WebAssembly as a Multi-Language Platform

WebAssembly as a Multi-Language Platform

A THESIS PRESENTED
BY
ALEXANDER R. WENDLAND
TO
THE JOVIAL AND WISE READERS OF
THE DEPARTMENT OF COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MAY 2020

Thesis advisor: Professor Nada Amin Alexander R. Wendland

WebAssembly as a Multi-Language Platform

Abstract

Developers choose languages primarily off of the quality of libraries in their ecosystems. However, what if languages and libraries were orthogonal? What if multiple languages could be seamlessly adopted in a single application, depending on the immediate task at hand? Currently, multi-language interoperability is usually pairwise, with explicit support provided to call from one specific language to another. When it's not, developers usually give up substantial safety guarantees such as type soundness or memory integrity. Yet with ongoing work in modular secure compilation, it should be possible to maintain the integrity of each language's abstractions when interoperating. What is needed for these techniques to enter common software development practice, and to evolve research further, is a sufficiently capable intermediate language that can act as a target for these modular secure compilers.

WebAssembly is a nascent language with wide industry backing and strong security fundamentals. It has strong typing of instructions and function calls, and provides module-based encapsulation of functions and memory—powerful primitives for maintaining the integrity of source-level abstractions. By extending WebAssembly further, through the introduction of a lightweight abstract type system, these abilities are strengthened even more. This thesis will show practical examples of how this extended WebAssembly is sufficient to protect additional abstractions, like object method access, without needing to introduce heavy language features like first-class object support. This thesis speculates, but does not prove, that with this simple abstract types extension WebAssembly can function as a performant target language for a modular secure compiler.

Contents

Acknowledgments

I'd like to extend a deep thank you to the many people that helped me with this thesis and throughout my Harvard Computer Science career. My thesis advisor, Professor Nada Amin, took me on as a last minute thesis project, fed me rosemary french fries from Clover, and dispensed deep-seated wisdom on a broad range of CS topics. Yizhou Zhang and Professor Steve Chong took the time to understand my gesticulated rants and provide critical feedback. Before this thesis endeavor, a bevy of intelligent, kind, committed professors helped me work through a world of CS material, frequently with stand-up comedy level delivery or captivating narratives that you'd expect out of a TED talk for. Thank you to Professors Margo Seltzer (CS 61), Stuart Shieber (CS 51), Todd Zickler (my undergraduate advisor), Michael Mitzenmacher (CS 222), James Mickens (CS 263)*, Minlan Yu (CS 245), Eddie Kohler (CS 260R), and Elena Glassman (CS 279R) for answering countless questions and engaging in many office hours discussions.

I'd also like to thank an invaluable group outside of Harvard's faculty. Harvard's CS Program Coordinator, Beth Musser, created an undergraduate experience that was seamless, fun, and stress-free. Two of my undergraduate peers, Theodore Liu and Garrett Tanzer, deciphered and answered non-sensical questions I sent them at all hours of the night. Luke Wagner from Mozilla was a priceless source of guidance and information on all-things-WebAssembly, and was one of the most rapid, consistent, and thoughtful email respondents I've had the pleasure of interacting with. Andreas Rossberg and Ross Tate wrote extensive WebAssembly proposals and quickly responded to Github issues that I opened. Andreas Haas provided me with a copy of the LaTeXsource for their PLDI '17 paper so that I didn't have to recreate their syntax tables. To everyone who helped me on this journey, thank you!

*Professor Mickens current webpage bio says *"Abstract: In this bleak, relentlessly morbid talk, James Mickens will describe why making computers secure is an intrinsically impossible task.* ***He will explain why no programming language makes it easy to write secure code."*** *(emphasis mine).* Oh well, this thesis was fun to work on anyways.

0

Introduction

Multi-language interoperability is a well-trodden path; across the decades many language execution platforms have approached this problem. Two of the most notable, the Java Virtual Machine and Microsoft's Common Language Infrastructure, have found large measures of success. While initially created for the Java language, the JVM has since been expanded upon and used as a target for many more languages, including such varied paradigms as JRuby[1] and Clojure[2]. Microsoft's Common Language Runtime (CLR) was designed from the start as a multi-language platform[3] and is now employed widely to run C#, C++, and much more. Both have succeeded with massive adoptions and evolution, growing from their initial designs to introduce new features to ease multi-language usage (such as `invokedynamic` on the JVM for languages such as Smalltalk and

Ruby[4] or the Java Native Interface for unmanaged interactions with language such as C[5]).

Both the JVM and CLR aspired to several core principles[4]:

Portability

The JVM had a famous slogan, "Write Once, Run Anywhere"[6] which epitomized its goals of portability. Instead of compiling programs to system-specific assembly, the JVM bytecode and Common Intermediate Language of the CLR offered an opportunity to compile programs once and enable them to execute on any platform. These promises of portability came with restrictions (and escape hatches) for using platform specific instructions, but with the standardization of instruction sets and the expanded usage of JIT compilers it permitted program developers to not have to worry about platform specific details.

Security

The JVM and CLR are both stack-based virtual machines which provide protection from arbitrary memory access and usage of system resources. This virtual machine-based execution model provided increased security on top of the operating system's own process isolation. Furthermore, both platforms performed verification of any bytecode prior to execution in order to ensure that unsafe operations, such as arbitrary memory accesses, were not permitted[7].

Efficiency

Both platforms aspired to be widely leveraged, with an understanding that with more users on the platform more effort would be focused on improving it and therefore more developers would benefit from any incremental changes. Both the JVM and CLR introduced Just-In-Time compilation (JIT) techniques, enabling them to perform on-par with native C programs[8,9]. Any improvements to this JIT engine by the practitioners of one high-level

language would likely benefit the users of other languages on top of the platform.

Interoperability

Potentially the most challenging principle, but also potentially the most useful, was the aspiration for interoperability of higher-level languages on these platforms. By providing core mechanisms for object creation, method invocation, access control, garbage collection, and execution threading, the JVM and CLR anticipated that library reuse would become possible ontop of their platforms between different higher-level languages [10].

0.1 Changing Times

It cannot be overstated how successful these runtimes have been and how much the software engineering world (and the world at large) has benefited from their creation and development. It's on the shoulders of these giants that any future progress towards achieving these principles will be built, and it's also historically been through the obstacle of these existing languages that any new upstart approach has had to overcome.

Over the past several years an interesting new era has begun to emerge. After a decade of torrential growth JVM usage has begun to decline and Microsoft's centralized development of the CLR has landed it on the outside of the broader open-source trends.

Most importantly, the manner in which programs are distributed and consumed looks very different than it did 20-years, or even 10-years, ago. While desktop programs used to dominate the field of executed code, the Web and its plethora of questionably-provinced programs are the new normal. This has forced a need to reimagine what the thresholds for security should be, and how trust (or more importantly, lack of trust) factors into execution.

System architectures have similarly undergone a transition, both narrowing the field of platforms that programs will execute on, while at the same time expanding them drastically. With the demise of PowerPC came the birth of ARM and other low-energy architectures. Meanwhile, new execution models became prevalent with the surgence of the cloud and hardware sharing. Similarly, edge computing has introduced a more pressing need for small, quick-starting program execution.

All-in-all, expectations from the past no longer align with the goals of today. Starting in 2015, browser vendors began removing the ability to execute Java applets (i.e. programs loaded via a website and run on the JVM)[11]. New principles of security—a distrust-first approach to execution; an assumption that any feature, especially those that are undocumented, will be used for exploits—became the norm, and the underpinnings of the JVM and CLR has made their ability to meet these new standards difficult.

Furthermore, with the ongoing march of time, more and more code is getting written for specific languages, with different languages developing different ecosystems for computing tasks. For example, many machine-learning libraries are written in C++ and Python, and plans to switch to different languages for performance and ease-of-development have led to extensive discussion and consternation[12]. A large survey by Meyerovich and Rabkin published in 2013[13] found that,

> Intrinsic features have only secondary importance in adoption. Open source libraries, existing code, and experience strongly influence developers when selecting a language for a project. Language features such as performance, reliability, and simple semantics do not.

What could a future era look like where existing code and libraries wasn't the driving

factor of language adoption? Instead, what if languages were selected due to intrinsic traits and adopted with low switching costs? Interoperability provides one necessary part of this path by allowing developers to treat libraries and languages orthgonally.

0.2 WebAssembly, A New Champion?

WebAssembly[14] was first announced in 2015 and is described as,

> A safe virtual instruction set architecture that can be embedded into a range of host environments, such as Web browsers, content delivery networks, or cloud computing platforms. It is represented as a byte code designed to be just-in-time- compiled to native code on the target platform. Wasm is positioned to be an efficient compilation target for low-level languages like C++.[15]

As can be seen, WebAssembly explicitly targets the first three aforementioned principles of *Portability*, *Security*, and *Efficiency*. It does so through several core design decisions.

Formal Semantics

WebAssembly is based on a normative specification with fully formal semantics. The authors intended to avoid any undefined behavior and subsequent language extensions are required to have a "full formal specification before final adoption"[15]. Mechanized verification work is also ongoing with a 2018 paper by Watt demonstrating a verified implementation of the type checker and interpreter[16].

Dynamic Linking

The core encapsulation unit of a WebAssembly progam is a *module*. WebAssembly modules can contain private private memory, global variables, and functions. Each module defines a set of exports (it can export any of the aforementioned elements) and defines a set

of imports that the host environment must resolve prior to the module's execution. Modules are loaded in three phases: 1.) *Validation*, during which types are checked, 2.) *Instantiation*, during which imports are resolved and type checked, and 3.) *Invocation*, when a function within the module is executed.

