-Specific Types	✗	✗	✗	✗	`class` vs. `struct`	✓

$$
\begin{aligned}
\mathit{type} := {} & \texttt{primitive}\ \mathit{primitive} && \text{(primitive types)}\\
\mid {} & \texttt{pointer}\ \mathit{type} \mid \texttt{array}\ \mathit{type} && \text{(pointers and arrays)}\\
\mid {} & \texttt{const}\ \mathit{type} && \text{(const-ness)}\\
\mid {} & \texttt{name}\ \mathit{name}\ \mathit{type} && \text{(nominal types and typedefs)}\\
\mid {} & \texttt{struct} \mid \texttt{class} \mid \texttt{union} \mid \texttt{enum} && \text{(aggregates)}\\
\mid {} & \texttt{function} && \text{(for function pointers)}\\
\mid {} & \texttt{unknown} && \text{(uninformative type)}\\
\mathit{primitive} := {} & \texttt{bool} && \text{(booleans)}\\
\mid {} & \texttt{int}\ \mathit{bits}_{\texttt{int}} \mid \texttt{uint}\ \mathit{bits}_{\texttt{int}} && \text{(integers)}\\
\mid {} & \texttt{float}\ \mathit{bits}_{\texttt{float}} \mid \texttt{complex} && \text{(floating-point)}\\
\mid {} & \texttt{cchar} \mid \texttt{wchar}\ \mathit{bits}_{\texttt{wchar}} && \text{(characters)}\\
\mathit{name} \in {} & \{\texttt{"size\_t"}, \texttt{"FILE"}, \texttt{"string"}, \ldots\} && \text{(names)}
\end{aligned}
$$

$$\mathit{bits}_{\texttt{int}} \in \{8,16,32,64\},\ \mathit{bits}_{\texttt{float}} \in \{32,64,128\},\ \mathit{bits}_{\texttt{wchar}} \in \{16,32\}$$

Figure 7.2: Grammar of the high-level type language $\mathcal{L}_{\text{SW}}$.

are unlikely to be recoverable from binaries, e.g., optimization hints, language-specific constructs, and domain specific names.

With our type language, we aim to strike a balance between those two extremes. The grammar in Figure 7.2 produces types that are represented as a *linear sequence of type tokens*. The set of all possible types is infinite, and while we do not represent the fields of aggregates, we do allow nested types for pointers, arrays, `const` and names. Besides describing a larger set of types than prior learning-based type prediction, the fact that each type in our language is a sequence of type tokens allows us to formulate type prediction as a sequence prediction task.

To produce a type sequence in our language from the DWARF information in a binary, we recursively traverse the DWARF type graph, pattern match on the type constructor (e.g., `DW_TAG_pointer_type` in Listing 7.1c) and convert it to a type constructor of Figure 7.2 or remove it. Figures 7.1c and d show an example of a DWARF type represented in our language. We break cycles to prevent generating infinite type sequences. We now describe the features of our grammar in more detail.

7.3.2 *Primitive Types*

Primitive types in binaries, as represented in DWARF, are surprisingly complex. All type prediction approaches have some representation of integers, but they differ in their precision and how they handle other primitives, which can lead to ambiguous or incorrect type labels (rows 3 to 7 of Table 7.1).

First, our language has an explicit boolean type. Even though on a machine-level, boolean values are represented as integers, we believe the type distinction is important and instructive for reverse engineers, e.g., `bool` is more telling than just `int`. Eklavya and StateFormer do not distinguish the two and map booleans to integers instead.

Second, we support floats of different width (single, double, and quad precision) and the C built-in `complex` type. In contrast, floating point types are not handled by Debin at all, and Eklavya and TypeMiner do not capture their width.

Third, our language represents integer types precisely. Eklavya does not distinguish different sizes or signed-ness of integers, e.g., `short` and `long long` are mapped to the same type. A better, but still naive approach is to represent integer types by their name in the source code, e.g., `int` or `unsigned long`. This is problematic, because the relation between the source code name and the machine representation of integers is both ambiguous and not injective. The mapping of name to representation is not injective because different names can map to the same type, e.g., `short`, `short int`, and `signed short int` (and even permutations thereof) are all the same type in C. In other languages, the same type is called differently again, e.g., `i16` in Rust. Using the source code name for identification would thus introduce distinct classes for what is really the same type. Additionally, the mapping from name to representation is ambiguous, e.g., `long` can be both 32 bits wide or 64 bits wide, depending on the compiler's data model (ILP32 vs. LP64). That is, a reverse engineer cannot tell the bit-width from `long` alone. Thus, unlike prior work, which uses C and C++ type names to identify integers, we choose an unambiguous and language-independent representation based on bit-width and signed-ness.

Finally, our type language models character types precisely. In C and C++, `signed` and `unsigned char` are just 8-bit integers, and encoded in our approach as such. A "plain" `char` in C is different from both, and used only for character data, not in arithmetic operations. It commonly

appears in string-handling functions. We represent it as `cchar`. *Wide char* types, e.g., `wchar16_t` in C++, are used for 16 and 32-bit unicode characters and modeled in our language as well. Prior approaches do not distinguish between those types.

In summary, our encoding normalizes all primitive types appearing in the dataset to an unambiguous representation of 16 types. It does not conflate different types with each other, and assigns exactly one type name per unique underlying primitive type. Notably absent are high-level data structures such as strings, lists, or dictionaries, as they are not built-in in systems languages, unlike, e.g., in Python or JavaScript.

7.3.3 *Pointers and Aggregate Types*

More complex aggregate types can be built up from constituents, e.g., in array, pointer, or struct types. Our type language supports both arrays and pointers (unlike Eklavya, which maps the former to the latter) and also captures the nested type of their elements and pointee, respectively. This is unlike Eklavya and Debin, where all pointers are `*void`, regardless of what they actually point to. TypeMiner and StateFormer discern certain classes of pointers from each other, e.g., pointers-to-structs from function pointers, but are not recursive, and as such cannot represent, e.g., the C type `*char[]` (which would be `array pointer char` in our type language).

We do not capture individual fields of aggregate types like structs and unions, which is where we lose information compared with full DWARF types. As not every field of a struct or union is used in a given function, prediction of field types is a challenge left for future work. To model function pointers, our language includes a `function` constructor.

7.3.4 *Type Attributes and Language-Specific Types*

DWARF information includes type attributes, e.g., `const`. We include `const` as a type constructor into our language. This allows a reverse engineer to discern between a pointer to constant data (`pointer const` *t*), a pointer whose value is constant (`const pointer` *t*), and a mutable `pointer` *t*, and thus gives useful information about the invariants of the source program. This is similar to the constraint-based Retypd [Noonan et al. 2016], but unlike all prior learning-based approaches. DWARF types also contain the attributes `volatile` and `restrict`, but since those

are optimization hints for the compiler and likely hard to recover, we remove them when traversing the input type.

We also aim to recover some language specific types, notably the distinction between a `class` and a `struct`. We believe this is useful to reverse engineers because classes point to object-oriented programming, frequently have methods, and implicitly identify the source language as C++, whereas structs are idiomatic for plain old data. Neither learning-based nor constrained-based approaches so far aimed to recover this distinction, instead equating classes and structs into a single concept. The distinction between C++ references and C pointers is less instructive and more difficult to recover, so we map those to a single `pointer` constructor.

7.3.5 *Unknown and Unspecified Types*

In some cases, the type information in DWARF can be incomplete. One common cause are forward declarations in C and C++, such as

```
struct name;
```

Even though a forward declared type cannot be used directly in parameters or return types, they frequently appear behind pointers. Similar are `void` pointers, e.g., in the return type of `malloc` and other generic functions. A third case is the C++ `nullptr` expression, which has an unnameable type `decltype(nullptr)`. In all three cases the element type is unknown, but we still know that this is a generic pointer. We thus feature an `unknown` constructor, similar to an uninformative type $\top$ in other type systems, and encode all three mentioned cases as `pointer unknown`.

7.3.6 *Names and Typedefs*

The type language so far is precise, but still purely structural and fairly low-level, thus capturing little "human intuition" about the high-level semantics of types. Type names can convey such semantics and are included in DWARF, so we would like to recover them (at least partially) in our prediction task. This sets us apart from prior work on learning-based type prediction from binaries, that never went beyond primitive types and simple aggregates. Constraint-based approaches also either ignore type names fully, or rely on manually written rules for some well-known functions [Noonan et al. 2016]. There are several challenges when representing names in types.

Names in DWARF types appear in two places, namely in typedefs and in named aggregate type definitions, such as `struct`, `class`, or `union`. In both cases, names are used to ascribe meaning, but only the latter introduces a nominal type with strict typing discipline. Typedefs are merely aliases, i.e., can be freely exchanged with the underlying type. Should we thus remove typedefs? We argue not, because their names still convey useful information, e.g., the type `size_t` is more instructive than just integer. We thus map names in typedefs and names in datatype definitions to a single `name` constructor that also contains the underlying structural type. E.g., `name "size_t" uint 32` captures both the name and the underlying structural type.

Next, what to do with nested names? Those can appear either because of the previous conversion, or simply because of repeated typedefs. Consider the frequent example of a typedef in conjunction with a struct definition:

```
typedef struct sname { /* fields... */ } tname;
```

Which name should be used for the type: `tname` or `sname`? We solve this by filtering the names (see below), and then keeping only the outermost (i.e., first) `name` constructor in a type sequence, as this is most likely the user-visible name. That is, the example would be represented as `name "tname" struct` in our type language.

Finally, out of all names in the DWARF information, we keep only a subset in our language. First, because of the age and low-level nature of C and C++, there is less of a shared type vocabulary than there is, e.g., in Python. Many programs define their own data structures and even aliases for primitives. We do not want to predict such domain or project specific names. Second, very infrequent type names cannot be realistically predicted, and unlike in type-prediction approaches for high-level languages, the input binary contains no natural language data that a copy mechanism [Gu et al. 2016] could use to generate type names from the input.

We thus extract a set of k common names (from typedefs and named datatypes), where we define common as all names that appear at least once in 1% of all packages in our dataset. Additionally, we filter out names that start with an underscore (as those are likely internal) or where the typename is equal to what we already capture in our primitive type representation, such as the name `uint32_t`. For generic types, e.g., the C++ template `std::vector<int>`, type arguments are included

in the name. An alternative would be to abstract over the argument, but for simplicity and to retain more information, we keep the name as-is.

7.3.7 *Type Language Variants*

To evaluate the effect of different type languages on the type distribution and the accuracy of type prediction, we also define two variants of our language. We call the type language described so far $\mathcal{L}_{\text{SnowWhite}}$ (or $\mathcal{L}_{\text{SW}}$ for short).

First, we define a variant of $\mathcal{L}_{\text{SW}}$ that contains all type names in the dataset. That is, it has the same grammar as in Figure 7.2, and performs the same mapping of typedef and datatype names to a `name` constructor, and keeps only the outermost name as described in Section 7.3.6, but it does not restrict the set of names based on the number of packages they appear in. Consequently, many more types are named in this language.

Second, we define a simplified version of $\mathcal{L}_{\text{SW}}$, which removes the following constructors from Figure 7.2: `const`, `class`, and `name`. Consequently, types in this language are never named, classes are represented as `structs`, and `const` constructors are flattened away. This makes the language considerably simpler and closer to the languages in prior binary type prediction work.

We discuss the effect of those variants on prediction in Section 7.6.

7.4 TYPE PREDICTION MODEL

We now present how SnowWhite predicts high-level types for WebAssembly functions with a neural model. Our work is the first to formulate prediction of a single type as a sequence prediction task p

$$p : (i_1, ..., i_m) \rightarrow (t_1, ..., t_n)$$

where $(i_1, ..., i_m)$ are instruction tokens extracted from the WebAssembly function body (Section 7.4.1) and $(t_1, ..., t_n)$ are type tokens as described by our type language. We address the prediction task p with a state-of-the-art, sequence-to-sequence neural network model (Section 7.4.2) trained in a supervised manner to minimize the difference between the predicted type and the known actual type.

7.4.1 *WebAssembly Input Representation*

Section 7.3 covered the type language and its type tokens t_i, but we also need to represent the WebAssembly input as tokens i_i.

INSTRUCTION TOKENS To predict the parameter types and the return type of a function f, SNOWWHITE creates a sequence $(i_1, ..., i_m)$ of instruction tokens from f's representation in the WebAssembly binary. First, we disassemble the binary into a sequence of instructions for each function. In contrast to native binary formats, e.g., x86, static disassembly is well-specified and robust for WebAssembly (Section 2.1). Then, we represent each instruction as in the WebAssembly text format (e.g., `i32.const 42` for the instruction that pushes 42 on the stack), and delimit individual instructions by ';'. We omit some static arguments of instructions that are likely unhelpful and unnecessarily increase the number of tokens, namely alignment hints for memory accesses and the function index of the callee in `call` instructions. For predicting the type of a parameter p, we replace the index of p in `local.get`, `local.set`, and `local.tee` instructions with the special token `<param>`, to indicate to the model which parameter to focus on. Finally, we also add the low-level type (e.g., `i32`) of the parameter or return value to predict at the beginning of the sequence, delimited by a `<begin>` token.

HANDLING LONG FUNCTIONS Binary code can have very long functions, which results in even longer token sequences. In our dataset, 10% of the functions have more than 1000 tokens and 1% even more than 5500 tokens. Because recurrent neural networks have trouble handling long sequences [Pascanu et al. 2013], and to facilitate efficient training in mini-batches, sequences are truncated and padded to a fixed length (here: 500 tokens) during training.

Prior learning-based approaches on code often completely filter out long samples from the dataset, both during training and evaluation [Chua et al. 2017; Hellendoorn, Bird, et al. 2018; Lacomis et al. 2019]. As removing long functions from the test data may unrealistically inflate accuracy numbers, we do not follow this practice.

Instead, SNOWWHITE extracts windows of instructions around instructions related to the to-be-predicted type. For predicting a parameter type, the approach extracts fixed-size windows around all instructions that use the parameter (`local.get`, `local.set`, and `local.tee`), and concatenates the windows, omitting the instructions in between.

For prediction of a return type, SnowWhite extracts all windows ending in a `return` instruction. Windows are delimited by a `<window>` token from each other.

EXAMPLE For illustration, consider the following sequence of i_i tokens, extracted for predicting a parameter type:

```
( 'i32', '<begin>',
  'i32.const', '42', ';', 'local.get', '<param>', ';', 'call', '<window>',
  'i32.add', ';', 'local.set', '<param>', ';', 'i32.eqz' )
```

It starts with the low-level type `i32` of the parameter to predict and then contains two windows of $w = 3$ instructions each, extracted around usages of the parameter. By default, we extract windows of size $w = 21$, i.e., 10 instructions to the left and right of parameter usages, and 20 instructions before a return instruction.

TOKEN-LEVEL EMBEDDING Finally, the tokens, both for the WebAssembly code and the type language, need to be converted to real-valued vectors for the neural network. We jointly train embedding layers that map each individual token to a dense vector of dimension e, with one embedding for WebAssembly tokens and one for type tokens.

If we would naively embed all WebAssembly tokens, one issue is the very large number of unique, but infrequent tokens in code [Karampatsis et al. 2020]. In particular, our dataset contains more than $v = 427{,}000$ unique WebAssembly tokens. The majority of those are numbers in the instructions, such as memory offsets, or integer and floating point constants. Using a very large vocabulary is undesirable due to increasing the number of model parameters (and thus memory usage), so instead we first build up a subword model based on byte-pair encoding (BPE) [Sennrich et al. 2016] that re-tokenizes the input into only $v' \ll v$ subword tokens. This breaks down infrequent source tokens into multiple subword tokens, which are then embedded with a much smaller embedding matrix, at the cost of slightly increased sequence length. We employ subword tokenization both for WebAssembly and for the type language.

7.4.2 *Sequence-to-Sequence Model Architecture*

To address the sequence-to-sequence prediction task, we reuse state-of-the-art results from neural machine translation (NMT). As the model

architecture is standard, we keep this description short and refer to the available implementation and literature for details.

The model is queried separately for every parameter of a function and its return type, that is, only a single type (which is itself a sequence) is generated per prediction. For each type-to-predict, we present the model with a separate input sequence. For example, to predict the types of a function of two arguments, we would query the same model twice, but with slightly different inputs.

We train two separate models, one for parameter and one for return type prediction. We use the same configuration for both models, namely a bidirectional LSTM model with global attention [Bahdanau et al. 2015; Luong et al. 2015] and dropout for regularization [N. Srivastava et al. 2014], as implemented in the OpenNMT framework.[2] The network's weights are optimized by standard backpropagation through time gradient descent with the Adam optimizer [Kingma and Ba 2015]. As an alternative sequence-to-sequence architecture, we also explored Transformers [Vaswani et al. 2017], but did not find it improving accuracy, so we select the computationally much cheaper LSTM model. More experimentation with other model architectures is orthogonal to our work; there is ample work on alternative architectures and representations of code [Allamanis, Brockschmidt, et al. 2018; Alon, Zilberstein, et al. 2019; Feng et al. 2020; Guo et al. 2021; Hellendoorn, Sutton, et al. 2020].