Module linking is left to the host environment. At the language level, the import system supports two-level namespacing, with the first level referring to a foreign module and the second level referring to an element to be imported.

Encapsulated Memory

WebAssembly modules can contain private instances of linear memory which is a byte addressable vector of uninterpreted data. This memory is private unless exported, and therefore incorruptible by execution outside of the current module.

Strong Typing

Unlike x86 Assembly Language, WebAssembly is strongly typed. Function signatures are verified before invocation, even when they occur dynamically.

0.2.1 WebAssembly's Future

Arguably most important of all, WebAssembly has the correct political backing. Both the JVM and CLR benefited greatly due to the context in which they were created in (mega-corporations who were defining the software used by millions of other businesses). In addition to it's technical merits, WebAssembly will benefit from a similar context: it is being designed by a working group under the auspice of the W3C with participation from every major browser vendor and many additional leading technology companies[17]. Given this context, it's less of a question of "Will WebAssembly be widely used?" and more of a ques-

tion of "When will WebAssembly be widely used?" and "How can WebAssembly improve the software ecosystem?".

0.3 Contributions

Assuming that WebAssembly did manage to succeed at these aims, it would be on track to compete in the same multi-language field as the JVM and CLR. However, WebAssembly as defined by its v1.0 specification is missing several core features that would be critical for ergonomic and secure multi-language interoperation. Most notably, WebAssembly doesn't provide a mechanism for unforgeable addresses such as abstract types.

In Chapter 1 I'll review the successes and difficulties of using existing intermediate languages for multi-language interoperability as well as the literature on secure compilation to provide motivation for a new solution. In Chapter 2 I'll introduce abstract types into the WebAssembly specification and reference interpreter as a building block for maintaining source-level abstractions. In Chapter 3 I'll present tangible demonstrations of the source-language abstractions that can now be maintained when using WebAssembly as an intermediate language (IL), along with demonstrations of multiple languages interoperating*. In Chapter 4 I'll present additional extensions that should be researched, and I'll discuss how existing, important WebAssembly proposals interplay with abstract types.

Primarily, this thesis contributes a basic implementation of abstract types in WebAssembly and takes a small step down the path of understanding what those abstract types enable. I believe that WebAssembly can be the ideal multi-language platform, and I hope that tangible demonstrations like the ones contained here will help achieve that.

---

*The source code supporting this thesis is located at github.com/awendland/2020-thesis

I created everyone's least favorite Java feature: checked exceptions... but I can live with that.

Jim Waldo (during a dinner discussion in Winthrop on December 3rd, 2019)

1

Historical and Related Work

"As we will see later, the contents of the call stack for execution are not exposed, and thus cannot be directly accessed by a running WebAssembly program, even a buggy or malicious one.". Oh what a wonderful world it would be if this excerpt from Haas et al.'s 2017 WebAssembly introduction [14] was the case for all programs. Fortunately, for many programs this is the case. Since the introduction of the Java Virtual Machine in 1995, and the Common Language Runtime in 2002, buffer overflow vulnerabilities that were prolific in C/C++ and other manual-memory-managed languages became much less common in user code.

Both the JVM and CLR introduced powerful features that enabled higher-level abstractions to be maintained, such as object references which were opaque and obviated buggy

pointer arithmetic. A large world of programs existed in languages outside of the JVM and CLR sweet spot though, and over their lifetimes the JVM and CLR acquired a reputation for security vulnerabilities or proprietaryness[18,19,20,21]. These reputations are of arguable validity, but nevertheless the JVM is no longer used as a platform for web applications[11] and the browser-based Silverlight variant of the CLR was deprecated in 2013[22].

1.1 Java Virtual Machine

The JVM had many compelling safety features. Two in particular are its constraints on control flow and its support for a form of abstract types. For the former, control flow is restricted such that `goto` instructions mandate that "the target address must be that of an opcode of an instruction within the method that contains this goto instruction"[23]. For the latter, `private` modifiers on object fields are enforced at runtime[24], so object references can be used as unforgeable existential types.

Despite these useful constraints, one of the difficulties with the JVM as a universal target platform is its eminent focus on object-oriented paradigms. Compiling non-garbage collected languages to run on the JVM is doable (many projects–such as NestedVM[25], LLJVM[26], GCC-Bridge[27], and others[28]–have done so), but results in a subpar compilation strategy and performance characteristics (most use a large array object to simulate virtual memory). In the world of garbage collected languages though, the JVM shines, with prominently maintained implementations of Ruby[1], Python[29], and Clojure[2].

The Java Native Interface[5] (JNI) was introduced in 1999 to improve interoperability with non-garbage collected languages. The JNI provides bidirectional interoperability between JVM programs and native code through the use of function invocations outside the

JVM or external handles into JVM operations.

However, usage of the JNI can introduce many subtle errors into a program. As stated in the Java SE 7 design documentation [30],

> The JNI does not check for programming errors such as passing in NULL pointers or illegal argument types. [...] The programmer must not pass illegal pointers or arguments of the wrong type to JNI functions. Doing so could result in arbitrary consequences, including a corrupted system state or VM crash.

The JNI's flexibility in these matters makes it difficult to maintain the integrity of source-level abstractions.

Furthermore, because the native code execution occurs outside the JVM, the JVM cannot inline function calls between the two. Therefore, "it would be grossly inefficient to iterate through a Java array and retrieve every element with a function call" [30]. To address this, the JNI "has a nation of 'pinning' so that the native method can ask the VM to pin down the contents of an array", which it then receives a direct pointer to [30].

1.2 Common Language Runtime

The CLR follows a similar narrative to the JVM. Like the JVM, the CLR was developed under the auspice of a single company, Microsoft. However, in 2006 the specifications for the CLR and related frameworks were submitted to the ECMA and ISO standards bodies, and one main alternative implementation called Mono has arisen from it. Since 2014, the governance has expanded to include a steering committee with membership outside of Microsoft [31].

The CLR experiences many of the same limitations as the JVM as a universal target platform. However, several details differ. In 2005 Microsoft introduced C++/CLI which was an ECMA standardized language presented as a "binding between the Standard C++ programming language and the Common Language Infrastructure"[32]. This was a second attempt after the introduction of Managed Extensions for C++ which had "problems where it obscured essential differences, and the design for overloaded syntaxes like [pointers] was both technically unsound and confusing to use"[10].

Notably, many efforts existed around the CLR with the direct goal of supporting language interoperability. The rationale for C++/CLI[10] included the following,

> Windows Vista [...] offers over 10,000 CLI classes for everything from web service programming [...] to the new 3D graphics subsystem [...] and programmers who want to use those features [can] use one of the 20 or so other languages that do support CLI development. Languages that support CLI include COBOL, C#, Eiffel, Java, Mercury, Perl, Python, and others; at least two of these have standardized language-level bindings.

However, as of 2020, the C++/CLI language is intended only for the Windows platform[33]. Despite many portions of the CLR being cross-platform (called .NET Core[34]), the C++/CLI, and other native language implementations, remain non-portable because only a portion of them actually runs on the CLR, other portions are still compiled and executed natively[35].

1.3 PNaCl

PNaCl, or Portable Native Client, was introduced by Google in 2010 as a way to run untrusted native code in a safe, portable manner by compiling source-languages such as C/C++ to LLVM bitcode, which is an architecture independent intermediate language[36]. This bitcode is then translated into an architecture dependent executable to be run in sandbox called NaCl, or Native Client[37].

Several constraints exist in the PNaCl model. Firstly, programs must be deployed as statically linked binaries; they can't depend upon execution-time resolution of their dependencies. Each NaCl program executes in its own address space and must communicate with other components via an interprocess communication channel. Secondly, substantial work was required to make the underlying NaCl sandbox secure, since it is supporting the broad set of x86 instructions which historically permit intra-opcode jumps, call stack overwriting, buffer overflows, and various other threat vectors. A static validation phase occurs prior to execution during which the NaCl binary is disassembled, instructions are checked for safe usage and bounded memory access, and control flow is analyzed. Furthermore, execution sandboxing leverages several types of hardware-based memory isolation, such as 80386 segments on x86, to further enforce memory access constraints. These features are dependent on hardware support though and therefore unavailable on all platforms.

Despite being an impressive feat of security engineering, PNaCl adoption was low and only ever implemented in the Google Chrome browser. In 2016 it was publicly revealed that the PNaCl team had been destaffed at Google[38] and most use cases had be deprecated.

1.4 Asm.js

In 2012, a Mozilla team introduced a "strict subset of JavaScript that can be used as a low-level, efficient target language for compilers" called *asm.js*[39]. Concurrently, a compiler called *emscripten*[40] was released as a way to compile LLVM bitcode into *asm.js*. This JavaScript subset relied on type coercions in the JavaScript language to annotate values with type information and to ensure values originating outside of the source language were marshalled into appropriate types.

asm.js was highly portable since it was executable anywhere that JavaScript was, and it was able to achieve performance that was within 2x of native code for single-threaded computational tasks. *asm.js* targeted a similar use case as PNaCl—performance—and therefore didn't introduce additional features like module encapsulation and unforgeable addresses that would have made it a compelling platform for multi-language interoperation.

1.5 Truffle & GraalVM

Introduced by Oracle in 2012 on top of the GraalVM (a new JVM implementation)[41], Truffle provides a framework for composing multiple languages in a single managed runtime. Truffle and GraalVM were created with the explicit goal of "reus[ing] libraries from Java, R, or Python" (and more) in "polyglot applications with a seamless way to pass values from one language to another"[42]. To do so, Truffle translates source languages into an intermediate representation that is run on top of a shared runtime system[43].

To reduce development time for new languages, Truffle provides a language implementation framework for handling AST construction and interpreting, as well as an object

storage model that language implementors can leverage[44]. Truffle permits safe object field accesses without marshalling by using a dynamic dispatch model where read and write messages are sent, verified, and then responded to. The Truffle just-in-time compiler removes the cost of these foreign object accesses by resolving them at AST compilation time, and as a just-in-time compiler, Truffle is also able to perform cross-language inlining. Additionally, message resolution happens upon first usage and is then cached for performant subsequent access[43].

Truffle even supports low-level languages via an implementation of the LLVM bitcode[45]. However, as of their latest paper in 2016, standard libraries were compiled fully to native code, not left as Truffle executed LLVM bitcode intermediaries, due to performance concerns.

GraalVM's main drawback is that it is dual-licensed by Oracle as paid software (as of April 2020, it costs $18 per processor core per month to use the Enterprise Edition[46]) and is developed privately by Oracle Research Labs without a public standardization process.

1.6 Secure Compilation

Much academic work has been done on the topic of *secure compilation*, i.e. the preservation of *source-level abstractions* from *target-level attacks* once programs have been compiled to a *target language*[*]. Notably, most of this prior work has focused on *full abstraction*, which is the preservation of these source-level abstractions under the condition that the entire

[*]The terms "source-level abstractions" and "target-level attacks" come from Abadi's 1998 paper[47] and Patrignani et al.'s 2019 paper[48]. "source-level abstractions" refers to language features such as Java's `private`, `protected`, `public` field modifiers "target-level attacks" refers to a threat model where the attacker is only constrained by the target language's semantics, not the semantics of the source language.

program is compiled at once[49,50,51]. However, multi-language interoperability is unable to rely on this precondition, since each language will likely require its own compilation pipeline. Therefore, an extension of secure compilation called *modular secure compilation* is the main focus here.

Modular secure compilation provides the foundation for multi-language interoperability because it ensures that once a given module is linked to another, the behavior will be unchanged and the assumptions that the source-language of the module had will still be maintained[52]. Though multi-language interoperability can occur without modular secure compilation guarantees, the chance that subtle, latent bugs will be introduced during interoperation is greatly reduced with a modular secure compiler and therefore developer experience is substantially better.

In their 2016 paper[52], Patrignani et al. review existing secure compilers targeting *protected modular architectures* such as those based on memory layout randomization[53]. They propose that these secure compilers could be extended to support modular compilation, but that the target platforms would require support for "state [...] divided over [...] various protected modules" and "object references [that] can also be shared between some modules and still be unknown to other ones". Combined with support for abstract types, WebAssembly's modular encapsulation provides a foundation that can address 6 of the 8 threats they discuss: *P1. object-id guessing*, *P2. call stack shortcutting*, *P3. mis-typed objects in other modules*, *P4. existance of objects in other modules*, *P6. module id at the target level*, and *P8. object-id shuffling*. The extension of WebAssembly proposed in this thesis would not be able to address *P5. determining arbitrary object type*, and further work is needed to determine if it could performantly address *P7. dynamic dispatch*.

Additional secure compilation work is based on the preservation of types, and therefore runs into difficulties when targetting platforms like x86, LLVM, or JavaScript which lack sufficient type systems. Swamy et al.'s 2014 gradual type-system depends on JS*, a dependently-typed variant of JavaScript[54]. Barthe et al. demonstrate secure compilation with noninterference for a multi-threaded system when targeting a typed assembly language[55]. Baltopoulos et al. showcase a multi-tier web application (with code executing on the server and client) that maintains security in the face of a malicious client and ensures full abstraction by targeting F7, an ML derivative with refinement types[56]. Similar to the theoretical languages used as the underpinnings of these papers, WebAssembly is strongly typed and abstract types provide the foundation to enforce refinement conditions; therefore it could likely serve as a suitable target for these secure compilers.

Furthermore, most approaches to full abstraction have focused on bilateral proofs, from a source language to a target language. Could a proof exist that traversed through an intermediate language first, in order to prove full abstraction between two disparate source languages? This thesis doesn't attempt to showcase that proof, but instead will speculate that if it exists, WebAssembly may make a compelling intermediate language to target in the resulting compilers.

Given the needs laid out by the literature on secure compilation, WebAssembly could be well positioned to serve as the target language in many of these systems, bringing the benefits of currently impractical secure compilers to everyday usage. The next chapters will extend WebAssembly to help support these ends and will provide practical demonstrations of how it meets these requirements.

WebAssembly is great, I only wish it had abstract types.

Donald Knuth (in a dream I had on March 13th, 2020)

2

Extending WebAssembly with Abstract Types

Despite self-identifying as "so far so boring"*, the overview of the formal WebAssembly specification presented by Haas et al. in their 2017 paper, "Bringing the Web up to Speed with WebAssembly" is succinct and informative[14]. It presents a language for computations on a stack machine with,

> [v]alidation rules ... defined succinctly as a type system. This type system is, by design, embarrassingly simple. It is designed to be efficiently checkable in a

*You can find this lovely phrase as the hook at the end of Section 2.1 encouraging readers to keep going.

single linear pass, interleaved with binary decoding and compilation.

These typed, stack-based operations are augmented via a module system in which functions, globals, memories, and other elements can be defined and exported for consumption by other modules. Function calls are type checked, even between modules, and call stacks are hidden. Only structured control flow is supported, so jumps to the middle of instructions or to privately-scoped functions is not possible.

Already, this provides a nice foundation for maintaining the integrity of source-level abstractions against the threat of target-level attacks. However, though Haas et al. discuss the goal of interoperability in their 2017 paper, there is a key omission in these initial designs (emphasis mine):

> It is possible to link multiple modules that have been created by different producers. **However, as a low-level language, WebAssembly does not provide any built-in object model. It is up to producers to map their data types to numbers or the memory.** This design provides maximum flexibility to producers, and unlike previous VMs, does not privilege any specific programming or object model while penalizing others. Though WebAssembly has a programming language shape, it is an abstraction over hardware, not over a programming language.

The principles of this approach align with learnings from the shortcomings of the JVM and CLR, and will likely enable WebAssembly to be a good platform for any widely used source language. However, as it currently stands, this decision leaves a gap in WebAssembly's ability to maintain certain source-level abstractions in a multi-language environment.

In particular, referring to objects across language boundaries cannot be done in an unforgeable manner, and therefore malicious (or poorly configured) modules can corrupt the state of other modules.

Consider the following library written in (pseudo) C++.

```
/* lib01.cpp */
class CandyBag {
  private: int numCandies = 10;
  public:
    takeCandy() {
      this.numCandies--;
    }
}

class TrashCan {
  private: int fillLevel = 0;
  public:
    throwItemAway() {
      this.fillLevel++;
    }
}
```

This library will be consumed by a different language, and to demonstrate the ability to violate the library's source-level abstractions, we'll consume it directly via WebAssembly. Under the v1.0 WebAssembly specification, the eminent way to support the source-level abstraction of *object references* is by using `int32` values as memory addresses.

2.1 `int32` Addresses as Object References

`lib01.cpp` could be compiled to the following (detail elided) WebAssembly[†].

[†] WebAssembly, though defined as a binary format, has a canonical textual representation in the form of S-expressions.

```
(module lib01
  (memory ...)
  (...alloc + other supporting functions...)
  ;; CandyBag methods, the first i32 param represents "this", the
  ;; memory address of the instance to operate on
  (func (export "CandyBag.new") ...)
  (func (export "CandyBag.destruct") (param i32) ...)
  (func (export "CandyBag.takeCandy") (param i32) ...)
  ;; TrashCan methods, with the same "this" semantics
  (func (export "TrashCan.new") ...)
  (func (export "TrashCan.destruct") (param i32) ...)
  (func (export "TrashCan.throwItemAway") (param i32) ...)
)
```

Consider a malicious WebAssembly module *E* that consumes this library and was aware of the `int32` representation being used to refer to different `CandyBag` and `TrashCan` (as opposed to a source-level consumer who respects the object reference abstraction). Two primary integrity violations can arise.

Firstly, module *E* could abuse the isomorphism of the `CandyBag` and `TrashCan` object references (they are isomorphic because they're both `int32`s) to perform unintended operations. Consider a malicious module that holds a reference *C* from `CandyBag.new`. Normally, a `CandyBag` can only have its amount of candy depleted. However, if memory contents for `CandyBag` instances are laid out the same as `TrashCan` instances, then the malicious module could pass *C* to `TrashCan.throwItemAway` which would increment `numCandies`!

This attack could be mitigated by introducing a header in each object's memory structure which identifies the type of the object. Then, a runtime check could occur in each method to verify if the memory address pointed to the right type. However, this introduces memory and computation overhead on every operation.

Secondly, even if module *E* had not constructed a `CandyBag` but had access to `lib01`'s exported functions, they could arbitrarily create an `int32` value and pass it to the `CandyBag.takeCandy` function. This would let module *E* eat someone else's candy!

Assuming that the memory being used was notably less than what an `int32` could address, several of the `int32`'s bits could be used as a unique tag. This tag could be stored in each object's memory structure and compared upon derefencing. This mitigation is probabilistic, so if 6 of the 32 bits were dedicated to the tag then the attacker has a 1 in $2^6 = 64$ chance of getting it right. A variation of this approach is pursued by LLVM via Hardware-assisted AddressSanitizers[57] where they find a 2x slowdown in CPU and 6% overhead in memory.

These are the same issues that have deeply plagued C/C++ for decadees; the pointers are forgeable and the mitigations are insufficient or costly.