As hyperparameters, we choose after experimentation: $h = 512$ as the dimension of the hidden vectors, $l_e = 2$ two layers for the encoder and $l_d = 1$ a single layer for the decoder, $lr = 0.001$ as the initial learning rate and default momentums for Adam, $d = 0.2$ as the dropout rate, $e = 100$ as the dimension of the embeddings, and $v' = 500$ as the subword vocabulary size. The models have about 5.5 million learnable parameters in total.

7.5 DATASET

Training the neural models in SnowWhite for predicting types requires a dataset (i) from a diverse set of source programs, (ii) compiled to WebAssembly binaries, (iii) including the appropriate DWARF infor-

2 [Klein et al. 2017], https://github.com/OpenNMT/OpenNMT-py

mation, and (iv) resulting in an overall dataset size that is conductive to training a deep neural network.

Unfortunately, the datasets we have collected for earlier projects in this dissertation so far, do not fulfill all those requirements. The programs used in the evaluations of Chapters 3, 5, and 6 are relatively small, in the order of tens of programs. While the WASMBENCH dataset from Chapter 4 is considerably larger, with 8,400 WebAssembly binaries, it contains neither the required source code nor DWARF debug information. Similar caveats apply also to WebAssembly datasets used in other work [Haas et al. 2017; Jangda et al. 2019].

We thus collect our own large-scale dataset, which comprises 6.3 million samples of WebAssembly code and types, extracted from 300,905 object files, which were compiled from 4,081 C and C++ Ubuntu packages. Beyond type prediction, we envision the dataset to also serve other purposes, e.g., for work on recovering names from stripped binaries, decompilation, or finding compilation issues.

COMPILING TO WEBASSEMBLY BINARIES We start from all 70,065 source packages in the Ubuntu 18.04 repositories. Filtering out Linux kernel modules, which are unlikely to be compilable to WebAssembly, duplicates of applications for different locales, and fonts, 61,261 packages remain. We download their source code and keep all packages with at least one C or C++ source file. To compile the packages to WebAssembly, we modify the build scripts to use Emscripten, which is based on LLVM and the currently most popular compiler for WebAssembly. We add the `-g` flag to add debugging information in the DWARF format, but leave all other compilation options unchanged. In particular, all packages are compiled with their original optimization level (e.g., `-O2` or `-O3`), reflecting the options used for realistic binaries found in the wild and which a reverse engineer would encounter in practice. In total, 4,081 packages can be partially built to at least one object file, producing a total of 300,905 object files with WebAssembly code and DWARF information.[3]

DEDUPLICATION Following the advice by Allamanis [2019], we deduplicate the dataset to avoid artificially inflating our results. One option is to deduplicate individual functions or type samples. However,

3 Final linking frequently fails because Emscripten uses `musl` libc instead of `glibc`. As pre-linking object files still contain WebAssembly and DWARF information, those can still be used to train and evaluate our approach.

Allamanis [2019] notes that in some tasks, duplication is part of the true data distribution. Functions are frequently duplicated across different binaries when they stem from the same statically linked library, which is the case in WebAssembly. Thus, instead of deduplicating on the level of functions or samples, we deduplicate at the level of binaries. As a first step, we remove exact duplicates of binaries, identified by hashing the full file contents.

To also remove near-duplicates, e.g., because of included strings like build time in the binary, we also compute an approximate signature of each binary. The signature takes only the function bodies into account, where each function gets a hash based on its *abstracted* instructions. The abstraction removes immediate arguments from the instructions, i.e., `local.get $0` is mapped to `local.get`, or `i32.load offset=8` to just `i32.load`. The function hashes are then concatenated (i.e., function order is taken into account) and the result is hashed again to obtain an approximate signature for the whole binary. Out of multiple binaries with the same signature, only one is finally retained in the dataset.

Overall, deduplication reduces the dataset from 3.8 billion instructions and 31 million functions in 300,905 object files, down to 866 million instructions and 7.9 million functions in 46,856 object files. Even after this strict deduplication, our dataset is much larger than prior binary type prediction datasets: It is 1.8 times the size, in terms of number of instructions, of the four-architecture dataset in [Pei et al. 2021], 21× of [Chua et al. 2017], 32× of [J. He et al. 2018], and 1,059× of [A. Maier et al. 2019]. With an average function length of 109 instructions, we also believe our dataset is representative of real-world code, whereas an average length of ≈ 8 in [Pei et al. 2021] might indicate a dataset biased towards very short functions. Our deduplicated set comprises 21 GiB of WebAssembly binaries. This is 3.2× the size of the previously largest WebAssembly corpus, our WasmBench dataset (Chapter 4), which additionally lacks debug information.

MATCHING WEBASSEMBLY TO DWARF AND FILTERING To train SnowWhite in a supervised manner and as a ground truth for the evaluation, we associate WebAssembly functions with DWARF type information about the parameters and the return value. Function bodies are in the `code` section of a WebAssembly binary, whereas debug information is split over several custom sections `.debug_info`, `.debug_str`, etc. We match each WebAssembly function with the corresponding DWARF

information via the function's offset in the binary. This can be seen in the `DW_AT_low_pc` attribute of Listing 7.1c, which identifies the function by its offset from the start of the code section (Listing 7.1b).

The number of parameters of a function and also whether a function has a return value, may differ between the source code and the compiled binary, e.g., due to optimizations. If the number of function parameters in the DWARF information and the number of parameters in the WebAssembly bytecode is the same, then we extract one parameter type sample for each parameter. Likewise, if a function has a non-`void` return type in DWARF and returns a value in WebAssembly, then we extract a return type sample. Because of this, we do not extract type samples for 6% of the 7.9 million functions in the deduplicated dataset, which we believe does not introduce a significant bias. Finally, to avoid that one Ubuntu package with many samples biases our dataset, we limit the number of samples per package to at most the number of samples in the second most frequent package.

The final dataset after all deduplication and filtering comprises 5.5 million parameter type samples and 796 thousand return type samples. The lower number of return type samples can be attributed to many C and C++ functions returning `void`, where no type needs to be predicted.

SPLITTING THE DATASET We split the dataset into three portions: one for training, one for early stopping and evaluating hyperparameters, and a held-out test set. Randomly assigning each function or type sample to one of the three portions could cause (i) functions from the same binaries, or (ii) binaries from the same Ubuntu package ending up in two different portions of the dataset. Issue (i) is definitely unrealistic compared with the usage scenario of our approach, as the reverse engineer encounters a previously unseen binary, and (ii) means information from related binaries can leak from test to training data. To avoid both issues, we hence split the dataset by original Ubuntu packages. Since the total number of samples in our dataset is in the order of millions, the validation and test sets can be a relatively small portion of the overall dataset [Amari et al. 1997]. We choose 96% of the packages for training, and 2% for validation and testing, respectively.

7.6 EVALUATION

We evaluate SNOWWHITE on the previously described dataset. In Section 7.6.2, we focus on the expressiveness of our type language and the resulting distribution of realized types. We show that the default variant of our language distinguishes 1,225 unique types and provides a more uniform type distribution than the small set of types predicted by prior learning-based approaches. In Section 7.6.3, we evaluate the accuracy of the type prediction model. We show that SNOWWHITE predicts 44.5% (75.2%) of all parameter types and 57.7% (80.5%) of all return types exactly within the top-1 (top-5) predictions. Finally, Section 7.6.4 qualitatively discusses the strengths and weaknesses of our approach with some representative examples of predicted and ground truth types.

7.6.1 *Implementation, Setup, and Runtime*

Our implementation, the dataset, and all scripts required to reproduce our work are publicly available at `https://github.com/sola-st/wasm-type-prediction`. The implementation consists of about 500 lines of Python and Bash for gathering the dataset, about 4,000 lines of Rust for extracting and processing DWARF types and WebAssembly code, and about 1,700 lines of Python for preparing the data and the neural model. We use the *gimli* and *wasmparser* libraries for parsing DWARF and WebAssembly, respectively.[4] The neural sequence-to-sequence model is built on top of *OpenNMT-py* for the neural model and *SentencePiece* for the subword tokenization.[5]

We run all experiments on a server with with two Intel Xeon 12-core 24-thread CPUs running at 2.2 GHz, using 256 GiB of system memory, and Ubuntu 18.04 LTS as the operating system. For training the neural networks and during inference, we also use two NVIDIA Tesla T4 GPUs with 16 GiB of GPU memory each.

As usual, training neural networks takes orders of magnitude more time than prediction, but needs to be done only once. In our case, the fastest training run took about 1 hour 25 minutes and the slowest 11 hours 55 minutes on a single GPU. As a general rule, training a return type model takes less time than a parameter type model (due to fewer

4 `https://github.com/gimli-rs/gimli`, `https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasmparser`

5 `https://github.com/OpenNMT/OpenNMT-py`, `https://github.com/google/sentencepiece`

Table 7.2: Most common types, expressed in $\mathcal{L}_{\text{SnowWhite}}$, in our dataset. Short explanations for select types in italics.

Rank	Type	Sample Count	% Total
1	`pointer class` *a pointer to a class*	1,307,617	20.5%
2	`pointer struct`	918,332	14.4%
3	`primitive int 32` *a 32-bit signed integer*	771,690	12.1%
4	`pointer const class`	468,184	7.3%
5	`pointer const struct`	185,635	2.9%
6	`pointer const primitive cchar` *a pointer to constant character(s)*	184,586	2.9%
7	`name "size_t" primitive uint 32` *a 32-bit unsigned integer, named "size_t"*	181,204	2.8%
8	`primitive uint 32`	144,519	2.3%
9	`pointer unknown` *a pointer of unspecified pointee type*	114,139	1.8%
10	`pointer primitive int 32`	101,947	1.6%
Total Samples in Dataset		6,376,307	100%

samples), and training with a simple type language takes less time than with a complex language (due to shorter type sequences). Prediction takes on average between 3 ms and 40 ms per input sample, including beam search to produce multiple predictions. Such near instantaneous results are another advantage of learning-based approaches, as no complex constraint solving is required.

We train all models on the training portion of the data set. During training, we check the accuracy on the validation set and stop early if it regresses. Due to the large dataset, the models converge after one to four epochs. We then take the best model from validation for final evaluation. All final type predictions are obtained on the test data, which the model has never seen and was not used to select the best model.

7.6.2 *High-Level Type Language*

The following evaluates the expressiveness of our type language and the type distribution that results from it.

Table 7.3: Most common extracted type names.

Name	Sample Count	Packages
`size_t`	516,451	63.8 %
`FILE`	20,949	45.2 %
`basic_string<char, ...>`	135,900	17.2 %
`basic_ostream<char, ...>`	35,460	16.3 %
`ios_base`	10,002	16.1 %
`ostreambuf_iterator<char, ...>`	7,801	15.8 %
`va_list`	2,470	15.8 %
`string`	45,081	15.5 %

MOST COMMON TYPES Table 7.2 shows the ten most common types in the dataset expressed in our type language. We observe that several features make the language distinguish large groups of types from each other that would otherwise be merged into imprecise labels. First and most importantly, we see that 7 out of the 10 most common types are some kind of pointer. Without tracking their pointee type, all of those labels would collapse into one, so the recursive nature of our type language is essential for informative predictions. Second, if we did not distinguish `classes` from `structs`, the largest two types would be merged into a single type accounting for 35% of all data, instead of only 20% and 14%, respectively. Third, `const`-ness is also useful, in particular in the pointee type of pointers. Without it, the types with rank four and five would be merged into the first two. Finally, we see that type names are useful to distinguish `size_t` from other, non-specified integers.

MOST COMMON NAMES Table 7.3 lists the eight most common type names as defined in Section 7.3.6, ordered by in how many packages they appear. In total, we extract 239 commonly used names from the dataset. Those names are well-known, semantic types of C and C++ programs, and they are not domain or project specific. The most common name `size_t` appears in almost two thirds of all projects, followed by `FILE` handles in a bit less than half of all projects. The distribution levels off quickly, with ranks three to six containing common types from the C++ standard library related to strings and I/O. All of these names are more useful to a reverse engineer than just the underlying structural type, e.g., `FILE` instead of `pointer struct`. Of the 239 names

Table 7.4: Type distributions of different type languages compared.

Type Language	$\lvert\mathcal{L}\rvert$	$\frac{H}{H_{max}}$	Most Frequent Type			
			Parameter		Return	
$\mathcal{L}_{SW}$, All Names	146,883	0.69	`primitive int 32`	5%	`primitive int 32`	30%
$\mathcal{L}_{SW}$	1,225	0.49	`pointer class`	22%	`primitive int 32`	39%
$\mathcal{L}_{SW}$, Simplified	120	0.42	`pointer struct`	57%	`primitive int 32`	41%
$\mathcal{L}_{Eklavya}$	7	0.38	`pointer`	78%	`int`	51%

in our dataset, 141 (59%) also appear in the test data, so this feature is sufficiently exercised during testing.

When comparing the name distribution to type distributions from high-level languages [Allamanis, Barr, et al. 2020; Pradel, Gousios, et al. 2020], complex data structures are notably absent, e.g., lists or maps. This shows that there is much less of a shared "type vocabulary" in binaries compiled from C and C++ than there is, e.g., in Python.

EXPRESSIVENESS To quantify how expressive our type language is, e.g., compared to a fixed set of types as considered in prior work [Chua et al. 2017; J. He et al. 2018; A. Maier et al. 2019; Pei et al. 2021], we measure how many different types it describes in our dataset. The underlying assumption is that a larger set of types provides more precise type information to users of type prediction, e.g., during reverse engineering.

Table 7.4 compares our language $\mathcal{L}_{SW}$ (short for $\mathcal{L}_{SnowWhite}$), against the two variants described in Section 7.3.7 and the type language of Eklavya [Chua et al. 2017]. Column $\lvert\mathcal{L}\rvert$ shows the number of unique types in the dataset, if expressed in the respective language. For that, we re-extract samples from the binaries with different configuration settings that map DWARF types to the respective languages.

$\mathcal{L}_{SW}$ can distinguish 1,225 unique types, far more than the 7 types of Eklavya, 11 of TypeMiner [A. Maier et al. 2019], 17 of Debin [J. He et al. 2018], or 35 of StateFormer [Pei et al. 2021]. Just by removing names, `const`-ness and the distinction between classes and structs, the simplified variant in the third row results in only 120 unique types in the dataset, so our aforementioned type language features are clearly necessary to express many types more precisely. In the "All Names" vari-

ant, the large amount of project and domain specific names increases the set of types than 100-fold to 146,883 unique types.

We also check that recursion in our type language is useful and even necessary for many types. Of all type samples expressed in $\mathcal{L}_{\text{SW}}$, only 20.7% do not make use of recursion (e.g., primitive types), 48.3% have one nested type constructor (e.g., `pointer` from Figure 7.2), and 31% have an even deeper nesting depth of up to six nested type constructors.

TYPE DISTRIBUTION As another measure of how informative the type language is, we also inspect the resulting distribution of realized types when converting the samples in the dataset to the language. For brevity, we do not show the full distributions, but summarize two key aspects in Table 7.4.

First, column H/H_{max} gives the normalized entropy of the type distribution, where $H_{max} = \log_2 |\mathcal{L}|$ is the entropy of a uniform distribution of the same size. If a type distribution is very non-uniform, e.g., if one type is extremely common, less information can be gained from a single predicted type, and H becomes smaller compared to a more ideal, uniform distribution of types. With the normalized entropy, we can compare the entropy of distributions of different size. Evidently, more expressive type languages have not only more types, but are more uniformly distributed as well, as the entropy increases towards the maximum value of 1.

Table 7.4 also shows the most frequent type, separately for parameter and return types, and how much of the overall distribution that type accounts for. For $\mathcal{L}_{\text{Eklavya}}$, the `pointer` label makes up for almost 80% of the data, a very biased type distribution! Simplified $\mathcal{L}_{\text{SW}}$ without names, `const`, and `classes` is similar to the type language used in StateFormer [Pei et al. 2021], and is only slightly better, as the most common parameter type already accounts for 57% of the data. $\mathcal{L}_{\text{SW}}$ is much more uniform with 22% for the most common parameter type. Interestingly, the return type distribution is less affected by the different languages than the parameter types. Regardless of the type language, the most common label is a primitive integer, accounting for 30% (most expressive language) to 51% (least expressive) of all return types. This may be an artifact of C and C++, where complex results are often written via pointers instead of being returned by value.

SUMMARY We conclude that all features of our type language, in particular type names, recursion, `const`, and the distinction of `class` vs. `struct` help to distinguish types and avoid a biased type distribution. The extracted names convey useful intuitions and are project and domain independent. In general, parameter types are more sensitive to the type language than return types. While more names can be added to the language to increase the number of unique types, the next section shows that accurate prediction also becomes much harder in that case.

7.6.3 *Type Prediction Model*

We now evaluate the accuracy of our type prediction model.

METRICS To compare a predicted type against the ground truth type, we use two metrics. One is *perfect match accuracy*, i.e., the percentage of all predicted types that exactly match the ground truth. We report perfect match accuracy within the top-1 and the top-5 predictions, where the latter retrieves the top five most likely type predictions via beam search.

Perfect match accuracy does not consider partially correct predictions. For example, if the ground truth is `pointer struct`, a prediction of `pointer class` is intuitively better than `primitive int 32`, but since neither are exact matches, they do not count towards accuracy. We thus introduce a metric for type accuracy based on the longest common prefix of the prediction and the ground truth. The *type prefix score* of a prediction t' and ground truth t is the length of the common prefix $TPS(t', t) = |commonPrefix(t', t)|$. That is,

$$TPS(\texttt{pointer struct}, \texttt{pointer class}) = 1, \text{ but}$$
$$TPS(\texttt{pointer struct}, \texttt{primitive int 32}) = 0.$$

Computed over the whole test set, *TPS* gives the average number of type tokens that are correct until the predicted sequence diverges from the ground truth.

BASELINE As there is no existing learning-based type prediction for WebAssembly binaries, we compare our model against a statistical baseline. This baseline exploits that the low-level WebAssembly type t_{low} is available in the binary for each parameter and return sample. Given only the t_{low} of an input, we can "generate" top-k predictions by copy-

Table 7.5: Model accuracy on different type prediction tasks, compared with a conditional probability baseline.

(a) Parameter type prediction task.

Type Language	$\mathcal{L}_{\text{SW}}$	$\mathcal{L}_{\text{SW}}$, All Names	$\mathcal{L}_{\text{SW}}$, Simplified	$\mathcal{L}_{\text{Eklavya}}$	$\mathcal{L}_{\text{SW}}$, t_{low} not given
Seq-to-seq Model, see Section 7.4					
Top-1 Accuracy	44.5%	18.6%	65.1%	87.9%	42.4%
Top-5 Accuracy	75.2%	27.1%	86.2%	100.0%	73.4%
Type Prefix Score	1.47	1.31	1.62	0.88	1.45
Statistical Baseline, based on $P(t_{high} \mid t_{low})$					
Top-1 Accuracy	28.7%	13.0%	47.1%	77.1%	
Top-5 Accuracy	61.4%	20.8%	78.1%	99.9%	N/A
Type Prefix Score	1.05	0.28	1.24	0.77	

(b) Return type prediction task.

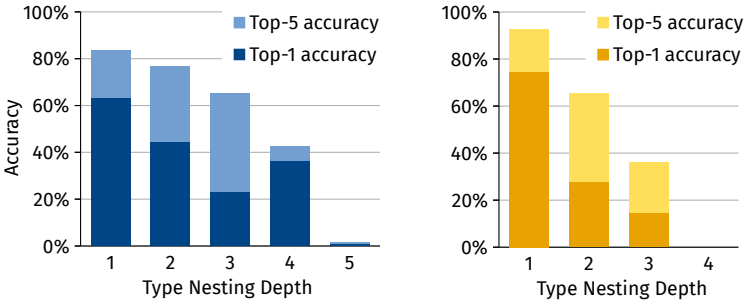
Type Language	$\mathcal{L}_{\text{SW}}$	$\mathcal{L}_{\text{SW}}$, All Names	$\mathcal{L}_{\text{SW}}$, Simplified	$\mathcal{L}_{\text{Eklavya}}$	$\mathcal{L}_{\text{SW}}$, t_{low} not given
Seq-to-seq Model, see Section 7.4					
Top-1 Accuracy	57.7%	40.6%	60.6%	76.3%	50.7%
Top-5 Accuracy	80.5%	47.3%	87.9%	100.0%	81.2%
Type Prefix Score	1.37	1.00	1.38	0.76	1.02
Statistical Baseline, based on $P(t_{high} \mid t_{low})$					
Top-1 Accuracy	49.9%	41.7%	50.7%	64.6%	
Top-5 Accuracy	74.2%	48.5%	81.4%	100.0%	N/A
Type Prefix Score	1.14	0.92	1.16	0.65	

ing the k most likely high-level types for a given t_{low} from the conditional probability distribution $P(t_{high} \mid t_{low})$ that was empirical observed on the training data. For example, the most common type for t_{low} = `i32` is `pointer class`, and for t_{low} = `f32` it is `primitive float 32`.

RESULTS Table 7.5 shows the model accuracy for parameter (a) and return type prediction (b). For our proposed type language $\mathcal{L}_{\text{SW}}$, we see that the model predicts the exactly correct parameter type in 44.5% of the cases, even though there is a large choice out of 1,225 unique types. If we accept any of the model's top five predictions, the model is right for 75.2% of the test samples. This is also not just because of a skewed data distribution, as the baseline exploiting the underlying data distribution achieves only 28.7% top-1 exact match accuracy, significantly less than the neural model. The model accuracy is even better for return type prediction with a top-1 (top-5) accuracy of 57.7% (80.5%). On average, the type prediction model gets the first 1.47 (1.37) tokens of the type sequence correct for parameter (return) types. We expect the first tokens to be likely the most relevant to a reverse engineer, so this is a good result.

We also judge the hardness of type prediction with different languages. Without filtering names ($\mathcal{L}_{\text{SW}}$, All Names), the task becomes much too difficult, with a top-1 accuracy of only 18.6% on parameter types. This motivates our restriction to a vocabulary of common type names. At the other end, for the simple language $\mathcal{L}_{\text{Eklavya}}$ we achieve a top-1 accuracy of 87.9%, compared with an accuracy of around 81% on native binaries reported in [Chua et al. 2017]. However, the statistical baseline also casts doubt on whether this is really an achievement of the model, or simply a very easy task, as even the baseline achieves 77.1% top-1 accuracy without any neural network. The model for simplified $\mathcal{L}_{\text{SW}}$ sits between $\mathcal{L}_{\text{SW}}$ and $\mathcal{L}_{\text{Eklavya}}$, with a top-1 accuracy of 65.1% (60.6%) on parameter (return) types. In general, we can conclude that the neural model accuracy is consistent with the complexity of the type language. A good type language for learning-based type prediction must balance the trade-off between being precise and allowing for accurate predictions.

ABLATION STUDY AND TYPE DEPTH The rightmost column for both parameter and return type prediction in Table 7.5 is an ablation study as to how much passing the WebAssembly low-level type t_{low}



(a) Parameter types. **(b)** Return types.

Figure 7.3: Prediction accuracy of $\mathcal{L}_{\text{SW}}$ by type nesting depth.

helps the model to predict high-level types. For that, we take the same language as in $\mathcal{L}_{\text{SW}}$, but remove the low-level type from the beginning of the input sequence. For parameter types, the low-level type seems to help only marginally (a difference of $\approx$ 2% in accuracy), possibly because there are enough cues from the parameter usage even without explicitly passing the low-level type. For return type prediction, the low-level type seems more useful, with an accuracy difference of approximately 9%.

Finally, Figure 7.3 shows the prediction accuracy of $\mathcal{L}_{\text{SW}}$, separately for different type nesting depths. The general trend is that accuracy decreases with more deeply nested types, as expected. However, even for types with three (four) nested levels, parameters can still be predicted exactly with a top-5 accuracy of 65% (43%). Return types are less deeply nested in general and prediction accuracy is also worse beyond types with a single or two nested levels.

7.6.4 *Case Studies of Predictions*

We show representative examples of predictions produced by the model, to get an intuition how useful it is in practice.

EXAMPLES IN `libgdal` For predicting the first parameter sample in the shuffled test set, the $\mathcal{L}_{\text{SW}}$ model is given a WebAssembly input of 8 instruction windows, containing 168 instructions and 453 tokens in total. The input starts with:

```
i32 <begin> global.get 1 ; i32.const 294552 ; i32.add ;
```

```
i32.const 3 ; i32.const 1 ; local.get <param> ; call ...
```

From just this input, the model's top five predictions are:

```
pointer name "FILE" struct
pointer struct
primitive int 32
pointer primitive cchar
pointer const primitive cchar
```

The sample corresponds to the fourth parameter[6] `fp` of a class method in `libgdal`, a geospatial library written in C++. The method declaration in the source code reads:

```
void DDFSubfieldDefn::DumpData(
    const char * pachData, int nMaxBytes, FILE * fp ) ...
```

As we can see, the top-most prediction of the model is exactly correct, the parameter is indeed a pointer to a file handle. For a reverse engineer, we believe this prediction in our type language is much more useful than, e.g., `pointer struct` by StateFormer or just `pointer` by Eklavya. The top-2 prediction would not have been incorrect either, just not as precise, justifying our type prefix metric for evaluation.

Staying with the example, the model's predictions for the parameter `nMaxBytes` of the same function are as follows:

```
pointer const primitive uint 8
primitive int 32
pointer const primitive cchar
pointer struct
pointer primitive uint 8
```

Here, the top-1 prediction is not correct, but the top-2 prediction is. It is unclear how the model came up with the top-1 prediction; we can only speculate that it got confused by instructions related to the other parameter, whose string type is closer to the predictions.

EXAMPLES IN `libtiff` Taking the first return type sample in the test set from the next library, the model attempts to predict the return type of the following function in `libtiff`:

```
int JPEGVGetField(TIFF* tif, uint32 tag, va_list ap) ...
```

The model's top five return type predictions are:

```
primitive int 8
primitive uint 32
```

6 We regard the methods' receiver object as the implicit first parameter.

```
primitive uint 32
pointer name "Exception" class
primitive int 32
```

The correct prediction is on place five. As the raw model is not constrained to generate five unique predictions, we also see two duplicate predictions. In a production-grade type prediction tool, the raw model outputs could be filtered to only include unique types. Interestingly, the body of the function (not shown) returns a literal 1, so the other primitive predictions would have actually been type compatible in C, showing the difficulty in getting accurate training data.

Finally, we inspect the top-most prediction for the first parameter of the same function, where the model returns `pointer struct`. As a domain specific name, `TIFF` is not shared among enough projects to be in our list of common type names. The predicted type is thus correct and as precise as possible as per our type language $\mathcal{L}_{\text{SW}}$. Future work could explore to predict information about the struct fields as well.

7.7 SUMMARY

This chapter presents the first neural approach for recovering precise types in WebAssembly binaries. In contrast to prior work on learning-based binary type prediction, we represent types through an expressive type language. The language allows for thousands of different types, instead of the 7 to 35 types considered previously. Despite this increase of expressiveness, we find our type prediction model to be highly accurate, exactly predicting 44.5% (75.2%) of all parameter types and 57.7% (80.5%) of all return types within the top-1 (top-5) predictions. SnowWhite is an important first step toward reverse engineering WebAssembly binaries. Beyond our technique, we share a novel large-scale dataset of C and C++ code compiled to WebAssembly with debug information, which is two orders of magnitude larger than existing datasets for WebAssembly.

8 RELATED WORK

In this chapter, we discuss work that is closely related to this dissertation. Naturally, research can be related along multiple dimensions, suggesting different but equally valid ways of slicing and dicing them into a linear structure for this chapter. We chose to categorize the related work as follows.

- Several chapters of this dissertation focus on *security aspects of WebAssembly*. Section 8.1 gives an overview of related work in this area. The following Section 8.2 then discusses *other work on WebAssembly*.
- In particular WASABI draws a lot of inspiration from established *dynamic analysis and instrumentation* tools, e.g., for native binaries and JavaScript. We discuss those in Section 8.3.
- *Studies* on large bodies of code and software ecosystems (in particular the Web), are related to our work on WASMBENCH. We discuss them in Section 8.4.
- In Chapter 3, we frequently compare WebAssembly and *native code* with respect to *software security*. Section 8.5 covers existing work on native code, both attacks and defenses.
- *Fuzzing* is at the heart of our FUZZM project. Section 8.6 discusses related work, especially on fuzzing for native code.
- Finally, in Section 8.7 we discuss related work for SNOWWHITE. We discuss *reverse engineering* and *type recovery*, and *neural methods for code* due to the learning-based approach.

8.1 SECURITY ASPECTS OF WEBASSEMBLY

One way to categorize work on security aspects of WebAssembly is whether the WebAssembly code is seen as malicious, and thus shall be protected against, or as vulnerable, and thus shall itself be protected. In the first category falls work on attacks against WebAssembly runtime implementations, cryptomining, and side-channel attacks. Our work focuses on the second category, namely vulnerable WebAssembly code and how we can protect it from malicious inputs. We detail work in both categories in the following.

CRYPTOMINING Among the early adopters of WebAssembly have been websites that use the computing resources of unsuspecting visitors to mine cryptocurrencies, a practice also known as *cryptojacking* [Musch et al. 2019b; Rüth et al. 2018]. This is often unwelcome and hence an instance of malicious usage of WebAssembly. Several approaches detect and defend against cryptojacking [Kharraz et al. 2019; Konoth et al. 2018; W. Wang et al. 2018], e.g., by analyzing the dynamically executed instruction profile, which can be characteristic for certain cryptographic hash functions used in the mining process. WASABI can be used to conveniently collect such a profile, as we show with the example analysis in Listing 5.1.

A study by Musch et al. [2019a] finds that many WebAssembly modules loaded by the top one million websites in 2019 were used for cryptomining. Together with intentionally obfuscated code, malicious uses of WebAssembly account for 50% of the samples in their dataset. In our study in Chapter 4, we also analyze the uses for WebAssembly on the Web. We find that today, cryptomining is no longer a dominating use case, and that other (benign) uses are much more prevalent. This is also supported by the observation of Varlioglu et al. [2020] that cryptojacking subsided after *Coinhive*, a major provider of cryptomining infrastructure, ceased operation. W. Wang et al. [2018] discuss limitations of VirusTotal in identifying cryptomining, which may impact the validity of our results in Section 4.3.5. However, we also perform manual inspection of randomly selected binaries and compare our dataset with the one provided by Musch et al. [2019a], both of which additionally support our findings of Chapter 4.

OTHER MALICIOUS USES Malicious WebAssembly binaries are also crafted to escape browser sandboxes and gain remote code execution

[Plaskett et al. 2018; Silvanovich 2018]. Unlike our work in Chapter 3, those exploits attack bugs in specific VM implementations and fall into the realm of *host security*, as discussed in Section 3.1. In contrast, we do not aim to escape the sandbox, and our attacks assume nothing but a standards-compliant WebAssembly VM. For example, the exploit in Section 3.4.1 works in both Firefox and Chrome. Since we do not escape the VM, we depend on the available imported host functions for malicious actions. However, as we show with our end-to-end exploits in Section 3.4, cross-site scripting, remote code execution, and file writes can still be consequences.

With untrusted, low-level code such as WebAssembly, *side-channels attacks* are another cause for worry. The language and its predictable compilation offer increased control for an attacker, e.g., over the memory layout of program data. Genkin et al. [2018] extract cryptographic keys by exploiting a cache timing side-channel with WebAssembly. Maisuradze and Rossow [2018] demonstrate speculative execution attacks with a WebAssembly exploit. On the defense side, Narayan, Disselkoen, Moghimi, et al. [2021] propose software-only and hardware-assisted hardening of WebAssembly VMs against speculative execution attacks. Browser vendors also try to reduce the likelihood of successful side-channel attacks, but a general solution is hard to come by. Firefox has artificially reduced JavaScript timer precision [Wagner 2018], to make reading the side-channel signal more noisy, but that is not a principled defense. Chrome relies on site isolation,[1] i.e., putting different websites (and thus protection domains) into separate processes [Reis 2018]. Firefox will follow suit [Gakhokidze 2021].

Finally, in recent work with collaborators [Romano, Lehmann, et al. 2022], we explore yet another malicious use case for WebAssembly, namely *obfuscation*. By translating parts of JavaScript code to WebAssembly, e.g., control-flow constructs, string literals, and suspicious function calls, we can evade static malware detectors for JavaScript. That work is not part of this dissertation.

VULNERABLE WEBASSEMBLY Two industry whitepapers show example attacks against vulnerable WebAssembly binaries [Bergbom 2018; McFadden et al. 2018]. Their pioneering work inspired us to investigate WebAssembly binary security more thoroughly and expand this research significantly in several directions (Chapter 3).

1 https://www.chromium.org/Home/chromium-security/ssca/

In Section 3.2, we systematically analyze how data in the source code is mapped to linear memory by three different compilers, two backends, and two linker configurations, whereas previous work has only looked at select examples from a single compiler. From our analysis we conclude that, fundamentally due to linear memory, WebAssembly cannot separate static data, heap, and unmanaged stack, as guard pages like in native binaries are unavailable. Unlike previous work, we thus show a much larger set of attack primitives, including primitives have not been reported for WebAssembly at all. Because linear memory has no page protections, not even string literals are safe from modification (Section 3.3.2.3), which has not been reported before. Our end-to-end exploits show that this can render a single call to `fprintf` dangerous, even when passing only "constant" arguments. We are also the first to propose stack overflows (not to confuse with stack-based buffer overflows) as an attack primitive (Section 3.3.1.2). Prior work has hypothesized that exploitation is possible, but we are the first to demonstrate it in practice. One whitepaper and a blog post [Denis 2018; McFadden et al. 2018] warn that WebAssembly binaries come with their own allocator, which is potentially not hardened. Our exploits against two different versions of Emscripten's `emmalloc` substantiate their hypotheses.