2.2 Abstract Types: Unforgeable References + More

Creating truly unforgeable references requires additional assistance from the language. One of the ways to do this through **abstract types** (also known as *existential types* or *abstract data types*)[58]. Abstract types are a feature in many languages, from Ada to the ML family, since they provide a valuable function: they can create nominal constructs around a hidden underyling type.

For example, consider this demo modeled from the OCaml manual:

```
(* lib.ml *)
module Date = sig
  type date (* public, abstract type *)
end =
struct
  type date = {day : int;  month : int;  year : int} (* private, concrete type *)
  val create : ?days:int -> ?months:int -> ?years:int -> unit -> date
  val yearsBetween : date -> date -> int
  val month : date -> int
  ...
end

(* consumer.ml *)
let kjohnson_bday : Date.date = Date.create 8 26 1918 () in
let mercury_launch : Date.date = Date.create 2 20 1962 () in
let kj_age_at_launch = Date.yearsBetween kjohnson_bday mercury_launch in ...
(* kjognson_bday.day <- this access is invalid *)
```

The author of `lib.ml` would be free to change the underlying type of `Date.date` (such as using an `int` to represent milliseconds since the Unix epoch instead of the current record type) and the author of `consumer.ml` would be none-the-wiser. As an abstract type, `Date.date`'s actual representation is opaque to any users outside the source module.

2.2.1 Adding Abstract Types to WebAssembly

The WebAssembly specification comes with a reference interpreter that is closely modeled off of the operand and value stack-machine described in the WebAssembly specification, making it a convenient WebAssembly implementation to build upon for language experimentation.

(raw value types) $t ::= \text{i32} \mid \text{i64} \mid \text{f32} \mid \text{f64}$
(value types) $ta ::= t \mid \text{abstype\_ref } inst\ i$
(packed types) $tp ::= \text{i8} \mid \text{i16} \mid \text{i32}$
(function types) $tf ::= ta^* \to ta^*$
(global types) $tg ::= \text{mut}^?\ ta$

$unop_{iN} ::= \textbf{clz} \mid \textbf{ctz} \mid \textbf{popcnt}$
$unop_{fN} ::= \textbf{neg} \mid \textbf{abs} \mid \textbf{ceil} \mid \textbf{floor} \mid \textbf{trunc} \mid \textbf{nearest} \mid \textbf{sqrt}$
$binop_{iN} ::= \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div\_}sx \mid \textbf{rem\_}sx \mid \textbf{and} \mid \textbf{or} \mid \textbf{xor} \mid \textbf{shl} \mid \textbf{shr\_}sx \mid \textbf{rotl} \mid \textbf{rotr}$
$binop_{fN} ::= \textbf{add} \mid \textbf{sub} \mid \textbf{mul} \mid \textbf{div} \mid \textbf{min} \mid \textbf{max} \mid \textbf{copysign}$
$testop_{iN} ::= \textbf{eqz}$
$relop_{iN} ::= \textbf{eq} \mid \textbf{ne} \mid \textbf{lt\_}sx \mid \textbf{gt\_}sx \mid \textbf{le\_}sx \mid \textbf{ge\_}sx$
$relop_{fN} ::= \textbf{eq} \mid \textbf{ne} \mid \textbf{lt} \mid \textbf{gt} \mid \textbf{le} \mid \textbf{ge}$
$cvtop ::= \textbf{convert} \mid \textbf{reinterpret}$
$sx ::= \textbf{s} \mid \textbf{u}$

(instructions) $e ::= \textbf{unreachable} \mid \textbf{nop} \mid \textbf{drop} \mid \textbf{select} \mid$
$\textbf{block } tf\ e^*\ \textbf{end} \mid \textbf{loop } tf\ e^*\ \textbf{end} \mid \textbf{if } tf\ e^*\ \textbf{else } e^*\ \textbf{end} \mid$
$\textbf{br } i \mid \textbf{br\_if } i \mid \textbf{br\_table } i^+ \mid \textbf{return} \mid \textbf{call } i \mid \textbf{call\_indirect } tf \mid$
$\textbf{get\_local } i \mid \textbf{set\_local } i \mid \textbf{tee\_local } i \mid \textbf{get\_global } i \mid$
$\textbf{set\_global } i \mid t.\textbf{load } (tp\_sx)^?\ a\ o \mid t.\textbf{store } tp^?\ a\ o \mid$
$\textbf{current\_memory} \mid \textbf{grow\_memory} \mid t.\textbf{const } c \mid$
$t.unop_t \mid t.binop_t \mid t.testop_t \mid t.relop_t \mid t.cvtop\ t\_sx^?$

(abstypes) $abs ::= ex^*\ \textbf{abstype\_new } ta \mid ex^*\ \textbf{abstype\_sealed } im$
(functions) $f ::= ex^*\ \textbf{func } tf\ \textbf{local } t^*\ e^* \mid ex^*\ \textbf{func } tf\ im$
(globals) $glob ::= ex^*\ \textbf{global } tg\ e^* \mid ex^*\ \textbf{global } tg\ im$
(tables) $tab ::= ex^*\ \textbf{table } n\ i^* \mid ex^*\ \textbf{table } n\ im$
(memories) $mem ::= ex^*\ \textbf{memory } n \mid ex^*\ \textbf{memory } n\ im$
(imports) $im ::= \textbf{import } \text{“}name\text{”}\ \text{“}name\text{”}$
(exports) $ex ::= \textbf{export } \text{“}name\text{”}$
(modules) $m ::= \textbf{module } abs^*\ f^*\ glob^*\ tab^?\ mem^?$

Figure 2.1: Changes to the WebAssembly syntax (as defined by Haas et al. 2017) to support abstract types are colored in red

Figure 2.1 shows the abstract syntax changes that were made to the WebAssembly v1.0 specification in order to support abstract types. The changes are built around a modification to the core value type, which was extended to include an *abstract type reference*. These *abstract type references* refer to abstract types constructed by the `abstype_new` operation.

When used inside its originating module, an *abstract type reference* is “unwrapped” to its underlying *raw value type*. In that manner, new abstract types do not affect local module type checking. However, when imported by another module via the `abstype_sealed` operation, these *abstract type reference* are now opaque and act as nominal identifiers structurally composed of a reference to the originating module instance and the `abstype_new` operation which created it.

Type checking occurs in three phases in WebAssembly:

The first phase, called *Validation*[‡], operates on a *module definition* and ensures that *module fields* are well-formed and that *instructions* consume and create appropriate types on the stack. By resolving new local *abstract type references* to their underlying *raw value types*, most *instructions* are unaffected by these language changes. However, in the *Validation* phase imported *abstract type reference* created via the `abstype_sealed` operation are unable to be fully resolved and are therefore represented an an intermediate local form that is uniquely identified by the `abstype_sealed` operation that created it. These are type checked like any other *raw value type* and will therefore cause type errors when used with most *instructions*.

The second phase, called *Instantiation*, executes the *module definition* to create a *module instance*. During this process, imports are linked and *abstract type reference* created via the `abstype_sealed` operation are fully resolved. In this phase, *abstract type reference* within function types and global types are compared between *module instances* to ensure that they represent the same abstract type.

The third phase, called *Invocation*, executes computation within the *module instances*. Most operations no longer perform type checking during this phase, since all types have already been validated, however, the `call_indirect` instruction still needs to perform type checking due to its dynamic operations. The `call_indirect` allows callees to invoke functions dynamically registered in a *table*. The types of these functions can be heterogeneous, and therefore, to ensure sound function invocations, their types must be checked dynamically at runtime.

As anticipated, no additional type checking phases had to be added in order to support

[‡]These phases are defined in the WebAssembly v1.0 specification under the "Semantic Phases" section.

abstract types, and the same one-pass linear approach was sufficient to check them. Approximately 625 lines of source code were changed to support abstract types, with the bulk of changes required in the parser definition to introduce the *abstract type reference* wrapping around the *raw value types*[§]. Furthermore, besides the `call_indirect` instruction, abstract types are a zero-runtime-cost abstraction.

Figure 2.2 showcases what a S-expression WebAssembly script implementing similar functionality to the previous OCaml demo would look like with these extensions. An alternative syntax which makes greater use of existing operator names and may be more user friendly is included in the Appendix in Section A.1.

[§]These changes can be found at github.com/awendland/webassembly-spec-abstypes. The stated line counts do not include changes to the binary encoding or decoding procedures, or the JS translations.

```
(module $lib_date
  (export "Date" (abstype_new $Date i32))
  (func (export "createDate")
    (param $day i32) (param $month i32) (param $year i32)
    (result (abstype_new_ref $Date))
    (i32.add ;; Day, Mon, Year -> Unix milliseconds
      (i32.mul (local.get $day) (i32.const 86400))
      (i32.add
        (i32.mul (local.get $month) (i32.const 2592000))
        (i32.mul (i32.const 31557600)
          (i32.sub (local.get $year) (i32.const 1970))))))
  )
  (func (export "yearsBetweenDates") (param (abstype_new_ref $Date))
    (param (abstype_new_ref $Date)) (result i32)
    (i32.sub (local.get 0) (local.get 1))
    (i32.div_s (i32.const 31557600))
  )
)
(register "lib_date" $lib_date)

(module $main
  (import "lib_date" "Date" (abstype_sealed $Date))
  (import "lib_date" "createDate" (func $createDate
    (param i32) (param i32) (param i32)
    (result (abstype_sealed_ref $Date))))
  (import "lib_date" "yearsBetweenDates" (func $yearsBetweenDates
    (param (abstype_sealed_ref $Date))
    (param (abstype_sealed_ref $Date)) (result i32)))
  (func (export "main") (result i32)
    (call $createDate
      (i32.const 2) (i32.const 20) (i32.const 1962))
    (call $createDate
      (i32.const 8) (i32.const 26) (i32.const 1918))
    (call $yearsBetweenDates)
  )
)
(assert_return (invoke $main "main") (i32.const 43))
```

Figure 2.2: Sample textual WebAssembly in S-expression format with the abstract types extension. *This sample is a poor implementation for a date library because it incorrectly assumes that 1 year is always 31,557,600,000 milliseconds. Do not use it.*

Then they will call on me, but I will not answer; They will seek me diligently but they will not find me.