In Section 3.5, we also perform the first quantitative security evaluation on a set of 26 WebAssembly binaries with more than 19 million instructions in total. A previous blog post [Foote 2018] explores that indirect calls can be redirected to unintended functions on a single example. We make this observation quantifiable and measure that almost every second function can be reached via an indirect call that takes its argument directly from linear memory. We are also the first to estimate how much data resides on the unmanaged stack in linear memory (with extended measurements on real-world binaries in Section 4.3), a relevant number for estimating the risk from previously described data overwrite primitives. Finally, we are the first to compare WebAssembly's type-checking of indirect calls with native CFI schemes.

WEBASSEMBLY FOR SFI As we discuss in Chapter 3, the protections inside WebAssembly's own memory are severely limited. However, some approaches piggyback on the well-designed host security of WebAssembly, to isolate at least some components of an application from the rest of the system. This is a classical goal of *software fault isolation (SFI)* [Wahbe et al. 1993]. Narayan, Disselkoen, Garfinkel, et al.

[2020] compile libraries that are common attack targets in browsers, e.g., image codecs, to WebAssembly and then run the code in the WebAssembly VM in the browser, instead of natively. Zakai [2020] presents a related, but slightly different approach. The code that shall be protected is also compiled to WebAssembly, but then translated into C again, statically "embedding" the WebAssembly host isolation in the code. Finally, the resulting C code becomes part of the overall application and is compiled with a regular compiler to native code.

8.2 OTHER WORK ON WEBASSEMBLY

Besides work on security, there are is also other related work on WebAssembly in general.

LANGUAGE AND SPECIFICATION The first publication on WebAssembly is of course the initial paper that introduced the language to the academic community [Haas et al. 2017]. It is followed by a more concise, updated journal version [Rossberg et al. 2018]. Both focus on the abstract syntax, execution semantics, and type system of the language. Fewer pages are devoted to the systems aspects, i.e., the binary format, implementations in existing browser engines, and performance.

Going beyond the pen-and-paper formalization, Watt [2018] presents a fully mechanized specification of the WebAssembly language in Isabelle/HOL, including a verified interpreter and a soundness proof for the type system. His work found an error in the type system, which was fixed before the official standardization. Watt, Rao, et al. [2021] expand this effort to a second mechanization in Coq.

EXTENSIONS After the initial *Minimum Viable Product* of WebAssembly and its 1.0 specification [WebAssembly Specification], several language extensions have been proposed. There is an official process by the language working group for adopting proposals into the standard; and some proposals have already been merged into the specification (see our discussion of language extensions at the end of Section 2.2).[2]

There are also proposal papers that do not follow this process. Those cover more niche applications, such as statically verifying that operations have constant runtime [Watt, Renner, et al. 2019], to ensure the absence of certain classes of side-channel attacks against cryptogra-

2 https://github.com/WebAssembly/proposals/blob/main/finished-proposals.md

phy implementations. Other proposals have wider impact, but are at an early stage. Disselkoen et al. [2019] propose to solve some of the memory safety issues we discuss in Chapter 3 by adding managed memory handles with defined (temporal) lifetime and (spatial) extent. However, as far as we know, this has not been implemented in a major runtime or otherwise progressed towards inclusion into the official language. We discuss other language proposals that could improve the security of WebAssembly binaries in Section 3.6. Finally, the effort for adding atomics and threads to the language is accompanied by a publication formalizing a weak memory model for WebAssembly [Watt, Rossberg, et al. 2019].

SOURCE LANGUAGES For an overview of source languages compiling to WebAssembly, we refer to the background in Section 2.3 and our findings in Chapter 4. In terms of publications, Protzenko et al. [2019] compile cryptographic libraries written in a verification-oriented programming language *Low** (a subset of *F**) to WebAssembly.

PERFORMANCE AND CORRECTNESS Since one use case of WebAssembly are compute-intensive applications, the performance of WebAssembly code and specific runtimes has been studied from early on. The original publication [Haas et al. 2017] has measured runtime overhead versus native code only on a small, uniform set of numerical benchmarks.[3] Jangda et al. [2019] compile the more diverse SPEC CPU benchmark suite to WebAssembly. They find a larger overhead than Haas et al. [2017], at about 50% slowdown when executing WebAssembly in Firefox and Chrome, compared with native code. In case studies, they attribute the overhead to suboptimal compilation to native code (which can be fixed, e.g., through improved register allocation), and to safety checks that are inherent to WebAssembly (e.g., runtime type checking of indirect calls). D. Herrera et al. [2018] compare the performance of WebAssembly against native code and JavaScript on the Ostrich benchmarks.[4] For those numerical programs, WebAssembly in Firefox and Chrome was faster than JavaScript and about as fast as native code. The performance of our fuzzer in Chapter 6 crucially relies on the execution speed of the integrated WebAssembly VM. We build on Wasmtime [Wasmtime Website]. It is quite young but actively developed by

3 https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/
4 http://www.sable.mcgill.ca/mclab/projects/ostrich/

Fastly and other open-source contributers. We can profit from future performance improvements in the VM, as our approach is orthogonal to them. Our instrumentation at the bytecode level is also completely independent of specific (versions of) WebAssembly runtimes.

Besides studies on WebAssembly runtimes and their performance, Romano, Liu, et al. [2021] also study bugs in compilers targeting WebAssembly, with one of their two studies focusing on *Emscripten*. They find that some bugs are silent miscompilations that only manifest at runtime, and that there are both bugs in the parts of compilers that are unrelated to WebAssembly and bugs possibly caused by unusual WebAssembly features, e.g., the different memory model than native code, or the asynchronous nature of the Web.

PROGRAM ANALYSIS Others have also worked on program analysis techniques for WebAssembly, although in different areas than we do. In terms of *static analyses*, Stiévenart, Binkley, et al. [2022] present a backwards slicing algorithm for WebAssembly. To make sure the resulting slice has the same stack typing as the original program fragment, they introduce synthetic `const` and `drop` instructions. They evaluate their implementation with binaries from our WASMBENCH dataset of Chapter 4. They also work on a general-purpose static analysis framework, called *Wassail*,[5] but beyond a presentation abstract [Stiévenart and De Roover 2021], there has been no full publication on it yet. The framework seems to have originated from earlier work on static information flow analysis for WebAssembly [Stiévenart and De Roover 2020].

There are also more heavy-weight static analyses for WebAssembly. N. He et al. [2021] present a symbolic execution framework for EOSIO smart contracts, which are deployed as WebAssembly bytecode. *Manticore* is another symbolic execution framework, which supports multiple native architectures and WebAssembly.[6] Brito et al. [2022] present *Wasmati*, a static analysis tool based on code property graphs for WebAssembly. They build on two results from this dissertation. First, they use one of our exploits from Section 3.4 as a motivating example in their paper. Second, they use our WASMBENCH dataset (Chapter 4) for testing the robustness of their analysis.

There are also some practical tools for statically optimizing WebAssembly binaries. A standard tool that is often run as a post-processing

5 https://github.com/acieroid/wassail
6 [Hennenfent 2020], https://github.com/trailofbits/manticore

step on binaries produced with other compilers, is `wasm-opt` from the *Binaryen* project.[7] There is no publication on it, but as far as we know from using it and inspecting its source code, it converts WebAssembly programs to its own intermediate representation and performs classical compiler optimizations, such as inlining or common subexpression elimination. A slightly narrower use case is covered by *Twiggy*,[8] which is a static code size analyzer. It lists each function with its usages and its retained size, i.e., how much the binary could be shrunk if the function and the then transitively dead functions were (manually) removed.

8.3 DYNAMIC ANALYSIS AND INSTRUMENTATION

WASABI in Chapter 5 relates to many works in the area of dynamic analysis. For binaries, instrumentation is one common implementation strategy, which we also cover here.

DYNAMIC ANALYSIS IN GENERAL Dynamic analysis [Ball 1999] has since long been recognized as an effective way to complement static analysis [Ernst 2003]. In the hierarchy of program analyses proposed by Zeller [2003], our dynamic analysis with WASABI is at the second level, namely observing the runtime behavior of programs. This contrasts with the lower, deductive level, where behavior is derived purely from static analysis of the program, and with the higher level of experimentation, where execution is actively steered towards some (testing) goal. FUZZM (Chapter 6) fits the latter level in this hierarchy.

Various concrete dynamic analyses have been proposed, including dynamic slicing [Agrawal and Horgan 1990], taint analyses for x86 binaries [Newsome and Song 2005] and Android [Enck et al. 2014], an analysis to track the origin of null and undefined values [Bond et al. 2007], and analyses to understand performance problems [Yu et al. 2014]. Given the increasing interest in WebAssembly, we expect an increased demand for dynamic analyses for WebAssembly, for which WASABI provides a reusable platform.

DYNAMIC ANALYSIS FOR THE WEB Motivated by the dynamic features of JavaScript, e.g., runtime loading of code, various dynamic analyses for JavaScript-web applications have been presented. Such work

7 https://github.com/WebAssembly/binaryen
8 https://github.com/rustwasm/twiggy

includes analyses to find type inconsistencies [Pradel, Schuh, et al. 2015], JIT-unfriendly code [Gong, Pradel, and Sen 2015], bad coding practices [Gong, Pradel, Sridharan, et al. 2015], data races [Petrov et al. 2012], and incorrect usage of promises [Alimadadi et al. 2018]. The survey of Andreasen et al. [2017] gives a broader overview. Many of these analyses are built on top of Jalangi [Sen et al. 2013], a general-purpose dynamic analysis framework for JavaScript. Because there was no comparable tool for WebAssembly, we aimed to fill this gap with WASABI.

DYNAMIC ANALYSIS OF WEBASSEMBLY Even though there was no dynamic analysis framework for WebAssembly, several manually written approaches had already been implemented when we started our work on WASABI. There are two dynamic taint analyses [W. Fu et al. 2018; Szanto et al. 2018] and a cryptomining detector [W. Wang et al. 2018]. They are implemented by modifying the V8 engine [W. Fu et al. 2018], by implementing a simple WebAssembly interpreter in JavaScript [Szanto et al. 2018], and through custom binary instrumentation [W. Wang et al. 2018], respectively. Our evaluation in Chapter 5 shows that these analyses and others can be implemented on top of WASABI with significantly less effort. These handwritten analyses also highlight that there is a real need for a general-purpose framework that abstracts away the low-level details of implementing a dynamic analysis.

BINARY INSTRUMENTATION Binary instrumentation has been a popular strategy to implement dynamic analyses in the past. Often used tools for x86 binaries include *DynamoRIO* [Bruening et al. 2003], *Pin* [Luk et al. 2005], and *Valgrind* [Nethercote and Seward 2007], which have provided inspiration for WASABI. These tools instrument binaries at runtime by translating basic blocks just before their execution, and by storing translations in a code cache. Effectively, those are just-in-time compilers for native code. This is necessary, because reliable static disassembly for native code is hard [Andriesse et al. 2016]. In contrast, WASABI instruments binaries statically, i.e., ahead of the analysis, which avoids any instrumentation overhead during execution. WASABI also differs with respect to the API it provides to analysis authors: While DynamoRIO provides an API to manipulate instructions, WASABI provides an API to observe the execution of instructions. Analyses written for Pin can specify "instrumentation routines", which determine where to place calls to analysis routines. Instead, WASABI se-

lects which kinds of instructions to instrument based on the hooks required for an analysis. *Umbra* [Zhao et al. 2010] is a dynamic binary instrumentation tool that focuses on efficient memory shadowing. In contrast, WASABI provides a general-purpose framework for arbitrary dynamic analysis, including memory shadowing. A difference compared to all the above tools is that in WASABI, the dynamic analysis is written and executed in a high-level language, JavaScript, instead of being compiled to binary code. The rationale is that JavaScript is already very popular on the Web, making it easier for analysis authors to adopt WASABI.

8.4 STUDIES OF CODE AND ECOSYSTEMS

Our work on WASMBENCH in Chapter 4 is related to the other studies of code. One important subset are studies of the Web, as a large part of our dataset comes from crawling live websites and because the Web is a major host environment for WebAssembly.

WEBASSEMBLY The most closely related work to our study in Chapter 4 is by Musch et al. [2019a]. They systematically collect WebAssembly from the Web and report cryptomining to be one of its prime use cases in 2019. Our work in WASMBENCH extends their findings in several ways: First, we consider a wider range of sources to gather binaries, not just the Web, but also package managers, Firefox extensions, source code repositories, and manually collected samples. This results in 58 times more binaries in our study. Second, we show that other applications than cryptomining have become much more prevalent for WebAssembly (see also Section 8.1). Third, we study several properties of WebAssembly not considered before, e.g., source languages, common toolchains, and security properties such as potentially dangerous imported functions.

One motivation for WASMBENCH was to serve as a realistic test set for tools processing WebAssembly binaries. We ourselves have used binaries from it for testing in Section 6.5. In less than a year also other researchers used WASMBENCH, e.g., Brito et al. [2022] and Stiévenart, Binkley, et al. [2022] evaluate their tools on our dataset. Besides testing, Brito et al. [2022] also apply their vulnerability scanner on our dataset, searching with hand-written queries for problematic patterns.

One such pattern is the usage of deprecated C functions, e.g., `gets` or `strcat`, of which they find more than 150 uses.

Before WASMBENCH, there was no large dataset of realistic WebAssembly binaries. E.g., for testing WASABI during development in 2018, we relied on a small set of hand-picked applications and the WebAssembly spec test suite. For their analysis of WebAssembly code generation, Jangda et al. [2019] use the SPEC CPU suite. We do so as well in Section 3.5. One limitation of our WASMBENCH dataset is that it only contains the WebAssembly binaries, but no surrounding JavaScript code or test inputs. That makes it difficult to apply in dynamic testing.

STUDIES OF THE WEB AND JAVASCRIPT Beyond WebAssembly, other related studies investigate JavaScript code bases and web security. Richards et al. [2011] analyze the usage of `eval` in JavaScript, which is discouraged because it can add (potentially malicious) code at runtime. We analyze how often it and similarly dangerous functions are imported into WebAssembly binaries in Section 4.3.4.3. Skolka et al. [2019] study how frequent minified and obfuscated code is on the Web. This relates to Section 4.3.7, where we look into whether useful names are present in WebAssembly binaries. Other studies of the JavaScript ecosystem are on outdated libraries in the Web [Lauinger et al. 2017], implicit type conversions in JavaScript code [Pradel, Schuh, et al. 2015], regular expression Denial of Service vulnerabilities in JavaScript-based web servers [Staicu and Pradel 2018], cross-site scripting vulnerabilities [Melicher et al. 2018], and how large amounts of code sharing can make JavaScript applications more vulnerable [Zimmermann et al. 2019]. Inspired by all that work, Chapter 4 fills in gaps in the existing knowledge about security properties of real-world WebAssembly.

Parts of our methodology in Section 4.2 were inspired by the following two works. First, unlike other crawling-based studies, we do not rely on the *Alexa* list of the top one million websites. One reason is that its latest version is no longer publicly available, and an older version might not reflect the latest state of the Web. Another reason is that it has been found prone to manipulation [Pochat et al. 2019]. We instead use the *Tranco* list as a seed list for our crawling.[9] Second, we realize that starting web crawling from top-level domains is causing only shallow exploration, which is unlikely (in our case) to find many WebAssembly binaries. Instead, the survey of the *Web Almanac* [Goel 2021]

9 https://tranco-list.eu/

is based on more extensive inputs, e.g., URL lists from the *Chrome User Experience Report* and large-scale crawled data from the *HTTP Archive* (which we cannot match by crawling ourselves with limited computing resources). Hence, we also integrate those sources in Section 4.2 for the Web portion of our dataset.

8.5 SECURITY OF NATIVE SOFTWARE

Our work in Sections 3 and 6 draws a lot of parallels to the security of native software. We give a short overview of attacks and defenses in that area in the following.

ATTACKS There exists ample work on binary exploitation; Szekeres et al. [2013] provide an excellent overview of techniques for exploiting memory vulnerabilities. In particular, their Figure 1 inspired us to create a WebAssembly-specific overview of attacks and defenses in Chapter 3, Figure 3.1. Their stages 1 to 3 correspond to our first attack primitive dimension (overwriting data), techniques from stage 4 (randomization such as ASLR) do not exist in WebAssembly, and stages 5 and 6 correspond to our third attack primitive dimension (triggering security-compromising behavior). Overall, we find that, although concrete exploits have to be adapted and effects depend on the runtime environment, many techniques that are effective in native binaries also transfer to WebAssembly. This is even true for long known attacks, such as simple stack-based buffer overflows [Aleph One 1996] and attacks on allocator metadata [Anonymous 2001; Kaempf 2001].

EXPLOIT MITIGATIONS Because it is hard to comprehensively fix the underlying memory vulnerabilities in all legacy code out there, and rewriting in safer languages is even harder, many exploit mitigations have been developed. Those do not prevent the memory error itself, but try to limit its consequences for the system. They include data execution prevention (DEP) [Andersen and Abella 2004], stack canaries [Cowan et al. 1998], address-space layout randomization (ASLR) [PaX Team 2002], and safe unlinking in allocators. Their applicability to WebAssembly is a mixed bag. DEP is truly not necessary, because data is not executable by design (Section 2.2). ASLR would be of arguable use for WebAssembly, as 32 bit addresses do not provide enough entropy for effective randomization [Shacham et al. 2004]. However, we

find stack canaries and safe unlinking to be missing in WebAssembly, unjustifiably so.