Proverbs 1:28 (most likely discussing the benefits of abstract types)

3

Maintaining the Integrity of Source-Level Abstractions

"Unfortunately, most target languages cannot preserve the abstractions of their source-level counterparts". This observation comes from Patrignani et al. in their 2019 survey of formal approaches to secure compilation[48]. Fortunately, the strongly typed underpinnings of WebAssembly and this paper's proposed abstract type extension provide a solid foundation for supporting many common source-level abstractions. What follows is a practical, non-proof based demonstration of maintaining the integrity of several widespread source-level abstractions.

To begin, we'll cover some demonstrations of more direct source-level abstractions in

order to lay the groundwork for more indirect ones. Notably, many intermediate languages already fail to maintain the integrity of any of these abstractions. Outside of the first one, the focus behind these examples is to show use cases that are currently undocumented in the WebAssembly test suite[59] and to highlight the rich capabilities that WebAssembly (with a few extensions) provides.

3.1 Type Safe Function Calls

Type safe function calls is an abstraction that supports several other important abstractions identified by Patrignani et al. such as *integrity of values* and *well-bracketed control flow*[48]. *Type safe function calls* ensure that functions are invoked with the correct number of arguments and with the correct type of arguments, to avoid issues seen in un/weakly-typed assemblies (such as x86 or LLVM bitcode) which allow call stack manipulation.

Consider the following C++ and Rust partial programs that a user wants to interoperate:

```cpp
/* demo01_m1.cpp */
bool isEven(int a) {
  return a % 2 == 0;
}
```

```rust
/* demo01_m2.rs */
extern "WASM" {
 pub fn isEven(a: i32) -> bool;
}
pub fn main() -> bool {
 return isEven(4) == true; // assert
}
```

These partial programs would compile to the following (shortened) WebAssembly:

```
(module $demo01_m1
  (func $isEven (export "isEven") (param i32)
    (result i32) ;; i32 is bool (0=false, 1=true)
    (i32.rem_u (local.get 0) (i32.const 2))
    (i32.const 0)
    (i32.eq))
)
(register "demo01_m1" $demo01_m1)

(module $demo01_m2
  (type (func (param i32) (result i32))) ;; 0
  (import "demo01_m1" "isEven" (func $isEven (type 0)))
  (func $main (export "main") (result i32)
    (i32.const 4)
    (call $isEven)
    (i32.eq (i32.const 1 (;true;))))
)
(register "demo01_m2" $demo01_m2)

;; Test that the linked program works
(assert_return (invoke $demo01_m2 "main") (i32.const 1 (;true;)))
```

Had the type definition on line 12 been changed, such as adding an additional `(param i32)`, the WebAssembly would fail during *Instantiation*. In addition to *type safe function calls*, this demo showcases dynamic linking between modules and the use of foreign functions.

3.2 (Somewhat) First Class Functions

One feature of *first class functions* abstraction is the ability to pass functions as arguments to other functions. This enables dynamic program composition. Consider the the follow-

ing example where a user wants to interoperate a Rust partial program, a Zig partial program, and the same C++ partial program (now called "$demo02_m1") from above.

```cpp
/* lib.cpp */
bool isEven(int a) {
  return a % 2 == 0;
}
```

```zig
/* demo02_m2.zig */
const const num: i32 = 53
export fn test_num(pred: fn(i32) -> bool) -> bool {
 return pred(elem);
}
```

```rust
/* demo02_m3.rs */
extern "WASM" {
 pub fn test_num(pred: &dyn Fn(i32) -> bool) -> bool;
 pub fn isEven(a: i32) -> bool;
}
pub fn main() -> bool {
 return test_num(isEven);
}
```

This compilation to WebAssembly requires more cooperation than the previous example, since all the Rust and Zig compilers need to agree on an interface for passing function instances around. In the following WebAssembly, both compilers have chosen to use a heterogeneous function table maintained by the module receiving the function. Additionally, the process for indexing into this table must be standardized by the two modules. Consider a compiled version of the Rust module first:

```
1  (module $demo02_m3
2    (type (func (param i32) (result i32))) ;; 0
3    ;; standard foreign function imports
4    (import "demo02_m1" "isEven" (func $isEven (type 0)))
5    (import "demo02_m2" "test_num" (func $test_num (type 0)))
6    ;; first-class function handling
7    (import "demo02_m2" "_fns" (table $m2_fns 1 funcref))
8    (import "demo02_m2" "_fns_slot" (func $m2_fns_slot (result i32)))
9    (import "demo02_m2" "_fns_free" (func $m2_fns_free (param i32)))
10   (elem declare func $isEven $isEven) ;; register the func for dynamic export
11   ;; main
12   (func $main (export "main") (result i32) (local i32 i32)
13     (local.set 0 (call $m2_fns_slot)) ;; request a slot to pass the func via
14     (table.set $m2_fns (local.get 0) (ref.func $isEven)) ;; pass the pred func
15     (local.set 1 (call $test_num (local.get 0)) ;; call the primary foreign func
16     (call $m2_fns_free (local.get 0)) ;; cleanup the predicate func
17     (local.get 1)) ;; return the result
18 )
19 (register "demo02_m3" $demo02_m3)
20 (assert_return (invoke $demo02_m3 "main") (i32.const 0)) ;; assert -> false
```

The module is importing and making use of a set of functions for managing `$demo02_m2`'s function table. These functions are provided by `$demo02_m2` since it owns the table. We can see these functions in operation in the body of `$main`. On line 13 the first function is being used to determine an index that the passed function should be registered in. On line 14 the function is registered in the table at that index. On line 15 the primary function is called and the index that the passed function is registered in is provided as an argument, and the result is stored. On line 16 the table slot used to pass the function is freed. Finally, on line 17 the result is returned.

For this demo, naive implementations of `_fns_slot` and `_fns_free` have been provided, because there is only one function that could be passed (so index 0 is always used). A more

complete implementation could use `loop` and `table.get` operations to determine the first available index.

```
(module $demo02_m2
  (global $num i32 (i32.const 53))
  (type (;0;) (func (param i32) (result i32)))
  ;; create a table which the predicate function will be provided through
  (table $fns (export "_fns") 1 funcref)
  ;; get the index of the slot to register a func in
  (func (export "_fns_slot") (result i32) (i32.const 0))
  ;; free up the slot
  (func (export "_fns_free") (param $slot i32)
    (table.set $fns (local.get $slot) (ref.null)))
  (func $test_num (export "test_num") (result i32)
    (global.get $num)
    ;; call the predicate fn
    (call_indirect $fns (type 0) (i32.const 0)))
)
(register "demo02_m2" $demo02_m2)
```

This WebAssembly implementation depend on v1.0 features and the reference types[60] proposal (which introduces the ability to dynamically set table values at runtime using the `table.set` instruction, not just at module instantiation using the `elem (table ...) ...)` instruction).*. Importantly, functions can be passed around using only the features of v1.0 WebAssembly and this extension.

However, this operation may become even easier in the future with acceptance of the function references[61] proposal which would make functions fully first class values that can be passed as arguments directly.

*The reference types proposal is one of the more developed WebAssembly proposals and is treated as a dependency for several future proposals.

3.3 Closures

Using the capabilities of the previous example, it's possible to construct closures over functions. Currently, each closure would have to be its own module instance, since module instances provide the only way to retain context when a function is passed to another module. An example has been elided for to reduce redundancy.

3.4 Object Methods

With the abstract types extension, the integrity of *object methods* become easy to maintain. In particular, the *self-application invariant* discussed by League et al. in their 2003 work for a type-preserving Java compiler[62] is now possible to maintain, as seen in Chapter 2. In addition, *object fields* would be represented as getter and setter methods (e.g. a `year : int` field in the `Date` object would be retrieved via `Date.getYear` and set via `Date.setYear`), the performance implications of which will be discussed in Chapter 4.

A further example has been elided to reduce redundancy.

3.5 Field Access Modifiers

Building on the representation of *object methods*, enforcing `public` and `private` field modifiers is straightforward. Methods that are private are not exported from the module, while methods that are public are exported. An example is elided.

3.6 Representation Invariants

Consider a C++ class for rational numbers that has the *representation invariant* of always being in the most simplified form.

```
/* demo05_m1.cpp */
class RationalNum {
  private:
    int _num, _den;
  public:
    RationalNum(int num, int den) {
      int gcd = std::gcd(num, den);
      _num = num / gcd;
      _den = den / gcd;
    }
}
```

Combined with WebAssembly's memory encapsulation support, abstract types enable partial programs to maintain these invariants. If the C++ program was compiled to the following WebAssembly, there is no way for a target-level attacker to violate the invariant since they are unable to directly view or manipulate the underlying `num` or `den` values, they are unable to overflow the `RationalNum` constructor's call stack to replace the `gcd` return value, and they can't modify the function pointer for the `std::gcd` function being invoked.

```
(module $demo03_m1
  (import "std" "gcd" (func $_std_gcd (param i32) (param i32) (result i32)))
  (memory 1)
  (func $_malloc (param i32) (result i32) ...)
  (abstype_new $RationalNum i32)
  ;; RationalNum struct = {int, int} = 4 + 4 = 8 bytes
  (func (export "RationalNum.new") (param $num i32) (param $den i32)
    (result (abstype_new_ref $RationalNum)) (local $gcd i32) (local $adr i32)
```

```
    (local.set $gcd (call $_std_gcd (local.get $num) (local.get $den)))
    (local.set $adr (call $_malloc (i32.const 8)))
    (i32.store offset=0 (local.get $adr)
      (i32.div_s (local.get $num) (local.get $gcd)))
    (i32.store offset=1 (local.get $adr)
      (i32.div_s (local.get $num) (local.get $gcd)))
    (local.get $adr))
  (func (export "RationalNum.getNumerator")
    (param $this (abstype_new_ref $RationalNum)) (result i32)
    (i32.load offset=0 (local.get $this)))
  ...
)
```

C is quirky, flawed, and an enormous success.