CONTROL-FLOW INTEGRITY In recent years, different variants of control-flow integrity [Abadi et al. 2005] have been proposed as another defense for native binaries. We discuss the high-level idea and how it relates to WebAssembly's type checking of indirect calls in Section 3.5.4. Burow et al. [2017] provide a survey assessing the security of different native CFI implementations. We empirically compare with some of their results and find that type checking of indirect calls in WebAssembly is not as restrictive as dedicated CFI enforcement. In particular, low-level WebAssembly types are much more cross-compatible than source-level types. The latter are used in compiler-inserted CFI defenses, such as GCC's VTV (vtable verification) or LLVM's CFI [Tice et al. 2014]. One upside of WebAssembly is however that return addresses are managed by the VM, which provides the same level of security as a shadow stack does for native code.

BINARY REWRITING AND OVERFLOW PROTECTION Our FUZZM project relies on adding oracles by statically rewriting WebAssembly binaries before fuzzing them (Section 6.3). We compare to similar rewriting techniques for native binaries in the following. Statically instrumenting native binaries is challenging due to mixed code and data, difficult function identification [Andriesse et al. 2016], and even undecidable disassembly in the general case [Wartell et al. 2011]. Many rewriting approaches for native binaries thus rely on metadata being available, e.g., debug information, relocations, or position-independent code [X. Chen et al. 2015; Dinesh et al. 2020; Slowinski et al. 2012]. Without it, one has to resort to tricks, such as instruction punning [Duck et al. 2020], which do not achieve full coverage of all instructions either. WebAssembly can be disassembled reliably and thus does not suffer from these problems. However, WebAssembly instrumentation in FUZZM poses also new challenges, such as handling relative branch target labels or multiple function returns.

Several papers have presented binary rewriting techniques for protecting the stack in x86 programs. Early work by Prasad and Chiueh [2003] uses a shadow stack approach that duplicates return addresses into a second location. *BodyArmor* inserts instrumentation that monitors reads and writes relative to pointers [Slowinski et al. 2012]. *StackAr-*

mor combines a randomized layout, isolation, and secure allocation for hardening binaries against stack-based vulnerabilities [X. Chen et al. 2015]. *RetroWrite* [Dinesh et al. 2020] uses an overflow detection mechanism similar to *AddressSanitizer* [Serebryany et al. 2012], i.e., using shadow memory to mark bytes that may not be accessed and instrumenting memory accesses. Unlike the compiler-based AddressSanitizer, their approach only detects overflows at the granularity of stack frames and heap objects. This is equally true for FUZZM.

There has also been work on protecting binaries against heap overflows. Robertson et al. [2003] and Nikiforakis et al. [2013] both present techniques that, like the FUZZM heap canaries, instrument the allocation and deallocation functions such that they insert canaries. Nikiforakis et al. check the canaries at system calls, which means it is likely that overflows are detected early, but at the cost of having to check the canaries often. We instead opted for the more efficient option of checking canaries during deallocation, which is more appropriate for our fuzzing and hardening use cases.

8.6 FUZZING

We now discuss related work in the area of fuzzing, and contrast it with FUZZM, our binary-only greybox fuzzer for WebAssembly presented in Chapter 6.

FUZZING WEBASSEMBLY There is one early prototype for fuzzing WebAssembly programs that is most closely related to our work.[10] Their approach is based on *libFuzzer*,[11] which implies several differences to our work. First, to be able to integrate libFuzzer into an application, they require modifying the source code, whereas FUZZM is binary-only. Also, the coverage information in their case is obtained through compiler instrumentation, whereas our instrumentation is applied to binaries and thus compiler independent. Finally, in their approach the input generation code is compiled into the application, and thus is executed in the WebAssembly VM as well, whereas our input generation is run in native code (namely that of AFL) and only the target binary and coverage instrumentation is run in a VM.

10 https://github.com/jonathanmetzman/wasm-fuzzing-demo
11 https://llvm.org/docs/LibFuzzer.html

Fuzzing has also been used for testing WebAssembly VMs,[12] which is orthogonal to fuzzing WebAssembly programs. In that scenario, the fuzzer generates WebAssembly programs hoping to trigger bugs in the parser, compiler, and optimizations of the virtual machine, not bugs in the WebAssembly applications themselves.

AFL AND OTHER FUZZERS Out of the many approaches for greybox fuzzing [P. Chen and H. Chen 2018; Manès et al. 2021; Mathis, Gopinath, Mera, et al. 2019; P. Srivastava and Payer 2021], we build FUZZM on the popular AFL fuzzer.[13] Unlike its commonly used GCC and LLVM modes and the majority of other fuzzers [Manès et al. 2021], we do not require source code, and do not rely on compiler-added oracles [Österlund et al. 2020; Serebryany et al. 2012]. Instead, due to a multitude of source languages compiling to WebAssembly, an effective fuzzer for WebAssembly must operate on binaries only. As we demonstrate in the motivating example in Section 6.2, the security properties of a native binary compiled from the same source code can differ from the corresponding WebAssembly binary, necessitating fuzzing WebAssembly binaries, and not their native counterparts. Improvements to AFL's input generation, e.g., improved power schedules and many other works [Böhme et al. 2020; A. Herrera et al. 2021; Lemieux and Sen 2018; D. Maier et al. 2020; Mathis, Gopinath, and Zeller 2020], are orthogonal to our work and could be integrated into FUZZM in the future.

BINARY-ONLY FUZZING There are several approaches to fuzz native binaries [Choi et al. 2019; Dinesh et al. 2020; Nagy et al. 2021]. Some rely on QEMU, DynInst, or Pin [Choi et al. 2019; Li et al. 2017; Rawat et al. 2017], i.e., dynamic instrumentation, which can incur a substantial runtime overhead. We instead do reliable, static instrumentation of WebAssembly binaries directly, which does not have this cost. Dinesh et al. [2020] propose static instrumentation of x86-64 binaries for fuzzing, but they cannot handle WebAssembly binaries due to the different architecture. Other recent work is about binary instrumentation for coverage [Nagy et al. 2021], but does not provide an oracle instrumentation similar to ours, and is limited to x86-64 again. Finally, Y. Chen et al. [2019] use special hardware support in Intel CPUs to recover AFL-compatible instrumentation directly from a binary. As Web-

12 Wasmer: https://github.com/wasmerio/wasmer/tree/master/fuzz and Wasmtime: https://github.com/bytecodealliance/wasmtime/tree/main/fuzz

13 https://github.com/google/AFL, https://lcamtuf.coredump.cx/afl/

Assembly binaries are run on many different hardware architectures, it does not apply to our use case. To the best of our knowledge, FUZZM is the first binary-only fuzzer for WebAssembly binaries.

EVALUATION AND BENCHMARKS Klees et al. [2018] outline common issues in previous evaluations of fuzzers. For example, they show that the amount of crashes detected across two runs of the same fuzzer with the same seed may vary drastically, yet many evaluations do not include repeated runs of the experiments. We follow their recommendations on fuzzer evaluation, e.g., by fuzzing repeatedly for 24 hours and reporting means and confidence intervals.

Finally, there are several benchmark programs to evaluate fuzzers on. *LAVA-M* [Dolan-Gavitt et al. 2016] is one such benchmark suite, where artificial bugs are inserted into simple UNIX utilities. It is a very common benchmark, but is also criticized for not being representative of real bugs, e.g., because the bugs are guarded by unrealistic comparisons. This causes fuzzers to overfit onto such types of bugs. A more up-to-date fuzzer benchmark suite is *Magma* [Hazimeh et al. 2020]. It consists of 7 programs with 118 bugs, where known triggering inputs were available only for 45% of all bugs when the paper was written. Unfortunately, all programs in Magma use features that were not (yet) supported by WebAssembly when we evaluated FUZZM, e.g., `setjmp`. For that reason, we use three types of benchmarks for evaluating FUZZM: 3 LAVA-M programs, 7 real-world programs with known vulnerabilities, and 17 binaries from our WASMBENCH dataset without access to their source code.

8.7 (NEURAL) REVERSE ENGINEERING

Finally, we discuss related work for SNOWWHITE, which we introduce in Chapter 7. We discuss reverse engineering and type recovery approaches, also for native code. As SNOWWHITE uses a neural network for prediction, it also relates to the larger field of machine learning applied to code.

TYPE RECOVERY Recovering types from binaries has received significant attention, mostly for x86 binaries. Several approaches aim to recover class hierarchies [Katz, Rinetzky, et al. 2018; Pawlowski et al. 2017] and other kinds of type information [ElWazeer et al. 2013; Katz,

El-Yaniv, et al. 2016; Lee et al. 2011; Mycroft 1999; Noonan et al. 2016], e.g., function types and variable types of varying precision. Caballero and Lin [2016] provide a good overview of approaches until 2016. An elaborate type inference approach for binaries is by Noonan et al. [2016]. They can recover very expressive types, with features such as struct fields, polymorphic functions, and recursive types. This makes their types more expressive than our language. However, as a classical, not learning-based approach, their implementation also comes with significant complexity and is tied to an underlying binary analysis framework. In contrast, our pipeline requires basic disassembly of the binary, as the program representation for the neural network is essentially just a sequence of tokens. On a technical level, their approach also cannot handle WebAssembly binaries.

More recently, several data-driven and *learning-based* type recovery approaches have been proposed [Chua et al. 2017; J. He et al. 2018; A. Maier et al. 2019; Pei et al. 2021]. Those are most closely related to our SNOWWHITE work. Learning-based approaches have several advantages over classical algorithms. For example, learning-based approaches can take statistical patterns into account that are not easily expressed as logical constraints. Non learning-based approaches also often rely on manually tuned heuristics or handwritten rules for certain functions, which is laborious and brittle. We compare the four prior learning-based approaches against each other and against our approach in terms of the set of types that can be predicted. Section 7.3 goes into detail and Table 7.1 gives an overview of the different type language features supported. Section 7.6.2 compares different metrics for a subset of those languages.

REVERSE ENGINEERING Beyond types, other information can also be predicted from a binary to support reverse engineering. *DIRE* [Lacomis et al. 2019] and *Nero* [David et al. 2020] are neural models to predict names of variables and functions, respectively. *Coda* [C. Fu et al. 2019] is a trained model that predicts an AST for code given in a simple assembly language. For WebAssembly specifically, `wasm-decompile` and `wasm2c` from the official WebAssembly Binary Toolkit (WABT)[14] aim to decompile binaries to more readable pseudocode or proper C, respectively. All the above tools and techniques are complementary to recovering types.

14 https://github.com/WebAssembly/wabt

To the best of our knowledge, no prior work addresses the problem of recovering precise, high-level types in WebAssembly binaries. In particular, `wasm2c`, despite the name, does not recover high-level source types. Instead, the low-level WebAssembly types in the binary are merely translated to matching C typedefs. That is, a variable with WebAssembly type `i32` would be translated to a number type, such as `uint32_t` in C, regardless of whether the variable might actually hold a pointer (which is not a separate low-level type in WebAssembly, it just coincides with `i32`, as we discuss in Section 2.2).

TYPE PREDICTION FOR SOURCE LANGUAGES Besides recovering types to help with reverse engineering of binaries, types can also be predicted at the level of source code. In particular dynamically typed languages can benefit from type prediction, e.g., to automatically add optional type annotations. There are several approaches for JavaScript [Hellendoorn, Bird, et al. 2018; Malik et al. 2019; Raychev et al. 2015; Wei et al. 2020], Python [Allamanis, Barr, et al. 2020; Pradel, Gousios, et al. 2020; Z. Xu et al. 2016], and Ruby [Kazerounian et al. 2020]. Most of them focus on how to represent and process the input to a prediction model, e.g., with a recurrent neural network (RNN) over a token sequence [Hellendoorn, Bird, et al. 2018] or a graph neural network (GNN) over a graph representation of the code [Wei et al. 2020]. *TypeWriter* [Pradel, Gousios, et al. 2020] combines neural type prediction and type checking-based validation to make sure the produced annotations are type-correct. Almost all of the above approaches predict types from a fixed set, except for Typilus [Allamanis, Barr, et al. 2020], which represents types as points in a continuous type space. SnowWhite differs from them by representing types as sentences in a type language, which turns type prediction into a sequence prediction task.

NEURAL MODELS OF CODE Deep learning on code is receiving significant interest [Pradel and Chandra 2021] beyond the work discussed above. One important question is how to represent a piece of code, e.g., using AST paths [Alon, Zilberstein, et al. 2019], control flow graphs [Y. Wang et al. 2020], abstract syntax trees [J. Zhang et al. 2019], or as a combination of token sequences and a graph representation of code [Hellendoorn, Sutton, et al. 2020]. Instead of the input representation, our work on SnowWhite focuses on how to represent the type *output* of a predictive model.

Other applications for neural models in software engineering are in the area of code changes [Brody et al. 2020; Hoang et al. 2020] and program repair [Dinella et al. 2020; Gupta et al. 2017], to complete partial code [Alon, Brody, et al. 2019; Kim et al. 2021], or bug detection [Pradel and Sen 2018].

9 CONCLUSIONS AND OUTLOOK

In this final chapter, we recapitulate the high-level contributions of the dissertation and look into possible future research directions.

9.1 SUMMARY OF CONTRIBUTIONS

As we set out in Section 1.3, this dissertation focuses on program analysis of WebAssembly binaries. It provides novel insights, datasets, and techniques to support developers in their practical problems around understanding, optimizing, and improving the security and reliability of WebAssembly applications. Concretely, this dissertation describes our work on the following five research projects:

BINARY SECURITY In Chapter 3, we analyze the WebAssembly language, in particular linear memory, and the (lack of) mitigations in the ecosystem. We find that WebAssembly binaries can be exploited, sometimes even more easily than native binaries compiled from the same source code. Our main insights are that the sole focus on host security is not enough, and that there is perhaps a surprising lack of binary security in WebAssembly. We are the first to comprehensively analyze this understudied aspect of WebAssembly security. We demonstrate its severity with proof-of-concept exploits on different platforms, as well as measurements on real-world binaries. To remedy the situation, we also discuss mitigations at different levels of the stack, from changing the language to direct advice to developers.

WASMBENCH Chapter 4 presents WasmBench, our dataset and empirical study of 8,461 unique WebAssembly binaries. It is the largest dataset of real-world WebAssembly binaries to date, which we collect from a wide range of sources, including package managers, code repositories, and live websites. Our analysis provides several insights, for example, that most WebAssembly binaries are compiled from memory-

unsafe languages, and that WebAssembly has a diverse set of use cases on the Web, overhauling previous warnings that the language is mainly used for cryptomining. We also use binaries from the dataset for testing the binary instrumentation implementation in Chapter 6 and have already seen it being used by other researchers (Section 8.4).

WASABI Chapter 5 provides very practical support for developers with WASABI, our general-purpose dynamic analysis framework. It is the first of its kind for WebAssembly. Its high-level, hook-based JavaScript API simplifies writing dynamic analyses and its static binary instrumentation is easier to maintain than ad-hoc binary patching or modifying a WebAssembly runtime. To solve WebAssembly-specific instrumentation challenges, we devise novel techniques, such as on-demand monomorphization of inserted analysis hooks. WASABI can reliably and efficiently instrument and analyze complex WebAssembly binaries with millions of instructions.

FUZZM In Chapter 6, we introduce FUZZM, the first binary-only grey-box fuzzer for WebAssembly. It addresses some of the previously uncovered security issues by finding and mitigating memory errors in WebAssembly binaries. The lack of built-in oracles, such as guard pages, is met by a binary instrumentation technique that adds heap and stack canaries to WebAssembly binaries. FUZZM also integrates a WebAssembly VM with AFL and the first AFL-compatible coverage instrumentation for WebAssembly binaries. Besides as oracles for the fuzzer, our canary instrumentation can also be used to retroactively harden production binaries against runtime attacks.

SNOWWHITE Finally, Chapter 7 presents SNOWWHITE, our neural approach for recovering high-level types from WebAssembly binaries. It helps developers and reverse engineers to understand this new low-level code format. It is the first learning-based type prediction approach, for any binary format, to feature an expressive type language and formulate the problem as a sequence prediction task. To train the model, we also collect the largest dataset of WebAssembly binaries with debugging information to date.

In summary, the projects in this dissertation show that WebAssembly binaries can be reliably and efficiently analyzed and instrumented. We have made our results, datasets, and tools publicly available (see Sec-

tion 1.6), to foster independent replication and in the hope that others can build on them in future work.

9.2 FUTURE WORK

With our work, we not only answer, but also uncover many more questions to investigate in future work. We begin with concrete improvements for the research projects presented earlier, and conclude with more open-ended ideas.

LANGUAGE EXTENSIONS Our tool implementations of Chapters 5 and 6 currently only handle WebAssembly binaries using the initial version of the language. While this is sufficient for most binaries found in the wild, over time more will be using language extensions, e.g., SIMD instructions for improved performance. Beyond the relatively straightforward engineering required for extending the binary parser, some extensions may also introduce more fundamental research challenges. E.g., the threading proposal introduces shared memory and atomics,[1] which may raise questions with regard to correct concurrent code.

NEURAL DECOMPILATION Our work in Chapter 7 on recovering high-level types from binaries addresses only one of many problems in reverse engineering. Neural networks might also be able to recover other types of source-level information from binaries that are lost during compilation. Given our large dataset of WebAssembly binaries with debug information, we could train neural models to automatically name functions and variables, or to recover idiomatic control-flow, e.g., deciding between `for` and `while` loops.

STATIC ANALYSIS On the one hand, WebAssembly is comparatively well-behaved for a binary format, e.g., disassembly is easy, reliable, and fast. This could enable more robust static analysis of WebAssembly binaries compared to native code. On the other hand, WebAssembly is much more low-level than, for example, Java bytecode. Without managed objects or a class hierarchy, many static analysis problems for WebAssembly are still tough nuts to crack, such as precise static call graph construction or points-to-analysis.

1 https://github.com/WebAssembly/threads

WEBASSEMBLY FOR SFI Chapter 3 shows that WebAssembly's linear memory leaves a lot to be desired in terms of protecting the memory of vulnerable WebAssembly programs. However, the host boundary defined by the language is solid, suggesting WebAssembly as a technology for software fault isolation (SFI). Narayan, Disselkoen, Garfinkel, et al. [2020] sandbox vulnerable libraries in Firefox in this manner. An interesting future direction is to use WebAssembly for SFI in applications that do not have a WebAssembly VM built-in, e.g., through ahead-of-time compilation.

UNIVERSAL BYTECODE Finally, we venture an outlook further into the future. If proponents of WebAssembly are correct, WebAssembly has the potential to become a universal bytecode, not just for web applications, but for portable, performant software in general. If this vision becomes a reality, the actual underlying hardware architecture will become more and more irrelevant for software developers and consumers alike. Conversely, there will be many new avenues for further research, e.g., optimizations on the level of WebAssembly bytecode, compilation to efficient native code, and cross-language interoperability.

BIBLIOGRAPHY

[Abadi et al. 2005]
Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. "Control-Flow Integrity." In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (CCS 05). DOI: 10.1145/1102120.1102165.

[Agrawal and Horgan 1990]
Hiralal Agrawal and Joseph R. Horgan. 1990. "Dynamic Program Slicing." In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (PLDI '90), 246–256. DOI: 10.1145/93542.93576.

[Ahmed et al. 2021]
Toufique Ahmed, Premkumar Devanbu, and Vincent J Hellendoorn. 2021. "Learning Lenient Parsing & Typing via Indirect Supervision." *Empirical Softw. Engg.*, 26, 2, (2021). DOI: 10.1007/s10664-021-09942-y.

[Aleph One 1996]
Aleph One. 1996. "Smashing the Stack for Fun and Profit." *Phrack*, 7, 49, (1996). http://www.phrack.com/issues.html?issue=49&id=14.

[Alimadadi et al. 2018]
Saba Alimadadi, Di Zhong, Magnus Madsen, and Frank Tip. 2018. "Finding Broken Promises in Asynchronous JavaScript Programs." *Proc. ACM Program. Lang.*, 2, OOPSLA, Article 162, (2018). DOI: 10.1145/3276532.

[Allamanis 2019]
Miltiadis Allamanis. 2019. "The Adverse Effects of Code Duplication in Machine Learning Models of Code." In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Onward! 2019), 143–153. DOI: 10.1145/3359591.3359735.

[Allamanis, Barr, et al. 2020]
Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. 2020. "Typilus: Neural Type Hints." In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2020), 91–105. DOI: 10.1145/3385412.3385997.

[Allamanis, Brockschmidt, et al. 2018]
Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. "Learning to Represent Programs with Graphs." In *International Conference on Learning Representations* (ICLR 2018). https://openreview.net/forum?id=BJOFETxR-.

[Alon, Brody, et al. 2019]
Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. "code2seq: Generating Sequences from Structured Representations of Code." In *7th International Conference on Learning Representations* (ICLR 2019). https://openreview.net/forum?id=H1gKYo09tX.

[Alon, Zilberstein, et al. 2019]
Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. "code2vec: Learning Distributed Representations of Code." *Proc. ACM Program. Lang.*, 3, POPL, 40:1–40:29. DOI: 10.1145/3290353.

[Amari et al. 1997]
Shun-ichi Amari, Noboru Murata, Klaus-Rovert Muller, Michael Finke, and Howard Hua Yang. 1997. "Asymptotic Statistical Theory of Overtraining and Cross-Validation." *IEEE Transactions on Neural Networks*, 8, 5, 985–996. DOI: 10.1109/72.623200.

[Andersen and Abella 2004]
Starr Andersen and Vincent Abella. 2004. *Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention.* Retrieved February 14, 2020 from https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-xp/bb457155(v=technet.10).

[Andreasen et al. 2017]
Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. "A Survey of Dynamic Analysis and Test Generation for JavaScript." *ACM Comput. Surv.*, 50, 5, Article 66, (2017), 66:1–66:36. DOI: 10.1145/3106739.

[Andriesse et al. 2016]
Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries." In *25th USENIX Security Symposium (USENIX Security 16)*. (2016), 583–600. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse.

[Anonymous 2001]
Anonymous. 2001. "Once upon a free." *Phrack*, 11, 9, (2001). http://phrack.org/issues/57/8.html.

[Arteaga, Donde, et al. 2020]
Javier Cabrera Arteaga, Shrinish Donde, Jian Gu, Orestis Floros, Lucas Satabin, Benoit Baudry, and Martin Monperrus. 2020. "Superoptimization of WebAssembly Bytecode." In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming*, 36–40. https://doi.org/10.1145/3397537.3397567.

[Arteaga, Malivitsis, et al. 2021]
Javier Cabrera Arteaga, Orestis Floros Malivitsis, Oscar Luis Vera Pérez, Benoit Baudry, and Martin Monperrus. 2021. "CROW: Code Diversification for WebAssembly." In *Proceedings of the Workshop on Measurements, Attacks, and Defenses for the Web* (MADWeb, NDSS 2021). DOI: https://doi.org/10.14722/madweb.2021.23004.

[Backes 2018]
Clemens Backes. 2018. *Liftoff: a new baseline compiler for WebAssembly in V8*. Retrieved April 24, 2022 from https://v8.dev/blog/liftoff.

[Bahdanau et al. 2015]
Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. "Neural Machine Translation by Jointly Learning to Align and Translate." In *3rd International Conference on Learning Representations* (ICLR 2015). http://arxiv.org/abs/1409.0473.

[Ball 1999]
Thomas Ball. 1999. "The Concept of Dynamic Analysis." In *ACM SIGSOFT Software Engineering Notes* 6. Volume 24. Springer-Verlag, 216–234. DOI: 10.1145/318774.318944.

[Bastien 2015]
J.F. Bastien. 2015. *WebAssembly – Going public launch bug*. Retrieved March 31, 2022 from https://github.com/WebAssembly/design/issues/150.

[Ben Khadra et al. 2020]
M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2020. "Efficient Binary-Level Coverage Analysis." In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2020), 1153–1164. DOI: 10.1145/3368089.3409694.

[Bergbom 2018]
John Bergbom. 2018. *Memory safety: old vulnerabilities become new with WebAssembly*. Retrieved June 2, 2020 from https://www.forcepoint.com/sites/default/files/resources/files/report-web-assembly-memory-safety-en.pdf.

[Bleigh 2014]
Michael Bleigh. 2014. *The Once And Future Web Platform*. Retrieved March 30, 2022 from https://techcrunch.com/2014/05/16/the-once-and-future-web-platform/.

[Böhme et al. 2020]
Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. 2020. "Boosting Fuzzer Efficiency: An Information Theoretic Perspective." In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2020), 678–689. DOI: 10.1145/3368089.3409748.

[Böhme et al. 2019]
Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. "Coverage-Based Greybox Fuzzing as Markov Chain." *IEEE Trans. Software Eng.*, 45, 5, 489–506. DOI: 10.1109/TSE.2017.2785841.

[Bond et al. 2007]
Michael D. Bond, Nicholas Nethercote, Stephen W. Kent, Samuel Z. Guyer, and Kathryn S. McKinley. 2007. "Tracking Bad Apples: Reporting the Origin of Null and Undefined Value Errors." In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented*

Programming Systems and Applications (OOPSLA '07), 405–422. DOI: 10.1145/1297027.1297057.

[Brito et al. 2022]
Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. 2022. "Wasmati: An Efficient Static Vulnerability Scanner for WebAssembly." *Computers & Security*, 102745. DOI: 10.1016/j.cose.2022.102745.

[Brody et al. 2020]
Shaked Brody, Uri Alon, and Eran Yahav. 2020. "A Structural Model for Contextual Code Changes." *Proc. ACM Program. Lang.*, 4, OOPSLA, Article 215, (2020). DOI: 10.1145/3428283.

[Bruening et al. 2003]
Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. "An Infrastructure for Adaptive Optimization." In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (CGO '03), 265–275. DOI: 10.5555/776261.776290.

[Burow et al. 2017]
Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. "Control-Flow Integrity: Precision, Security, and Performance." *ACM Comput. Surv.*, 50, 1, 16:1–16:33. DOI: 10.1145/3054924.

[Caballero and Lin 2016]
Juan Caballero and Zhiqiang Lin. 2016. "Type Inference on Executables." *ACM Comput. Surv.*, 48, 4, 65:1–65:35. DOI: 10.1145/2896499.

[P. Chen and H. Chen 2018]
Peng Chen and Hao Chen. 2018. "Angora: Efficient Fuzzing by Principled Search." In *Proceedings of the 2018 IEEE Symposium on Security and Privacy* (SP 2018), 711–725. DOI: 10.1109/SP.2018.00046.

[X. Chen et al. 2015]
Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. "StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries." In *22nd Annual Network and Distributed System Security Symposium* (NDSS 2015). DOI: 10.14722/ndss.2015.23248.

[Y. Chen et al. 2019]
Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. 2019. "PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary." In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (Asia CCS '19), 633–645. DOI: 10.1145/3321705.3329828.

[Z. Chen et al. 2021]
Zimin Chen, Vincent Josua Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. 2021. "PLUR: A Unifying, Graph-Based View of Program Learning, Understanding, and Repair." In *Advances in Neural Information Processing Systems* (NeurIPS 2021). https://openreview.net/forum?id=GEm4o9A6Jfb.

[Choi et al. 2019]
Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. "Grey-Box Concolic Testing on Binary Code." In *Proceedings of the 41st International Conference on Software Engineering* (ICSE '19), 736–747. DOI: 10.1109/ICSE.2019.00082.

[Chua et al. 2017]
Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. "Neural Nets Can Learn Function Type Signatures From Binaries." In *26th USENIX Security Symposium* (USENIX Security 17). (2017), 99–116. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chua.

[Clark 2019]
Lin Clark. 2019. *Standardizing WASI: A system interface to run WebAssembly outside the web.* Retrieved April 12, 2022 from https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

[Clark 2017]
Lin Clark. 2017. *What makes WebAssembly fast?* Retrieved March 31, 2022 from https://hacks.mozilla.org/2017/02/what-makes-webassembly-fast/.

[Cowan et al. 1998]
Crispan Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and

Qian Zhang. 1998. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks." In *Proceedings of the 7th USENIX Security Symposium (USENIX Security 98)*. https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention.

[Dang et al. 2015]
Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. "The Performance Cost of Shadow Stacks and Stack Canaries." In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (ASIA CCS '15), 555–566. DOI: 10.1145/2714576.2714635.

[David et al. 2020]
Yaniv David, Uri Alon, and Eran Yahav. 2020. "Neural Reverse Engineering of Stripped Binaries Using Augmented Control Flow Graphs." *Proc. ACM Program. Lang.*, 4, OOPSLA, 225:1–225:28. DOI: 10.1145/3428293.

[Denis 2018]
Frank Denis. 2018. *WebAssembly doesn't make unsafe languages safe (yet)*. Retrieved May 29, 2020 from https://00f.net/2018/11/25/webassembly-doesnt-make-unsafe-languages-safe/.

[Dinella et al. 2020]
Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. "Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs." In *8th International Conference on Learning Representations* (ICLR 2020). https://openreview.net/forum?id=SJeqs6EFvB.

[Dinesh et al. 2020]
Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization." In *2020 IEEE Symposium on Security and Privacy* (SP 2020), 1497–1511. DOI: 10.1109/SP40000.2020.00009.

[Disselkoen et al. 2019]
Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. 2019. "Position Paper: Progressive Memory Safety for WebAssembly." In *Proceedings of the 8th International*

Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19) Article 4. DOI: 10.1145/3337167.3337171.

[Dolan-Gavitt et al. 2016]
Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. "LAVA: Large-Scale Automated Vulnerability Addition." In *IEEE Symposium on Security and Privacy* (SP 2016), 110–121. DOI: 10.1109/SP.2016.15.

[Duck et al. 2020]
Gregory J. Duck, Xiang Gao, and Abhik Roychoudhury. 2020. "Binary Rewriting without Control Flow Recovery." In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2020), 151–163. DOI: 10.1145/3385412.3385972.

[DWARF 5 Standard]
DWARF Committee. 2017. *DWARF Debugging Information Format – Version 5*. (2017). http://www.dwarfstd.org/doc/DWARF5.pdf.

[ElWazeer et al. 2013]
Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. "Scalable Variable and Data Type Detection in a Binary Rewriter." In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '13), 51–60. DOI: 10.1145/2491956.2462165.

[Enck et al. 2014]
William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." *ACM Trans. Comput. Syst.*, 32, 2, Article 5, (2014), 5:1–5:29. DOI: 10.1145/2619091.

[Ernst 2003]
Michael D Ernst. 2003. "Static and dynamic analysis: Synergy and duality." In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis*. New Mexico State University Portland, OR, 24–27.

[Evans et al. 2020]
Ana Nora Evans, Bradford Campbell, and Mary Lou Soffa. 2020. "Is Rust Used Safely by Software Developers?" In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (ICSE '20), 246–257. DOI: 10.1145/3377811.3380413.

[Feng et al. 2020]
Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." In *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139.

[Flanagan and Freund 2010]
Cormac Flanagan and Stephen N. Freund. 2010. "The RoadRunner Dynamic Analysis Framework for Concurrent Programs." In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (PASTE '10), 1–8. DOI: 10.1145/1806672.1806674.

[Foote 2018]
Jonathan Foote. 2018. *Hijacking the control flow of a WebAssembly program*. Retrieved May 1, 2022 from https://www.fastly.com/blog/hijacking-control-flow-webassembly.

[C. Fu et al. 2019]
Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. "Coda: An End-to-End Neural Program Decompiler." In *Advances in Neural Information Processing Systems*. Volume 32. https://proceedings.neurips.cc/paper/2019/file/093b60fd0557804c8ba0cbf1453da22f-Paper.pdf.

[W. Fu et al. 2018]
William Fu, Raymond Lin, and Daniel Inge. 2018. "TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly." *CoRR*, abs/1802.01050. arXiv: 1802.01050 [cs.CR].

[Gadepalli et al. 2020]
Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. "Sledge: a Serverless-first,

Light-weight Wasm Runtimefor the Edge." In *Proceedings of the 21st International Middleware Conference* (Middleware '20). (2020). DOI: 10.1145/3423211.3425680.

[Gakhokidze 2021]
Anny Gakhokidze. 2021. *Introducing Firefox's new Site Isolation Security Architecture*. Retrieved May 2, 2022 from https://hacks.mozilla.org/2021/05/introducing-firefox-new-site-isolation-security-architecture/.

[Genkin et al. 2018]
Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. 2018. "Drive-By Key-Extraction Cache Attacks from Portable Code." In *Proceedings of the International Conference on Applied Cryptography and Network Security* (ACNS 2018). Springer, 83–102. DOI: 10.1007/978-3-319-93387-0_5.

[Godefroid et al. 2020]
Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. 2020. "Differential Regression Testing for REST APIs." In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2020). (2020), 312–312. DOI: 10.1145/3395363.3397374.

[Goel 2021]
Nishu Goel. 2021. *The Web Almanac by HTTP Archive, Part I, Chapter 2 – JavaScript*. HTTP Archive. Retrieved March 30, 2022 from https://almanac.httparchive.org/en/2021/javascript.

[Gong, Pradel, and Sen 2015]
Liang Gong, Michael Pradel, and Koushik Sen. 2015. "JITProf: Pinpointing JIT-unfriendly JavaScript Code." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 2015), 357–368. DOI: 10.1145/2786805.2786831.

[Gong, Pradel, Sridharan, et al. 2015]
Liang Gong, Michael Pradel, Manu Sridharan, and Koushik Sen. 2015. "DLint: Dynamically Checking Bad Coding Practices in JavaScript." In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (ISSTA 2015), 94–105. DOI: 10.1145/2771783.2771809.

[Groß and Burnett 2022]
Samuel Groß and Amy Burnett. 2022. *Attacking JavaScriptEngines in 2022*. https://saelo.github.io/presentations/offensivecon_22_attacking_javascript_engines.pdf.

[Gu et al. 2016]
Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. "Incorporating Copying Mechanism in Sequence-to-Sequence Learning." In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (ACL 2016), 1631–1640. DOI: 10.18653/v1/P16-1154.