Dennis M. Ritchie

4

Future & Ongoing Work

The best practical demonstration to show language and library orthogonality would be to arbitrarily pick a library in one language and consume it with a different language. We are not there just yet, but we are getting close. Many languages are adding support for WebAssembly as a compiler output, though usually these efforts are focused on single-module designs and do not target multi-module systems. This is already improving though with projects like Rust's JavaScript bindings through wasm-bindgen[63] and JavaScript bindings in Emscripten (a C/C++ to WebAssembly compiler)[64]. However, most of these efforts are primarily focused on WebAssembly-JavaScript interactions, not arbitrary-language to arbitrary-language interactions.

A core issue with these endeavors is the lack of a good ABI (application binary interface).

Currently, the aforementioned approaches either support exposing functions using a direct translation of the C calling convention, or they create bespoke interfaces for JavaScript to use. Unfortunately, the C calling convention is too feature light to maintain most source-level abstractions, and the JavaScript API isn't generalizable to other languages.

What's needed is a WebAssembly-native ABI. This WebAssembly ABI will have first class enforcement for type, mechanisms to refer to opaque values (like this thesis's abstract types), and conventions around how to performantly share contiguous chunks of memory. Notably, this ABI won't come from WebAssembly itself, since WebAssembly identifies as "an abstraction over hardware, not over a programming language"[14]. Instead, it will be a convention on top of WebAssembly. Compiler authors will be able to target this ABI from (most) any language and be able to interoperate with the ecosystem of other languages that also target the ABI.

However, before an ABI like this can arise, several additional features need to be added to WebAssembly, in addition to the abstract types extension proposed here. Before discussing those features though, a quick detour needs to be taken to discuss performance and the architecture of production WebAssembly runtimes.

4.1 Performance Concerns

Most WebAssembly runtimes* are based on a just-in-time compilation (JIT) structure where WebAssembly bytecode is converted directly into native instructions, as opposed to being interpreted. These JIT-based runtimes also perform additional performance op-

*Some popular WebAssembly runtimes with JIT architectures are Firefox's Spidermonkey[65], Chrome's V8[66], and the Bytecode Alliance's wasmtime[67].

timizations in addition to a first-pass translation to native instructions, such as function inlining[68] which replaces a function call site with the full function body.

A major benefit of an abstract types based approach to object references is that once the function calls to the object's methods have been type checked during the *Validation* and *Instantiation* phases, no more type checking needs to occur. Furthermore, due to the constraints in WebAssembly's stack machine operations, no more checks need to occur at runtime to ensure call stacks aren't corrupted or that source-module invariants are violated.

Therefore, object method calls are likely ripe for getting inlined by JITs, removing much of the overhead that comes with accessing an object's fields via function-based getters and setters. As seen with the Truffle framework, in many cases these operations should be able to be converted into single-machine-instruction load operations[43].

However, things get more complicated if `call_indirect` instructions are used, since they functions being called may be dynamically provided and therefore need to be type checked at runtime before they can be safely invoked. Techniques exist to improve performance on these as well though. In particular, JavaScript's dynamic nature has forced its runtimes to address this problem for the past decade[69].

Furthermore, the work of Li et al.[70] with runtime optimizations of dynamically linked C++ modules and Barrett et al.[71] with cross-language optimizations between Prolog and Python has shown that cross-module/language function inlining is viable and effective.

4.2 Additional Extensions

If JIT function inlining operated perfectly for abstract type-based object method calls, it might be possible to represent contiguous arrays of bytes efficiently using WebAssembly

v1.0 and an abstract type extension. The approach might expose an interface that looks something like this (see Appendix Section A.2 for a more complete implementation):

```
(module $lib_buffer
  (abstype_new $Buffer i32) ;; a sequence of bytes
  (func $Buffer.create (param $size i32) (result (abstype_new_ref $Buffer)))
  (func $Buffer.size (param $this (abstype_new_ref $Buffer)) (result i32) ...)
  (func $Buffer.i32_load (param $this (abstype_new_ref $Buffer))
            (param $idx i32) (result i32) ...)
  (func $Buffer.i32_load8_u (param $this (abstype_new_ref $Buffer))
            (param $idx i32) (result i32) ...)
  (func $Buffer.i32_store (param $this (abstype_new_ref $Buffer))
            (param $idx i32) (param $data i32) ...)
  (func $Buffer.i32_store8 (param $this (abstype_new_ref $Buffer))
            (param $idx i32) (param $data i32) ...)
  (abstype_new $ReadonlyBuffer (abstype_new_ref $Buffer)) ;; readonly view
  (func $ReadonlyBuffer.from (param $super (abstype_new_ref $Buffer))
            (result (abstype_new_ref $ReadonlyBuffer) ...)
  (func $ReadonlyBuffer.i32_load (param $this (abstype_new_ref $Buffer))
            (param $idx i32) (result i32) ...)
  (func $ReadonlyBuffer.i32_load8_u (param $this (abstype_new_ref $Buffer))
            (param $idx i32) (result i32) ...)
)
```

The implementation of `$Buffer.i32_load` might simply be `i32.load (i32.add (local.get $this) (local.get $idx))`, the same as any array access compiled from C/C++. With function inlining, the overhead of wrapping this operation in a function call is removed, and the executed instructions would be the same as if the array's memory was local to the calling module.

However, the non-deterministic nature of the JIT may cause people to avoid adopting this pattern, since without function inlining there is likely to be notable overhead. Language extensions may be necessary to annotate when function inlining must occur, in order to maintain a more determenistic performance profile.

4.2.1 Multi-Memory

There is also a multi-memory proposal[72] for WebAssembly that could provide the necessary primitives to improve this situation. With multiple memories a module could still maintain a private memory area that was uncorruptable by foreign modules. It could then create public memories were data that was intended for consumption by foreign functions could be stored.

Yet, even with multiple memories, some complications still remain. Firstly, once a memory is exported any other module can consume it. The exporting module can't restrict the consumer of the memory to only be the foreign function it's calling. At the moment this problem is solvable due to WebAssembly's single threaded nature (after calling the trusted foreign function the callee module can wipe the memory), but once multi-threading is added to WebAssembly this problem reemerges.

4.2.2 Strings

Related to the sharing of contiguous arrays, string sharing between modules is also problematic in v1.0 WebAssembly. Currently, it's up to each source-language compiler to determine how strings will be represented and encoded. A common method today is to store them in linear memory in a given encoding (such as UTF-8) and pass around a pair of `i32` values representing the memory address and the string's length.

A similar approach with abstract types or multi-memory could be taken, but encoding variations still remain. Providing first class support for a standard encoding, such as UTF-8, would ease multi-language interoperability but would also cause WebAssembly to stray into defining a language instead of its initial goals of only being a hardware abstraction.

4.2.3 Standard Library

If an abstract types based approach was taken to contiguous memory (e.g. the `$Buffer` example above), for modules to interop they would all need to use the same `$Buffer` abstract type. If different modules adopted different abstract types to represent these byte-sequences they would be incompatible.

Therefore, a standard library would need to be created with a core set of abstract types that users could consume and which would provide a common abstraction for languages to interoperate using.

4.2.4 Parameterized Modules

The abstract type system proposed in this thesis doesn't support generic data structures. For example, if one library module provided a binary tree abstract type `$BinaryTree` with appropriate methods for adding and retrieving from it, and another module library module provided a string abstract type, a third module would be unable to insert strings it had created into the binary tree. This is because the abstract type system currently defined does not support dynamic import of abstract types, since import dependencies can only go one way.

The system could be expanded to support a semantics like OCaml's functors, which would allow copies of a given module to be instantiated with different imports provided to it. Each module instance would then be able to operate on a specific abstract type (e.g. an instance of the binary tree module that works for strings).

This introduces a new concern though: modules needing the ability to direct the instantiation of other modules. Currently, all module instantiation is directed via the host

runtime. Fortunately, this likely wouldn't increase any existing threat vectors for system resource depletion (memory size is already controllable by modules today), but it would impact the module instance namespacing system that is currently in use (in v1.0 WebAssembly, module names are the same as module instance names, since only one module instance can exist per module).

4.2.5 Structural Typing?

Alternatively to the standard library approach proposed above, a structural typing model could be adopted instead. This would entail more direct support for the object methods abstraction discussed earlier, with the WebAssembly language needing to be aware of the functions that were attached to a given abstract type.

With structural typing and an ability to define traits (Rust terminology) or interfaces (Java/Go terminology), an module could provide a byte-sequence implementation; they wouldn't have to all agree with the implementation decisions of the standard library. Instead, the universal standard library would encompass a set of traits that standardized a common interface abstraction for any language that wants to interoperate.