[Guo et al. 2021]
Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. "GraphCodeBERT: Pre-training Code Representations with Data Flow." In *International Conference on Learning Representations* (ICLR 2021). https://openreview.net/forum?id=jLoC4ez43PZ.

[Gupta et al. 2017]
Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. 2017. "DeepFix: Fixing Common C Language Errors by Deep Learning." In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 1345–1351. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603.

[Gurdeep Singh and Scholliers 2019]
Robbert Gurdeep Singh and Christophe Scholliers. 2019. "WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers." In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2019)*, 27–36. DOI: 10.1145/3357390.3361029.

[Ha et al. 2018]
Dongsoo Ha, Wenhui Jin, and Heekuck Oh. 2018. "REPICA: Rewriting Position Independent Code of ARM." *IEEE Access*, 6, 50488–50509. DOI: 10.1109/ACCESS.2018.2868411.

[Haas et al. 2017]
Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. "Bringing the Web up to Speed with WebAssembly." In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2017), 185–200. DOI: 10.1145/3062341.3062363.

[Hall and Ramachandran 2019]
Adam Hall and Umakishore Ramachandran. 2019. "An Execution Model for Serverless Functions at the Edge." In *Proceedings of the International Conference on Internet of Things Design and Implementation* (IoTDI '19), 225–236. DOI: 10.1145/3302505.3310084.

[Hanenberg et al. 2014]
Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes, Éric Tanter, and Andreas Stefik. 2014. "An Empirical Study on the Impact of Static Typing on Software Maintainability." *Empir. Softw. Eng.*, 19, 5, 1335–1382. DOI: 10.1007/s10664-013-9289-1.

[Hazimeh et al. 2020]
Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. "Magma: A Ground-Truth Fuzzing Benchmark." *Proc. ACM Meas. Anal. Comput. Syst.*, 4, 3, Article 49, (2020). DOI: 10.1145/3428334.

[J. He et al. 2018]
Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. 2018. "Debin: Predicting Debug Information in Stripped Binaries." In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (CCS '18), 1667–1680. DOI: 10.1145/3243734.3243866.

[N. He et al. 2021]
Ningyu He, Ruiyi Zhang, Haoyu Wang, Lei Wu, Xiapu Luo, Yao Guo, Ting Yu, and Xuxian Jiang. 2021. "EOSAFE: Security Analysis of EOSIO Smart Contracts." In *30th USENIX Security Symposium (USENIX Security 21)*. https://www.usenix.org/conference/usenixsecurity21/presentation/he-ningyu.

[Hellendoorn, Bird, et al. 2018]
Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. "Deep Learning Type Inference." In *Proceedings of*

the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018), 152–162. DOI: 10.1145/3236024.3236051.

[Hellendoorn, Sutton, et al. 2020]
Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. "Global Relational Models of Source Code." In *8th International Conference on Learning Representations* (ICLR 2020). https://openreview.net/forum?id=B1lnbRNtwr.

[Hennenfent 2020]
Eric Hennenfent. 2020. *Symbolically Executing WebAssembly in Manticore*. Retrieved May 2, 2022 from https://blog.trailofbits.com/2020/01/31/symbolically-executing-webassembly-in-manticore/.

[A. Herrera et al. 2021]
Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. 2021. "Seed Selection for Successful Fuzzing." In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2021), 230–243. DOI: 10.1145/3460319.3464795.

[D. Herrera et al. 2018]
David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. 2018. *WebAssembly and JavaScript Challenge: Numerical program performance using modern browser technologies and devices*. Technical report. Technical report SABLE-TR-2018-2. Montréal, Québec, Canada: Sable Research Group, School of Computer Science, McGill University. http://www.sable.mcgill.ca/publications/techreports/2018-2/techrep.pdf.

[Hickey 2019]
Pat Hickey. 2019. *Announcing Lucet: Fastly's native WebAssembly compiler and runtime*. Retrieved October 30, 2019 from https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime.

[Hickey 2018]
Pat Hickey. 2018. *Edge programming with Rust and WebAssembly*. Retrieved October 30, 2019 from https://www.fastly.com/blog/edge-programming-rust-web-assembly.

[Hilbig et al. 2021]
Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. "An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases." In *Proceedings of the Web Conference 2021* (WWW '21), 2696–2708. DOI: 10.1145/3442381.3450138.

[Hoang et al. 2020]
Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. "CC2Vec: Distributed Representations of Code Changes." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (ICSE '20), 518–529. DOI: 10.1145/3377811.3380361.

[Jangda et al. 2019]
Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code." In *2019 USENIX Annual Technical Conference* (USENIX ATC '19), 107–120. https://www.usenix.org/conference/atc19/presentation/jangda.

[Kaempf 2001]
Michel Kaempf. 2001. "Vudo – An object superstitiously believed to embody magical powers." *Phrack*, 11, 8, (2001). http://phrack.org/issues/57/8.html.

[Karampatsis et al. 2020]
Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. "Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code." In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (ICSE '20), 1073–1085. DOI: 10.1145/3377811.3380342.

[Katz, Rinetzky, et al. 2018]
Omer Katz, Noam Rinetzky, and Eran Yahav. 2018. "Statistical Reconstruction of Class Hierarchies in Binaries." In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '18), 363–376. DOI: 10.1145/3173162.3173202.

[Katz, El-Yaniv, et al. 2016]
Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. "Estimating Types in Binaries Using Predictive Modeling." In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Pro-*

gramming Languages (POPL '16), 313–326. DOI: 10.1145/2837614.2837674.

[Kazerounian et al. 2020]
Milod Kazerounian, Brianna M. Ren, and Jeffrey S. Foster. 2020. "Sound, Heuristic Type Annotation Inference for Ruby." In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (DLS 2020), 112–125. DOI: 10.1145/3426422.3426985.

[Kharraz et al. 2019]
Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. 2019. "Outguard: Detecting In-Browser Covert Cryptocurrency Mining in the Wild." In *Proceedings of the 2019 World Wide Web Conference* (WWW '19), 840–852. DOI: 10.1145/3308558.3313665.

[Kim et al. 2021]
Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. "Code Prediction by Feeding Trees to Transformers." In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering* (ICSE 2021), 150–162. DOI: 10.1109/ICSE43902.2021.00026.

[Kingma and Ba 2015]
Diederik P. Kingma and Jimmy Ba. 2015. "Adam: A Method for Stochastic Optimization." In *3rd International Conference on Learning Representations* (ICLR 2015). http://arxiv.org/abs/1412.6980.

[Klabnik and Nichols 2018]
S. Klabnik and C. Nichols. 2018. *The Rust Programming Language*. https://books.google.de/books?id=lrgrDwAAQBAJ.

[Klees et al. 2018]
George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. "Evaluating Fuzz Testing." In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2018), 2123–2138. DOI: 10.1145/3243734.3243804.

[Klein et al. 2017]
Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. "OpenNMT: Open-Source Toolkit for Neural Machine Translation." In *Proceedings of ACL 2017, System Demonstrations*. (2017), 67–72. https://www.aclweb.org/anthology/P17-4012.

[Konoth et al. 2018]
Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 2018. "MineSweeper: An In-depth Look into Drive-by Cryptocurrency Mining and Its Defense." In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (CCS 2018), 1714–1730. DOI: 10.1145/3243734.3243858.

[Lacomis et al. 2019]
Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. "DIRE: A Neural Approach to Decompiled Identifier Naming." In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (ASE '19), 628–639. DOI: 10.1109/ASE.2019.00064.

[Lauinger et al. 2017]
Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. "Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web." In *Proceedings 2017 Network and Distributed System Security Symposium* (NDSS 2017). DOI: 10.14722/ndss.2017.23414.

[Lawrie et al. 2006]
Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2006. "What's in a Name? A Study of Identifiers." In *14th IEEE International Conference on Program Comprehension* (ICPC '06), 3–12. DOI: 10.1109/ICPC.2006.51.

[Lee et al. 2011]
JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. "TIE: Principled Reverse Engineering of Types in Binary Programs." In *Proceedings of the 2011 Network and Distributed System Security Symposium* (NDSS 2011). https://www.ndss-symposium.org/ndss2011/tie-principled-reverse-engineering-of-types-in-binary-programs.

[Lehmann, Kinder, et al. 2020]
Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. "Everything Old is New Again: Binary Security of WebAssembly." In *Proceedings of the 29th USENIX Security Symposium* (USENIX Secu-

rity 20). (2020), 217–217. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[Lehmann and Pradel 2018]
Daniel Lehmann and Michael Pradel. 2018. "Feedback-Directed Differential Testing of Interactive Debuggers." In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2018). (2018), 610–610. DOI: 10.1145/3236024.3236037.

[Lehmann and Pradel 2022]
Daniel Lehmann and Michael Pradel. 2022. "Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries." In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (PLDI '22). DOI: 10.1145/3519939.3523449.

[Lehmann and Pradel 2019]
Daniel Lehmann and Michael Pradel. 2019. "Wasabi: A Framework for Dynamically Analyzing WebAssembly." In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '19), 1045–1045. DOI: 10.1145/3297858.3304068.

[Lehmann, Torp, et al. 2021]
Daniel Lehmann, Martin Toldam Torp, and Michael Pradel. 2021. "Fuzzm: Finding Memory Bugs through Binary-Only Instrumentation and Fuzzing of WebAssembly." arXiv: 2110.15433 [cs.CR].

[Lemieux and Sen 2018]
Caroline Lemieux and Koushik Sen. 2018. "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage." In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (ASE '18), 475–485. DOI: 10.1145/3238147.3238176.

[Li et al. 2017]
Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. "Steelix: Program-State Based Binary Fuzzing." In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 2017), 627–637. DOI: 10.1145/3106237.3106295.

[JVM Specification]
Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2015. *The Java Virtual Machine Specification, Java SE 8 Edition*. (2015). https://docs.oracle.com/javase/specs/jvms/se8/html/.

[Internet Companies]
List of largest Internet companies. Retrieved March 30, 2022 from https://en.wikipedia.org/wiki/List_of_largest_Internet_companies.

[Luk et al. 2005]
Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation." In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '05), 190–200. DOI: 10.1145/1065010.1065034.

[Luong et al. 2015]
Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. "Effective Approaches to Attention-based Neural Machine Translation." In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (EMNLP 2015), 1412–1421. DOI: 10.18653/v1/D15-1166.

[A. Maier et al. 2019]
Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. "TypeMiner: Recovering Types in Binary Programs Using Machine Learning." In *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings* (Lecture Notes in Computer Science). Volume 11543, 288–308. DOI: 10.1007/978-3-030-22038-9_14.

[D. Maier et al. 2020]
Dominik Maier, Heiko Eißfeldt, Andrea Fioraldi, and Marc Heuse. 2020. "AFL++: Combining Incremental Steps of Fuzzing Research." In *14th USENIX Workshop on Offensive Technologies* (WOOT 20). (2020). https://www.usenix.org/conference/woot20/presentation/fioraldi.

[Maisuradze and Rossow 2018]
Giorgi Maisuradze and Christian Rossow. 2018. "Ret2spec: Speculative Execution Using Return Stack Buffers." In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (CCS '18), 2109–2122. DOI: 10.1145/3243734.3243761.

[Malik et al. 2019]
Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. "NL2Type: Inferring JavaScript function types from natural language information." In *Proceedings of the 41st International Conference on Software Engineering* (ICSE '19), 304–315. DOI: 10.1109/ICSE.2019.00045.

[Manès et al. 2021]
Valentin J.M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. "The Art, Science, and Engineering of Fuzzing: A Survey." *IEEE Transactions on Software Engineering*, 47, 11, 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[Manning et al. 2008]
Christopher D Manning, Hinrich Schütze, and Prabhakar Raghavan. 2008. *Introduction to Information Retrieval.*

[Marek et al. 2012]
Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. "DiSL: A Domain-specific Language for Bytecode Instrumentation." In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development* (AOSD '12), 239–250. DOI: 10.1145/2162049.2162077.

[Mathis, Gopinath, Mera, et al. 2019]
Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. "Parser-Directed Fuzzing." In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2019), 548–560. DOI: 10.1145/3314221.3314651.

[Mathis, Gopinath, and Zeller 2020]
Björn Mathis, Rahul Gopinath, and Andreas Zeller. 2020. "Learning Input Tokens for Effective Fuzzing." In *Proceedings of the 29th ACM*

SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2020), 27–37. DOI: 10.1145/3395363.3397348.

[Mayer et al. 2012]
Clemens Mayer, Stefan Hanenberg, Romain Robbes, Éric Tanter, and Andreas Stefik. 2012. "An Empirical Study of the Influence of Static Type Systems on the Usability of Undocumented Software." In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (OOPSLA '12), 683–702. DOI: 10.1145/2384616.2384666.

[McCallum 2019]
Timothy McCallum. 2019. *Diving into Ethereum's Virtual Machine (EVM): the future of Ewasm.* Retrieved October 30, 2019 from https://hackernoon.com/diving-into-ethereums-virtual-machine-the-future-of-ewasm-wrk32iy.

[McFadden et al. 2018]
Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. 2018. *Security Chasms of WASM.* NCC Group. https://i.blackhat.com/us-18/Thu-August-9/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf.

[Melicher et al. 2018]
William Melicher, Anupam Das, Mahmood Sharif, Lujo Bauer, and Limin Jia. 2018. "Riding out DOMsday: Towards Detecting and Preventing DOM Cross-Site Scripting." In *Proceedings of the 2018 Network and Distributed System Security Symposium* (NDSS 2018). DOI: 10.14722/ndss.2018.23309.

[Møller and Schwartzbach 2021]
Anders Møller and Michael I. Schwartzbach. 2021. *Static Program Analysis.* https://cs.au.dk/~amoeller/spa/.

[Musch et al. 2019a]
Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019a. "New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild." In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (DIMVA 2019). Springer, 23–42. DOI: 10.1007/978-3-030-22038-9_2.

[Musch et al. 2019b]
Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019b. "Thieves in the Browser: Web-Based Cryptojacking in the Wild." In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (ARES '19) Article 4. DOI: 10.1145/3339252.3339261.

[Musiol 2018]
Richard Musiol. 2018. *WebAssembly architecture for Go.* Retrieved April 19, 2022 from https://docs.google.com/document/d/131vjr4DH6JFnb-blm_uRdaC0_Nv3OUwjEY5qVCxCup4.

[Mycroft 1999]
Alan Mycroft. 1999. "Type-Based Decompilation (or Program Reconstruction via Type Reconstruction)." In *Proceedings of the 8th European Symposium on Programming Languages and Systems* (ESOP '99), 208–223. https://link.springer.com/content/pdf/10.1007/3-540-49099-X_14.pdf.

[Nagy et al. 2021]
Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. "Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing." In *30th USENIX Security Symposium* (USENIX Security 21), 1683–1700. https://www.usenix.org/conference/usenixsecurity21/presentation/nagy.

[Narayan, Disselkoen, Garfinkel, et al. 2020]
Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. "Retrofitting Fine Grain Isolation in the Firefox Renderer." In *Proceedings of the 29th USENIX Security Symposium* (USENIX Security 20). (2020), 699–716. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan.

[Narayan, Disselkoen, Moghimi, et al. 2021]
Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean M. Tullsen, and Deian Stefan. 2021. "Swivel: Hardening WebAssembly against Spectre." In *30th USENIX Security Symposium* (USENIX Security 2021), 1433–1450.

https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

[CVE-2018-14550]
National Vulnerability Database – CVE-2018-14550 Detail. Retrieved February 14, 2020 from https://nvd.nist.gov/vuln/detail/CVE-2018-14550.

[Nethercote and Seward 2007]
Nicholas Nethercote and Julian Seward. 2007. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation." In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '07), 89–100. DOI: 10.1145/1250734.1250746.

[Newsome and Song 2005]
James Newsome and Dawn Xiaodong Song. 2005. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software." In *Proceedings of the Network and Distributed System Security Symposium* (NDSS 2005). https://valgrind.org/docs/newsome2005.pdf.

[Nikiforakis et al. 2013]
Nick Nikiforakis, Frank Piessens, and Wouter Joosen. 2013. "HeapSentry: Kernel-Assisted Protection against Heap Overflows." In *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference* (DIMVA 2013). Volume 7967, 177–196. DOI: 10.1007/978-3-642-39235-1_11.

[Niu and Tan 2014]
Ben Niu and Gang Tan. 2014. "Modular Control-Flow Integrity." In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '14), 577–587. DOI: 10.1145/2594291.2594295.

[Niu and Tan 2015]
Ben Niu and Gang Tan. 2015. "Per-Input Control-Flow Integrity." In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (CCS '15), 914–926. DOI: 10.1145/2810103.2813644.

[Noonan et al. 2016]
Matt Noonan, Alexey Loginov, and David Cok. 2016. "Polymorphic Type Inference for Machine Code." In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '16), 27–41. DOI: 10.1145/2908080.2908119.

[Österlund et al. 2020]
Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. "ParmeSan: Sanitizer-guided Greybox Fuzzing." In *29th USENIX Security Symposium* (USENIX Security 20), 2289–2306. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund.

[Pascanu et al. 2013]
Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. "On the Difficulty of Training Recurrent Neural Networks." In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28* (ICML'13), III–1310–III–1318. https://proceedings.mlr.press/v28/pascanu13.pdf.