4.2.6 Dynamic Objects

So far, these examples have all focused on objects with stable properties (methods and fields). However, many languages, such as Python and JavaScript, support objects with dynamic properties which should be supported in a performant manner. The work behind this thesis has yet to engage with the optimizations required to efficiently handle these dynamic object accesses, however, the hope is that abstract types provide the means to con-

struct a stable, well-formed interface behind which these optimizations could be transparently implemented.

4.3 Interplay with Other WebAssembly Proposals

There are three major projects underway in the WebAssembly working group as of April 2020 (in addition to a smattering of orthogonal or smaller updates).

4.3.1 Garbage Collection

Garbage collection[73] is a broad proposal that is in heavy flux at the moment as it is broken up into smaller pieces and alternative paths. For example, the SOIL Initiative has proposed[74] an alternative to the original working group's proposal. The original proposal was based more around structural typing, while the SOIL proposal is based around nominal typing. The nominal typing approach is similar in many ways to the abstract types proposed by this thesis.

Considering the garbage collection proposals more broadly, their focus is on "providing access to industrial-strength GCs"[73] which are readily available thanks to the JavaScript runtimes that most WebAssembly runtimes are embedded within. The goal is for source-language compilers to not have to worry about implementing their own garbage collector on top of WebAssembly, and instead to be able to use an existing WebAssembly garbage collector implementation, thus reducing module size and development time and improving performance.

Abstract types complement this endeavor nicely. Because abstract type are opaque references, consumers don't need to worry whether the underlying representation is in lin-

ear memory or is of a GC object. Module authors can seamlessly switch between memory models without impacting their downstream consumers.

4.3.2 Interface Types

Interface Types[75] is a proposal to introduce higher-level values (like strings, byte-sequences, and more) into the WebAssembly ecosystem in a standardized way. Notably, the proposed semantics are not a change to core WebAssembly, instead they are intended to be layered on top of the existing semantics. Interface types are concerned with many of the same problems relating to interoperability that this thesis was motivated by, however, they are primarily focused on data representation and the conversion of data when interoperating between different modules (or a WebAssembly module and a JavaScript host environment, which is a primary motivating use case for the proposal).

Interface types would benefit from an abstract type primitive in the core WebAssembly semantics, and the vision this thesis hopes WebAssembly will achieve is served well through the layered semantics provided by interface types.

4.3.3 WASI

Currently, WebAssembly exists primarily as a computational platform with minimal access to the outside world. Today, programs address this restriction through bespoke host injected functions that provide I/O operations. One of the primary barriers to broader usage is a lack of a standardized system call interface for WebAssembly programs to leverage. WASI (WebAssembly System Interface)[76] is a proposal to standardize this interface.

With its concerted focus on principles of least authority and attenuated access controls,

the proposal introduces a need for unforgeable data types for uses such as file descriptors (forgeable data types would allow malicious modules to violate intended security constraints). Abstract types provide the necessary foundation to represent these unforgeable values and maintain capability-based invariants between modules with different levels of system access.

Since WASI is provided by the host environment, the abstract types that WASI provides would come from the host environment as well. This would require a small modification to the abstract type semantics given in Chapter 2, since these abstract types would not be identified by a module instance, but instead by the host. Since only one host is possible, a special indicator for a "host instance" could be added to the set of module instance identifiers.

5

Conclusion

The future is multi-language, and the work being done today is important for determining the shape of the platform that will underlie these future programs. With modest extensions, WebAssembly provides a capable foundation for building this platform. Its semantics provide many of the features needed in an intermediate-language for modular secure compilation, and its runtime architectures provide efficient execution models even in cross-language operations. WebAssembly has the right political backing and the benefit of two decades of hindsight about multi-language runtimes; if its designers choose to, it could be the platform of the multi-language future.

A

Expanded Code Samples

A.1 Alternative Abstract Type Syntax

This alternative syntax overloads existing operators (primarily `type`) in ways that should be intuitive to users, but reduces explicit clarity on the semantic meaning of each operator. If abstract types were implemented as presented in this thesis, it's likely that this style of syntax would be used instead of the syntax used throughout the rest of this work.

```
(module $lib_date
  (export "Date" (newtype $Date i32))
  (func (export "createDate")
    (param $day i32) (param $month i32) (param $year i32) (result (type $Date))
    (i32.add
      (i32.mul (local.get $day) (i32.const 86400))
      (i32.add
        (i32.mul (local.get $month) (i32.const 2592000))
```

```
        (i32.mul (i32.const 31557600)
          (i32.sub (local.get $year) (i32.const 1970))
    )))
  )
  (func (export "yearsBetweenDates")
    (param (type $Date)) (param (type $Date)) (result i32)
    (i32.sub (local.get 0) (local.get 1))
    (i32.div_s (i32.const 31557600))
  )
)
(register "lib_date" $lib_date)

(module $main
  (import "lib_date" "Date" (type $Date))
  (import "lib_date" "createDate" (func $createDate
    (param i32) (param i32) (param i32) (result (type $Date))))
  (import "lib_date" "yearsBetweenDates" (func $yearsBetweenDates
    (param (type $Date)) (param (type $Date)) (result i32)))
  (func (export "main") (result i32)
    (call $createDate
      (i32.const 2) (i32.const 20) (i32.const 1962))
    (call $createDate
      (i32.const 8) (i32.const 26) (i32.const 1918))
    (call $yearsBetweenDates)
  )
)
```

A.2 Array Library

The name "$Buffer" is adopted from the Node.js nomeclature for byte-sequences. Method names, such as "$Buffer.i32_load8_u", are based on the underlying instruction that they wrap. Further operations to reduce address arithmetic within each method are likely possible, but have not been explored here.

```
(module $lib_buffer
  (memory 1)
```

```
  (global $nextAddr (mut i32) (i32.const 0))
  (abstype_new $Buffer i32) ;; a sequence of bytes
  (func $Buffer.create (param $size i32) (result (abstype_new_ref $Buffer))
    (local i32)
    (local.set 1 (global.get $nextAddr))
    (i32.store (local.get 1) (local.get $size))
    (global.set $nextAddr
      (i32.add (local.get 1)
        (i32.add (local.get $size) (i32.const 4))))
    (local.get 1))
  (func $Buffer.size (param $this (abstype_new_ref $Buffer)) (result i32)
    (i32.load (local.get 0)))
  (func $Buffer.i32_load (param $this (abstype_new_ref $Buffer))
    (param $idx i32) (result i32)
    (i32.add (i32.add (local.get $this) (i32.const 4)) (local.get $idx))
    (i32.load))
  (func $Buffer.i32_load8_u (param $this (abstype_new_ref $Buffer))
    (param $idx i32) (result i32)
    (i32.add (i32.add (local.get $this) (i32.const 4)) (local.get $idx))
    (i32.load8_u))
  (func $Buffer.i32_store (param $this (abstype_new_ref $Buffer))
    (param $idx i32) (param $data i32)
    (i32.add (i32.add (local.get $this) (i32.const 4)) (local.get $idx))
    (local.get $data)
    (i32.store))
  (func $Buffer.i32_store8 (param $this (abstype_new_ref $Buffer))
    (param $idx i32) (param $data i32)
    (i32.add (i32.add (local.get $this) (i32.const 4)) (local.get $idx))
    (local.get $data)
    (i32.store))
  (abstype_new $ReadonlyBuffer (abstype_new_ref $Buffer))
  (func $ReadonlyBuffer.fromBuffer (param $super (abstype_new_ref $Buffer))
    (result (abstype_new_ref $ReadonlyBuffer))
    (local.get 0))
  (func $ReadonlyBuffer.i32_load (param $this (abstype_new_ref $Buffer))
    (param $idx i32) (result i32)
    (call $Buffer.i32_load (local.get $this) (local.get $idx)))
  (func $ReadonlyBuffer.i32_load8_u (param $this (abstype_new_ref $Buffer))
    (param $idx i32) (result i32)
    (call $Buffer.i32_load8_u (local.get $this) (local.get $idx)))
)
```

References

[1] Charles O. Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 2011.

[2] Rich Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1–1, 2008.

[3] Erik Meijer and John Gough. Technical overview of the common language runtime. *language*, 29:7, 2001.

[4] John R. Rose. Bytecodes meet combinators: invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages - VMIL '09*, page 1–11. ACM Press, 2009. ISBN 978-1-60558-874-2. doi: 10.1145/1711506.1711508. URL http://portal.acm.org/citation.cfm?doid=1711506.1711508.

[5] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999. ISBN 978-0-201-32577-5.

[6] Write once, run anywhere?, May 2002. URL https://www.computerweekly.com/feature/Write-once-run-anywhere.

[7] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, Jul 2006. ISSN 0164-0925. doi: 10.1145/1146809.1146811.

[8] Peter Sestoft. Numeric performance in c, c# and java. page 14, Feb 2010.

[9] Performance comparison of java/.net runtimes (oct 2004), Nov 2005. URL http://www.shudo.net/jit/perf/.

[10] Herb Sutter. A design rationale for c++/cli. *gotwa.ca*, Feb 2006.

[11] Dalibor Topic. Moving to a plugin-free web, Jan 2016. URL https://blogs.oracle.com/java-platform-group/moving-to-a-plugin-free-web.

[12] Dan Zheng. tensorflow/swift | whyswiftfortensorflow.md, Apr 2018. URL https://github.com/tensorflow/swift.

[13] Leo A. Meyerovich and Ariel S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, OOPSLA '13, page 1–18. Association for Computing Machinery, Oct 2013. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509515. URL https://doi.org/10.1145/2509136.2509515.

[14] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200. Association for Computing Machinery, Jun 2017. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062363. URL https://doi.org/10.1145/3062341.3062363.

[15] Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. Weakening webassembly. Association for Computing Machinery, Oct 2019. URL https://doi.org/10.1145/3360559.

[16] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 53–65. Association for Computing Machinery, Jan 2018. ISBN 978-1-4503-5586-5. doi: 10.1145/3167082. URL https://doi.org/10.1145/3167082.

[17] W3c webassembly working group. URL https://www.w3.org/wasm/.

[18] Lowell Heddings. Java is insecure and awful, it's time to disable it, and here's how, Oct 2012. URL https://www.howtogeek.com/122934/java-is-insecure-and-awful-its-time-to-disable-it-and-heres-how/.

[19] Insecure deserialization, Apr 2019. URL https://hdivsecurity.com/bornsecure/insecure-deserialization-attack-examples-mitigation/.

[20] Seth Puckett. 5 myths about .net, Nov 2016. URL https://www.excella.com/insights/5-myths-about-net.

[21] By David Ramel. Microsoft doubles down on open source .net – redmondmag.com, Dec 2018. URL https://redmondmag.com/articles/2018/12/04/microsoft-open-source-net.aspx.

[22] Jonathan Allen. What to use on the microsoft stack, Nov 2013. URL https://www.infoq.com/articles/Microsoft-Stack-2013/.

[23] Java se 8 - jvm - chapter 6. the java virtual machine instruction set, Mar 2018. URL https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html.

[24] Java se 8 - jvm - chapter 5. loading, linking, and initializing, Mar 2018. URL https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html#jvms-5.4.3.2.

[25] Nestedvm, Aug 2009. URL http://nestedvm.ibex.org/.

[26] David A. Roberts. *davidar/lljvm*. Nov 2010. URL https://github.com/davidar/lljvm.

[27] Alex Bertram. Renjin | introducing gcc-bridge: A c/fortran compiler targeting the jvm, Jan 2016. URL https://www.renjin.org/blog/2016-01-31-introducing-gcc-bridge.html.

[28] Other languages for the java vm, Jan 2013. URL https://wiki.c2.com/?OtherLanguagesForTheJavaVm.

[29] Jython. URL https://www.jython.org/.

[30] Java se 7 - jni design overview, Jul 2011. URL https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/design.html.

[31] Darryl Taft. .net foundation adds new governance model, projects, Nov 2014. URL https://www.eweek.com/development/net-foundation-adds-new-governance-model-projects.

[32] Standard ecma-372_2005, Dec 2005. URL https://www.ecma-international.org/publications/standards/Ecma-372.htm.

[33] Jan Kotas. Will coreclr support c++/cli crossplat? · issue #4116 · dotnet/runtime, Jan 2020. URL https://github.com/dotnet/runtime/issues/4116#issuecomment-580865237.

[34] .NET Platform, Apr 2020. URL https://github.com/dotnet/runtime.

[35] Native and .net interoperability, Nov 2016. URL https://docs.microsoft.com/en-us/cpp/dotnet/native-and-dotnet-interoperability.

[36] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. Pnacl: Portable native client executables. *Google White Paper*, 2010.

[37] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, page 79–93, May 2009. doi: 10.1109/SP.2009.25.

[38] Raymes Khoury. 239656 - refactor nacl integration to eliminate the trusted, in-process plugin. - chromium, Oct 2016. URL https://bugs.chromium.org/p/chromium/issues/detail?id=239656#c160.

[39] David Herman, Luke Wagner, and Alon Zakai. asm.js, Aug 2014. URL http://asmjs.org/spec/latest/.

[40] Alon Zakai. *emscripten-core/emscripten*. emscripten-core, Nov 2012. URL https://github.com/emscripten-core/emscripten.

[41] Christian Wimmer and Thomas Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, SPLASH '12, page 13–14. Association for Computing Machinery, Oct 2012. ISBN 978-1-4503-1563-0. doi: 10.1145/2384716.2384723. URL https://doi.org/10.1145/2384716.2384723.

[42] Graalvm documentation. URL https://www.graalvm.org/docs/reference-manual/polyglot/.

[43] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. High-performance cross-language interoperability in a multi-language runtime. In *Proceedings of the 11th Symposium on Dynamic Languages - DLS 2015*, page 78–90. ACM Press, 2015. ISBN 978-1-4503-3690-1. doi: 10.1145/2816707.2816714. URL http://dl.acm.org/citation.cfm?doid=2816707.2816714.

[44] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, PPPJ '14, page 133–144. Association for Computing Machinery, Sep 2014. ISBN 978-1-4503-2926-2. doi: 10.1145/2647508.2647517. URL https://doi.org/10.1145/2647508.2647517.

[45] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. Bringing low-level languages to the jvm: efficient execution of llvm ir on truffle. In *Proceedings of the 8th International Workshop on*

Virtual Machines and Intermediate Languages, VMIL 2016, page 6–15. Association for Computing Machinery, Oct 2016. ISBN 978-1-4503-4645-0. doi: 10.1145/2998415.2998416. URL https://doi.org/10.1145/2998415.2998416.

[46] Oracle graalvm global price list. Jul 2019. URL https://www.oracle.com/a/ocom/docs/corporate/pricing/graalvm-price-list.pdf.

[47] Martín Abadi. Protection in programming-language translations. In *In Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, page 868–883. Springer-Verlag, 1998.

[48] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 51(6):125:1–125:36, Feb 2019. ISSN 0360-0300. doi: 10.1145/3280984.

[49] Martín Abadi. *Protection in programming-language translations*, page 19–34. Springer-Verlag, Jun 2001. ISBN 978-3-540-66130-6.

[50] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to javascript. Jan 2013. URL https://doi.org/10.1145/2480359.2429114.

[51] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. Association for Computing Machinery, Apr 2015. URL https://doi.org/10.1145/2699503.

[52] Marco Patrignani, Dominique Devriese, and Frank Piessens. On modular and fully-abstract compilation. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, page 17–30, Jun 2016. doi: 10.1109/CSF.2016.9.

[53] Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. Jul 2012. URL https://doi.org/10.1145/2240276.2240279.

[54] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 425–437. Association for Computing Machinery, Jan 2014. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535889. URL https://doi.org/10.1145/2535838.2535889.

[55] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. Jul 2010. URL https://doi.org/10.1145/1805974.1805977.

[56] Ioannis G. Baltopoulos and Andrew D. Gordon. Secure compilation of a multi-tier web language. In *Proceedings of the 4th international workshop on Types in language design and implementation*, TLDI '09, page 27–38. Association for Computing Machinery, Jan 2009. ISBN 978-1-60558-420-1. doi: 10.1145/1481861.1481866. URL https://doi.org/10.1145/1481861.1481866.

[57] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory tagging and how it improves c/c++ memory safety. *arXiv preprint arXiv:1802.09517*, 2018.

[58] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(3):470–502, Jul 1988. ISSN 0164-0925, 1558-4593. doi: 10.1145/44501.45065.

[59] *WebAssembly/spec/test/core*. WebAssembly, Apr 2020. URL https://github.com/WebAssembly/spec.

[60] Webassembly/reference-types, . URL https://github.com/WebAssembly/reference-types.

[61] Webassembly/function-references, . URL https://github.com/WebAssembly/function-references.

[62] Christopher League, Zhong Shao, and Valery Trifonov. Precision in practice: A type-preserving java compiler. In Görel Hedin, editor, *Compiler Construction*, Lecture Notes in Computer Science, page 106–120. Springer, 2003. ISBN 978-3-540-36579-2.

[63] *rustwasm/wasm-bindgen*. Rust and WebAssembly, Apr 2020. URL https://github.com/rustwasm/wasm-bindgen.

[64] Interacting with code — emscripten 1.39.11 documentation. URL https://emscripten.org/docs/porting/connecting_cpp_and_javascript/Interacting-with-code.html.

[65] mozilla/gecko-dev, . URL https://github.com/mozilla/gecko-dev.

[66] v8/v8, . URL https://github.com/v8/v8.

[67] *bytecodealliance/wasmtime*. Bytecode Alliance, Apr 2020. URL https://github.com/bytecodealliance/wasmtime.

[68] Ariya Hidayat. Automatic inlining in javascript engines, Apr 2013. URL https://ariya.io/2013/04/automatic-inlining-in-javascript-engines.

[69] Andy Wingo. v8: a tale of two compilers, Jul 2011. URL http://wingolog.org/archives/2011/07/05/v8-a-tale-of-two-compilers.

[70] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. CGO '10, page 53–61. Association for Computing Machinery, Apr 2010. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772964. URL https://doi.org/10.1145/1772954.1772964.

[71] Edd Barrett, Carl Friedrich Bolz, and Laurence Tratt. Unipycation: a case study in cross-language tracing. In *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, VMIL '13, page 31–40. Association for Computing Machinery, Oct 2013. ISBN 978-1-4503-2601-8. doi: 10.1145/2542142.2542146. URL https://doi.org/10.1145/2542142.2542146.

[72] *WebAssembly/multi-memory*. WebAssembly, Feb 2020. URL https://github.com/WebAssembly/multi-memory.

[73] Webassembly/gc, . URL https://github.com/WebAssembly/gc.

[74] *soil-initiative/gc*. SOIL Initiative, Jan 2020. URL https://github.com/soil-initiative/gc.

[75] Webassembly/interface-types, . URL https://github.com/WebAssembly/interface-types.

[76] *WebAssembly/WASI*. WebAssembly, Apr 2020. URL https://github.com/WebAssembly/WASI.

This thesis was typeset using LaTeX, originally developed by Leslie Lamport and based on Donald Knuth's TeX. The body text is set in 11 point Egenolff-Berner Garamond, a revival of Claude Garamont's humanist typeface. Thank you, Jordan Suchow, for providing the MIT X11 licensed Dissertate template that this document was based off of.