[Pawlowski et al. 2017]
Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. "MARX: Uncovering Class Hierarchies in C++ Programs." In *24th Annual Network and Distributed System Security Symposium* (NDSS 2017). https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/marx-uncovering-class-hierarchies-c-programs/.

[PaX Team 2002]
PaX Team. 2002. *PaX Address Space Layout Randomization (ASLR).* Retrieved February 7, 2020 from https://pax.grsecurity.net/docs/aslr.txt.

[Pei et al. 2021]
Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. "StateFormer: Fine-Grained Type Recovery from Binaries Using Generative State Modeling." In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on*

the Foundations of Software Engineering (ESEC/FSE '21), 690–702. https://doi.org/10.1145/3468264.3468607.

[Petrov et al. 2012]
Boris Petrov, Martin Vechev, Manu Sridharan, and Julian Dolby. 2012. "Race Detection for Web Applications." In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '12), 251–262. DOI: 10.1145/2254064.2254095.

[Plaskett et al. 2018]
Alex Plaskett, Fabian Beterke, and Georgi Geshev. 2018. *Apple Safari – Wasm Section Exploit.* F-Secure. https://labs.f-secure.com/assets/BlogFiles/apple-safari-wasm-section-vuln-write-up-2018-04-16.pdf.

[Pochat et al. 2019]
Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. "Tranco: A research-oriented top sites ranking hardened against manipulation." In *Proceedings of the 2019 Network and Distributed Systems Security Symposium* (NDSS 2019). DOI: https://dx.doi.org/10.14722/ndss.2019.23386.

[Pradel and Chandra 2021]
Michael Pradel and Satish Chandra. 2021. "Neural Software Analysis." *Communications of the ACM*. To appear. https://arxiv.org/abs/2011.07986.

[Pradel, Gousios, et al. 2020]
Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. 2020. "TypeWriter: Neural Type Prediction with Search-based Validation." In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, 209–220. https://doi.org/10.1145/3368089.3409715.

[Pradel, Schuh, et al. 2015]
Michael Pradel, Parker Schuh, and Koushik Sen. 2015. "TypeDevil: Dynamic Type Inconsistency Analysis for JavaScript." In *Proceedings of the 37th International Conference on Software Engineering - Volume*

1 (ICSE '15), 314–324. http://dl.acm.org/citation.cfm?id=2818754.2818795.

[Pradel and Sen 2018]
Michael Pradel and Koushik Sen. 2018. "DeepBugs: A Learning Approach to Name-Based Bug Detection." *Proc. ACM Program. Lang.*, 2, OOPSLA, Article 147, (2018). DOI: 10.1145/3276517.

[Prasad and Chiueh 2003]
Manish Prasad and Tzi-cker Chiueh. 2003. "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks." In *Proceedings of the General Track: 2003 USENIX Annual Technical Conference* (USENIX ATC 2003), 211–224. https://www.usenix.org/conference/2003-usenix-annual-technical-conference/binary-rewriting-defense-against-stack-based.

[Protzenko et al. 2019]
J. Protzenko, B. Beurdouche, D. Merigoux, and K. Bhargavan. 2019. "Formally Verified Cryptographic Web Applications in WebAssembly." In *2019 IEEE Symposium on Security and Privacy (SP 2019)*, 1256–1274. DOI: 10.1109/SP.2019.00064.

[Rawat et al. 2017]
Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. "VUzzer: Application-aware Evolutionary Fuzzing." In *Proceedings of the 2017 Network and Distributed Systems Security Symposium* (NDSS 2017). Volume 17, 1–14. DOI: 10.14722/ndss.2017.23404.

[Raychev et al. 2015]
Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. "Predicting Program Properties from "Big Code"." In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '15), 111–124. DOI: 10.1145/2676726.2677009.

[Reis 2018]
Charlie Reis. 2018. *Mitigating Spectre with Site Isolation in Chrome*. Retrieved May 2, 2022 from https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html.

[Reiser and Bläser 2017]
Micha Reiser and Luc Bläser. 2017. "Accelerate JavaScript Applications by Cross-compiling to WebAssembly." In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages* (VMIL 2017), 10–17. DOI: 10.1145/3141871.3141873.

[Richards et al. 2011]
Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. "The Eval That Men Do: A Large-Scale Study of the Use of Eval in Javascript Applications." In *Proceedings of the 25th European Conference on Object-Oriented Programming* (ECOOP'11), 52–78. DOI: 10.1007/978-3-642-22655-7_4.

[Robertson et al. 2003]
William K. Robertson, Christopher Krügel, Darren Mutz, and Fredrik Valeur. 2003. "Run-time Detection of Heap-based Overflows." In *Proceedings of the 17th Large Installation Systems Administration Conference* (LISA 2003), 51–60. https://www.usenix.org/conference/lisa-03/run-time-detection-heap-based-overflows.

[Romano, Lehmann, et al. 2022]
Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. 2022. "Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly." In *Proceedings of the 2022 IEEE Symposium on Security and Privacy* (S&P 2022), 1101–1101. DOI: 10.1109/SP46214.2022.00064.

[Romano, Liu, et al. 2021]
Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. 2021. "An Empirical Study of Bugs in WebAssembly Compilers." In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–54.

[Rossberg 2019a]
Andreas Rossberg. 2019a. *Multiple per-module memories for Wasm*. Retrieved February 14, 2020 from https://github.com/WebAssembly/multi-memory.

[Rossberg 2019b]
Andreas Rossberg. 2019b. *Proposal for adding basic reference types*.

Retrieved February 14, 2020 from https://github.com/WebAssembly/reference-types.

[Rossberg et al. 2018]
Andreas Rossberg, Ben L. Titzer, Andreas Haas, Derek L. Schuff, Dan Gohman, Luke Wagner, Alon Zakai, J. F. Bastien, and Michael Holman. 2018. "Bringing the Web up to Speed with WebAssembly." *Commun. ACM*, 61, 12, (2018), 107–115. DOI: 10.1145/3282510.

[Rüth et al. 2018]
Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. 2018. "Digging into Browser-based Crypto Mining." In *Proceedings of the Internet Measurement Conference 2018* (IMC 2018).

[Sen et al. 2013]
Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. "Jalangi: A Selective Record-replay and Dynamic Analysis Framework for JavaScript." In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (ESEC/FSE 2013), 488–498. DOI: 10.1145/2491411.2491447.

[Sennrich et al. 2016]
Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. "Neural Machine Translation of Rare Words with Subword Units." In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 1715–1725. DOI: 10.18653/v1/P16-1162.

[Serebryany 2017]
Konstantin Serebryany. 2017. "OSS-Fuzz - Google's continuous fuzzing service for open source software." *USENIX Security*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany.

[Serebryany et al. 2012]
Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. "AddressSanitizer: A Fast Address Sanity Checker." In *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[Severance 2012]
Charles Severance. 2012. "JavaScript: Designing a Language in 10 Days." *Computer*, 45, 2, (2012), 7–8. DOI: 10.1109/mc.2012.57.

[Seward and Nethercote 2005]
Julian Seward and Nicholas Nethercote. 2005. "Using Valgrind to Detect Undefined Value Errors with Bit-precision." In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (ATC '05), 2–2. DOI: https://dl.acm.org/doi/10.5555/1247360.1247362.

[Shacham et al. 2004]
Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. "On the Effectiveness of Address-Space Randomization." In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*, 298–307. DOI: 10.1145/1030083.1030124.

[Shillaker and Pietzuch 2020]
Simon Shillaker and Peter Pietzuch. 2020. "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing." In *2020 USENIX Annual Technical Conference* (USENIX ATC 20). (2020), 419–433. https://www.usenix.org/conference/atc20/presentation/shillaker.

[Silvanovich 2018]
Natalie Silvanovich. 2018. *The Problems and Promise of WebAssembly*. Retrieved May 29, 2020 from https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html.

[Skolka et al. 2019]
Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. "Anything to Hide? Studying Minified and Obfuscated Code in the Web." In *The World Wide Web Conference* (WWW '19), 1735–1746. DOI: 10.1145/3308558.3313752.

[Slowinski et al. 2012]
Asia Slowinski, Traian Stancescu, and Herbert Bos. 2012. "Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation." In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. (2012), 125–137. https://www.usenix.org/conference/atc12/technical-sessions/presentation/slowinska.

[N. Srivastava et al. 2014]
Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research*, 15, 56, 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html.

[P. Srivastava and Payer 2021]
Prashast Srivastava and Mathias Payer. 2021. "Gramatron: Effective Grammar-Aware Fuzzing." In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2021), 244–256. DOI: 10.1145/3460319.3464814.

[StackOverflow Survey 2021]
StackOverflow Developer Survey – Most popular technologies – Programming, scripting, and markup languages. Retrieved March 31, 2022 from https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language.

[Staicu and Pradel 2018]
Cristian-Alexandru Staicu and Michael Pradel. 2018. "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers." In *27th USENIX Security Symposium* (USENIX Security 18), 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu.

[Stiévenart, Binkley, et al. 2022]
Quentin Stiévenart, Dave Binkley, and Coen De Roover. 2022. "Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries." In *The 44th International Conference on Software Engineering* (ICSE 2022). DOI: 10.1145/3510003.3510070.

[Stiévenart and De Roover 2020]
Quentin Stiévenart and Coen De Roover. 2020. "Compositional Information Flow Analysis for WebAssembly Programs." In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation* (SCAM 2020), 13–24. DOI: 10.1109/SCAM51674.2020.00007.

[Stiévenart and De Roover 2021]
Quentin Stiévenart and Coen De Roover. 2021. "Wassail: a WebAssembly Static Analysis Library." In (ProWeb21). Fifth International

Workshop on Programming Technology for the Future Web. (2021). https://2021.programming-conference.org/home/proweb-2021.

[Szanto et al. 2018]
Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. 2018. "Taint Tracking for WebAssembly." *CoRR*, abs/1807.08349. arXiv: 1807.08349.

[Szekeres et al. 2013]
Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. "SoK: Eternal War in Memory." In *2013 IEEE Symposium on Security and Privacy* (SP 2013). DOI: 10.1109/SP.2013.13.

[Tice et al. 2014]
Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM." In *Proceedings of the 23rd USENIX Security Symposium* (USENIX Security 14), 941–955. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice.

[TIOBE Index 2022]
TIOBE Index for March 2022. Retrieved March 31, 2022 from https://www.tiobe.com/tiobe-index/.

[Tolksdorf et al. 2019]
Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. "Interactive Metamorphic Testing of Debuggers." In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA '19), 273–273. DOI: 10.1145/3293882.3330567.

[Vandevoorde and Josuttis 2002]
David Vandevoorde and Nicolai M. Josuttis. 2002. *C++ Templates*.

[Varda 2018]
Kenton Varda. 2018. *WebAssembly on Cloudflare Workers*. Retrieved October 30, 2019 from https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.

[Varlioglu et al. 2020]
Said Varlioglu, Bilal Gonen, Murat Ozer, and Mehmet Bastug. 2020. "Is Cryptojacking Dead after Coinhive Shutdown?" In *2020 3rd International Conference on Information and Computer Technologies (ICICT)*. IEEE, 385–389. DOI: 10.1109/ICICT50521.2020.00068.

[Vaswani et al. 2017]
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. "Attention is All you Need." In *Advances in Neural Information Processing Systems* (NIPS 2017). Volume 30. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[Wagner 2018]
Luke Wagner. 2018. *Mitigations landing for new class of timing attack*. Retrieved May 2, 2022 from https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/.

[Wagner 2017]
Luke Wagner. 2017. *WebAssembly consensus and end of Browser Preview*. Retrieved March 31, 2022 from https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html.

[Wahbe et al. 1993]
Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. "Efficient Software-Based Fault Isolation." In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (SOSP '93), 203–216. DOI: 10.1145/168619.168635.

[W. Wang et al. 2018]
Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. 2018. "SEISMIC: SEcure In-lined Script Monitors for Interrupting Cryptojacks." In *24th European Symposium on Research in Computer Security* (ESORICS 2018). Springer, 122–142. DOI: https://doi.org/10.1007/978-3-319-98989-1_7.

[Y. Wang et al. 2020]
Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. "Learning Semantic Program Embeddings with Graph Interval Neural Network." *Proc. ACM Program. Lang.*, 4, OOPSLA, Article 137. DOI: 10.1145/3428205.

[Wartell et al. 2011]
Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani M. Thuraisingham. 2011. "Differentiating Code from Data in x86 Binaries." In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases, Part III*

(ECML PKDD 2011). Volume 6913, 522–536. DOI: 10.1007/978-3-642-23808-6_34.

[WASI Website]
WASI – The WebAssembly System Interface. Retrieved April 12, 2022 from https://wasi.dev/.

[Wasmtime Website]
Wasmtime – A small and efficient runtime for WebAssembly & WASI. Retrieved February 14, 2020 from https://wasmtime.dev/.

[Watt 2018]
Conrad Watt. 2018. "Mechanising and Verifying the WebAssembly Specification." In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (CPP 2018), 53–65. DOI: 10.1145/3167082.

[Watt, Rao, et al. 2021]
Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. "Two Mechanisations of WebAssembly 1.0." In *Formal Methods*, 61–79. DOI: 10.1007/978-3-030-90870-6_4.

[Watt, Renner, et al. 2019]
Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. "CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem." *Proc. ACM Program. Lang.*, 3, POPL, Article Article 77, (2019). DOI: 10.1145/3290390.

[Watt, Rossberg, et al. 2019]
Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. "Weakening WebAssembly." *Proc. ACM Program. Lang.*, 3, OOPSLA, Article 133, (2019). DOI: 10.1145/3360559.

[WebAssembly Website]
WebAssembly. Retrieved March 31, 2022 from https://webassembly.org/.

[WebAssembly Specification]
WebAssembly Core Specification. World Wide Web Consortium (W3C). Retrieved March 31, 2022 from https://www.w3.org/TR/wasm-core-1/.

[Wei et al. 2020]
Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. "LambdaNet: Probabilistic Type Inference using Graph Neural Networks." In *International Conference on Learning Representations* (ICLR 2020). https://openreview.net/forum?id=Hkx6hANtwH.

[H. Xu et al. 2021]
Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael R. Lyu. 2021. "Memory-Safety Challenge Considered Solved? An In-Depth Study with All Rust CVEs." *ACM Trans. Softw. Eng. Methodol.*, 31, 1, Article 3, (2021). DOI: 10.1145/3466642.

[X. Xu et al. 2019]
Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. 2019. "CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software." In *Proceedings of the 28th USENIX Security Symposium* (USENIX Security 19), 1805–1821. https://www.usenix.org/conference/usenixsecurity19/presentation/xu-xiaoyang.

[Z. Xu et al. 2016]
Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. 2016. "Python Probabilistic Type Inference with Natural Language Support." In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (FSE 2016), 607–618. DOI: 10.1145/2950290.2950343.

[Yu et al. 2014]
Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. 2014. "Comprehending Performance from Real-world Execution Traces: A Device-driver Case." In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '14), 193–206. DOI: 10.1145/2541940.2541968.

[Zakai 2011]
Alon Zakai. 2011. "Emscripten: An LLVM-to-JavaScript Compiler." In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (OOPSLA '11), 301–312. DOI: 10.1145/2048147.2048224.

[Zakai 2020]
Alon Zakai. 2020. *WasmBoxC: Simple, Easy, and Fast VM-less Sandboxing*. Retrieved April 25, 2022 from https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html.

[Zeller 2003]
Andreas Zeller. 2003. "Program Analysis: A Hierarchy." In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis* (WODA 2003). https://www.st.cs.uni-saarland.de/publications/files/zeller-woda-2003.pdf.

[Zeller et al. 2022]
Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2022. *The Fuzzing Book. Tools and Techniques for Generating Software Tests*. Retrieved April 20, 2022 from https://www.fuzzingbook.org/.

[B. Zhang et al. 2017]
Bin Zhang, Jiaxi Ye, Chao Feng, and Chaojing Tang. 2017. "S2F: Discover Hard-to-Reach Vulnerabilities by Semi-Symbolic Fuzz Testing." In *13th International Conference on Computational Intelligence and Security* (CIS 2017), 548–552. DOI: 10.1109/CIS.2017.00127.

[J. Zhang et al. 2019]
Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. "A Novel Neural Source Code Representation Based on Abstract Syntax Tree." In *Proceedings of the 41st International Conference on Software Engineering* (ICSE '19), 783–794. DOI: 10.1109/ICSE.2019.00086.

[M. Zhang and Sekar 2013]
Mingwei Zhang and R. Sekar. 2013. "Control Flow Integrity for COTS Binaries." In *Proceedings of the 22nd USENIX Security Symposium* (USENIX Security 13), 337–352. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang.

[Zhao et al. 2010]
Qin Zhao, Derek Bruening, and Saman Amarasinghe. 2010. "Umbra: Efficient and Scalable Memory Shadowing." In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (CGO '10), 22–31. DOI: 10.1145/1772954.1772960.

[Zimmermann et al. 2019]
Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. “Small World with High Risks: A Study of Security Threats in the npm Ecosystem.” In *28th USENIX Security Symposium* (USENIX Security 19), 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman.