WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly

CONRAD WATT, University of Cambridge, UK
MAJA TRELA, University of Cambridge, UK and Jane Street, UK
PETER LAMMICH, University of Twente, Netherlands
FLORIAN MÄRKL, Technical University of Munich, Germany

We present WasmRef-Isabelle, a monadic interpreter for WebAssembly written in Isabelle/HOL and proven correct with respect to the WasmCert-Isabelle mechanisation of WebAssembly. WasmRef-Isabelle has been adopted and deployed as a fuzzing oracle in the continuous integration infrastructure of Wasmtime, a widely used WebAssembly implementation. Previous efforts to fuzz Wasmtime against WebAssembly's official OCaml reference interpreter were abandoned by Wasmtime's developers after the reference interpreter exhibited unacceptable performance characteristics, which its maintainers decided not to fix in order to preserve the interpreter's close definitional correspondence with the official specification. With WasmRef-Isabelle, we achieve the best of both worlds — an interpreter fast enough to be useable as a fuzzing oracle that also maintains a close correspondence with the specification through a mechanised proof of correctness.

We verify the correctness of WasmRef-Isabelle through a two-step refinement proof in Isabelle/HOL. We demonstrate that WasmRef-Isabelle significantly outperforms the official reference interpreter, has performance comparable to a Rust debug build of the industry WebAssembly interpreter Wasmi, and competes with unverified oracles on fuzzing throughput when deployed in Wasmtime's fuzzing infrastructure. We also present several new extensions to WasmCert-Isabelle which enhance WasmRef-Isabelle's utility as a fuzzing oracle: we add support for a number of upcoming WebAssembly features, and fully mechanise the numeric semantics of WebAssembly's integer operations.

CCS Concepts: • **Software and its engineering** → **Software verification**.

Additional Key Words and Phrases: theorem proving, refinement, virtual machine, WasmCert

ACM Reference Format:
Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. 2023. WasmRef-Isabelle: A Verified Monadic Interpreter and Industrial Fuzzing Oracle for WebAssembly. *Proc. ACM Program. Lang.* 7, PLDI, Article 110 (June 2023), 24 pages. https://doi.org/10.1145/3591224

1 INTRODUCTION

WebAssembly (Wasm) is a low-level bytecode language first introduced by Haas et al. [2017] in 2017. It is the first programming language since JavaScript to enjoy wide native support in Web browsers, and is intended to be a natural compilation target for languages such as C, C++, and Rust, enabling code written in these languages to be compiled to Wasm and embedded in Web pages. Wasm is exceptional in that its normative specification is stated in terms of a small-step formal semantics.

Authors' addresses: Conrad Watt, conrad.watt@cl.cam.ac.uk, University of Cambridge, UK; Maja Trela, University of Cambridge, UK and Jane Street, UK; Peter Lammich, University of Twente, Netherlands; Florian Märkl, Technical University of Munich, Germany.

2475-1421/2023/6-ART110
https://doi.org/10.1145/3591224

This semantics has been mechanised by Watt et al. [2021] in both Isabelle/HOL (WasmCert-Isabelle) and Coq (WasmCert-Coq). Both of these mechanisations included simple verified interpreters for Wasm which were modelled after an existing official interpeter for Wasm written in OCaml, which is maintained by its standards body for testing purposes [WebAssembly Community Group 2022b]. These interpreters were designed first and foremost to stay as close as possible to the specification's formal semantics, making minimal changes only where that semantics was not directly executable. Because of this approach, they exhibit severe performance issues which limit their ability to execute non-trivial programs. Attempts have been made in the past by the Bytecode Alliance [Bytecode Alliance 2022a], a non-profit organisation which maintains the widely-used Wasmtime implementation of Wasm [Bytecode Alliance 2022b], to use the official OCaml interpreter as a source of ground truth when performing fuzz testing. However, its handling of control flow was found to be too inefficient, and when changes to improve its performance were suggested, they were rejected by maintainers on the grounds that the required changes would compromise the closeness of the interpreter's definitions to those of the original formal semantics [Fallin 2021]. The verified interpreters of WasmCert-Coq and WasmCert-Isabelle use a similar approach to control flow, and moreover feature a significantly less efficient list-based representation of heap memory, and are therefore also unlikely to be suitable as fuzzing oracles.

In this paper, we build on WasmCert-Isabelle to present WasmRef-Isabelle, a verified monadic interpreter for Wasm which has been fully adopted into Wasmtime's continuous integration infrastructure as a fuzzing oracle. Our interpreter is proven correct with respect to the semantics of WasmCert-Isabelle by way of a two-step refinement, the latter stage of which takes advantage of the Sepref tool's separation logic verification condition generator [Lammich 2015]. Our interpreter displays performance comparable to the Wasmi industry Wasm interpreter's unoptimised debug build[1] [Parity Technologies 2022], and comprehensively outperforms not only the previous verified interpreter of Watt et al. [2021], but also the official reference interpreter. We show that the fuzzing configuration deployed by Wasmtime which uses WasmRef-Isabelle as an oracle has comparable performance to equivalent configurations using unverified oracles.

We also extend WasmCert-Isabelle and our interpreter with several new Wasm features, in order to make it more valuable as a fuzzing oracle. Moreover we fully mechanise Wasm's integer semantics, previously elided by WasmCert-Isabelle.

In summary, our contributions are as follows:

- Extensions to WasmCert-Isabelle (§3):
  - A mechanisation of the semantics of Wasm's integer numeric operations.
  - Mechanisation of several Wasm features in the final stage of standardisation, most notably SIMD (vector) instructions (up to WasmCert-Isabelle's existing abstraction of floating-point numbers).
- WasmRef-Isabelle: a new efficient monadic interpeter for Wasm, fully verified, which has been adopted in industry as a fuzzing oracle (§4).
- Experimentation measuring the performance of WasmRef-Isabelle in comparison to other implementations (§5).

Our code is available in a public repository [WasmCert 2023] and as supplementary material [Watt et al. 2023]. At a rough count, our WasmCert-Isabelle extensions total ~2530 changed or added lines of code, while our implementation and verification of WasmRef-Isabelle comes to ~5500 new lines of code.

[1]Note that since Wasmi is written in Rust, its debug build is particularly slow. See §5.

2 BACKGROUND

Wasm is a low-level language, supported by all major Web browsers, designed primarily as a compilation target for non-garbage-collected languages such as C, C++, and Rust. It has a strict type system and instruction set centered around four value types: i32, i64, f32, f64 (32- and 64-bit integer and floating point values) The "SIMD" extension [WebAssembly Community Group 2021c], which is in its final stage of standardisation, introduces a fifth value type: v128 (128-bit vector). Values can be serialised to and deserialised from a simple linear byte buffer referred to as the *memory*. They can also be stored in global and function-local *variables*. Wasm is a stack-based language, with most instructions pushing and popping values to and from a *value stack* rather than referencing variables. Wasm's type system ensures that the shape of the stack is statically known at every program point, and type checking rejects any program with potential underflows or type mismatches when popping from the stack. The Java Virtual Machine (JVM) bytecode [Gosling 1995] has a similar type system, with the key distinction that Wasm currently does not support object types. Also, unlike JVM bytecode, Wasm's control flow is *semi-structured*, with type-annotated blocks, loops, and labelled break and continue; arbitrary `goto`s are not supported.

The unit of distribution and compilation in Wasm is the *module*, which encapsulates functions along with module-wide (global) state. Modules may share their functions and global state with other modules through a system of explicit imports and exports. Before execution, a module goes through a validation, linking, and allocation step called *instantiation*.

2.1 Abstract Syntax

Fig. 1 gives the abstract syntax of Wasm. Some details are elided here for brevity (indicated in grey), but full definitions can be found in the official specification [W3C 2019]. As detailed in §3.2, we have extended WasmCert-Isabelle to support the SIMD extension. The corresponding abstract syntax is highlighted in purple. We now give further description of the definitions in Fig. 1.

$$
\begin{array}{lrcl}
\text{(immediates)} & i, min, max & ::= & nat \\
\text{(numeric types)} & t_n & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
\text{(vector types)} & t_{vec} & ::= & \mathsf{v128} \\
\text{(value types)} & t & ::= & t_n \mid t_{vec} \\
\text{(func/block types)} & ft & ::= & t^* \to t^*
\end{array}
$$

$$
\begin{array}{l}
\text{(modules) } m ::= \\
\quad \{\ \text{types} :: ft^*,\ \text{funcs} :: func^*,\ \text{globs} :: glob^*,\ \text{mems} :: mem^*,\ \text{tabs} :: tab^*, \\
\quad\ \text{data} :: \ldots,\ \text{elem} :: \ldots,\ \text{imports} :: \ldots,\ \text{exports} :: \ldots,\ \text{start} :: \ldots\ \}
\end{array}
$$

$$
\begin{array}{llll}
\text{(functions)}\ func ::= \mathbf{func}\ i\ \ t^*\ e^* & & \text{(memories)}\ mem ::= \mathbf{mem}\ min\ max \\
\text{(globals)}\ \ glob ::= \mathbf{glob}\ \text{mutable}^?\ t\ e_{\text{init}} & & \text{(tables)}\ \ tab ::= \mathbf{tab}\ min\ max
\end{array}
$$

$$
\begin{array}{lrl}
\text{(instructions)}\ e & ::= & t.\mathbf{const}\ c \mid \mathbf{i32.add} \mid \textit{other numeric stackops} \mid \textit{vector stackops} \mid \\
& & \mathbf{local.\{get/set\}}\ i \mid \mathbf{global.\{get/set\}}\ i \mid \\
& & t_n.\mathbf{load}\ flags^? \mid t_n.\mathbf{store}\ flags^? \mid t_{vec}.\mathbf{load\_vec}\ flags^?_{vec} \mid t_{vec}.\mathbf{store\_vec}\ flags^?_{vec} \mid \\
& & \mathbf{memory.size} \mid \mathbf{memory.grow} \mid \mathbf{block}\ ft\ e^* \mid \mathbf{loop}\ ft\ e^* \mid \mathbf{if}\ ft\ e^*\ e^* \mid \\
& & \mathbf{br}\ i \mid \mathbf{br\_if}\ i \mid \mathbf{br\_table}\ i^+ \mid \mathbf{call}\ i \mid \mathbf{call\_indirect}\ i \mid \mathbf{return}
\end{array}
$$

Fig. 1. Wasm abstract syntax. Highlighted elements are part of Wasm's upcoming SIMD extension.

2.1.1 Preliminary Definitions. Rather than names, Wasm generally uses static indices (*immediates*) to refer to entities like functions or variables. Value types are either numeric types or the vector type. For lists of types t^* and t'^*, a function type $t^* \to t'^*$ describes how execution of a function or block changes the stack: the topmost stack values must be of types t^*, and are replaced by values of types t'^*.

2.1.2 Module. A module m consists of a list of the type annotations used by the module's functions, a list of declared functions, and three kinds of module-wide state: a list of declared global variables, a list of declared memories, and a list of declared tables (see §2.1.3). For the sake of brevity, we have elided some components related to initialisation and import/export.

2.1.3 Module-Level State. We briefly describe the structure of function, global, memory, and table declarations. Their post-instantiation runtime representations will be discussed in §2.2. A function declaration contains a type index i to the module's type field indicating the function's type annotation, a list of types t^* declaring the types of the function's zero-initialised local variables (see the description of **local.get** in §2.1.4), and a body of instructions e^*. A global variable declaration describes whether the global variable is mutable or immutable, its value type, and an initialiser instruction which must be executed during instantiation to set the variable's initial value. A Wasm memory is a zero-initialised buffer of raw bytes. A memory declaration declares the initial size at which the memory will be created (*min*), as well as the *max* size that the memory can be grown to (see the description of **memory.grow** in §2.1.4). A table is like a memory, but its elements store function closures instead of bytes. Note that across the entire module's definitions and imports, Wasm currently mandates that at most one memory and one table may be present. Thus instructions which interact with a module's memory or table will implicitly reference this one definition.

2.1.4 Instructions. Note that instructions are also referred to as *expressions*. We describe their intuitive semantics here, while the formal semantics is given in §2.3 and §2.4.

The t.**const** c instruction pushes a statically-known value c of type t onto the stack. The **i32.add** instruction pops two i32 values off the stack, and pushes the result of wrap-around integer addition back onto the stack. Other stack operations (details elided) work analogously.

The **local.get** i instruction pops a value from the stack and stores it into the i-th declared function-scoped local variable. The **local.set** instruction retrieves such a stored value and pushes it onto the stack. As noted in §2.1.3, local variables are declared up-front as a function-level list of value type annotations, and Wasm's type system ensures that the static indices are in-bounds and have the correct types. The **global.get,set** instructions work analogously for global variables.

The type-annotated t_n.**load** instruction pops an index of type i32 off the stack, interprets the bytes in the module's memory at this index as a value of type t_n, and pushes this value onto the stack. Similarly, t_n.**store** pops an index and a value, and stores the value at the specified index in the module's memory. Both of these instructions are dynamically bounds-checked, and out-of-bound indices immediately terminate the executing program. Moreover, **load** and **store** can be statically annotated with *flags*, which allow slight variant behaviours such as sign-extending a loaded value. These behaviours are fully supported by our formalizations, but we elide the details here. Separate instructions, **load_vec** and **store_vec**, perform similar functions for vector values. The **memory.size** instruction checks the current size of the memory, while **memory.grow** allows the memory to be extended to the right with new zero-initialised byte cells.

Wasm's block constructs[2] (**block**, **loop**, and **if**) are annotated with function types, which describe how the block's execution changes the shape of the stack. If the code within the construct contains no break instructions (see below) then **block** and **loop** function identically — the instructions

[2]Note that the expressions **block**, **loop**, and **if** are called block constructs in Wasm.

inside are executed until a result is reached, and this result becomes the result of the construct (**loop** does not inherently iterate). The **if** $ft\ e^*\ e'^*$ construct first pops an i32 value from the value stack, and then behaves as **block** $ft\ e^*$ if the value is non-zero and **block** $ft\ e'^*$ otherwise. Where **block** and **loop** differ is in their interaction with the **br** instruction. A **br** i instruction is said to *target* the i-th enclosing block construct, indexed from the inside out. When a **block** is targetted, control jumps to the end of that block. When a **loop** is targetted, control jumps to the *start* of the loop, commencing another iteration. The **br_if** instruction pops an i32 from the stack, and behaves as **br** if the value is non-zero, or a no-op otherwise. The **br_table** i^+ instruction contains a non-empty list of static indices, and its execution pops an i32 from the stack to use as an index into that list, taking the last element if the index is out of bounds, and acting as a **br** to that element.

The **call** instruction executes the i-th declared or imported function of the enclosing module. The **call_indirect** i instruction pops an i32 from the stack and calls the corresponding function in the module's table. The **return** instruction returns to the caller of the currently executing function.

2.2 Runtime State

Instantiation[3] of a Wasm module produces a *configuration* of the form $S;\ F;\ e^*$. Fig. 2 details the structure of a configuration:

The S component of the configuration is the global store, which keeps track of lists of all of the function/global/memory/table instances allocated by module instantiations. We only explain the memory instance here: it consists of the current values stored in memory (as a list of bytes), as well as the maximum size the memory is allowed to grow to during execution.

The F component is the *frame*. It keeps track of function-scoped information. Its first component holds the current values of the in-scope local variables. Its second component holds the current *module instance*, which is a record keeping track of which indices of the store are in-scope for the current function. We will see an example of how the module instance is used in §2.3.

The final component of the configuration, e^*, is the list of instructions under execution.

$$\begin{array}{rrcl}
\text{(stores)} & S & ::= & \{\ \text{funcs} :: \mathit{finst}^*,\ \text{globs} :: \mathit{ginst}^*,\ \text{mems} :: \mathit{minst}^*,\ \text{tabs} :: \mathit{tinst}^*\ \} \\
\text{(memory instances)} & \mathit{minst} & ::= & \{\ \text{data} :: \mathit{byte}^*,\ \text{max} :: \mathit{nat}\ \} \\
\text{(frames)} & F & ::= & \{\ \text{locals} :: v^*,\ \text{inst} :: \mathit{inst}\ \} \\
\text{(module instances)} & \mathit{inst} & ::= & \{\ \text{types} :: \mathit{ft}^*,\ \text{funcs} :: i^*,\ \text{globs} :: i^*,\ \text{mems} :: i^*,\ \text{tabs} :: i^*\ \}
\end{array}$$

Fig. 2. Wasm runtime state.

2.3 Basic execution

In §2.1.4 we have explained the intuitive meaning of instructions. We now briefly survey Wasm's formal runtime semantics, and highlight efficiency problems that may occur in naïve realisations of these semantics such as the official OCaml reference interpreter for Wasm and the interpreter of Watt et al. [2021], and which our work aims to address.

Fig 3 shows some of the small-step reduction rules that define Wasm execution as they appear in the Wasm specification. To represent unrecoverable errors, the abstract syntax is extended with a **trap** expression. It is not directly permitted in the program syntax but may appear as a reduct.

The rule for **i32.add** illustrates the implicit representation of the value stack as a left-leading list of **const** instructions in the redex. It requires two values c_1 and c_2 on the stack before execution, and produces one value c after execution. There are standard congruence rules to generalise these "local" reduction rules to larger stacks (cf. §2.4).

[3]The instantiation process is particularly large and complicated, and thus elided from this presentation. However, it is supported by our formalization. For more details, we refer the reader to the official Wasm specification [W3C 2019]

$$\text{(instructions) } e ::= \ldots \mid \textbf{trap} \mid \ldots$$

$$\frac{c_1 + c_2 = c}{S;\ F;\ (\textbf{i32.const}\ c_1)\ (\textbf{i32.const}\ c_2)\ (\textbf{i32.add}) \hookrightarrow S;\ F;\ (\textbf{i32.const}\ c)}$$

$$\frac{F.\text{locals}[i] = v}{S;\ F;\ (\textbf{local.get}\ i) \hookrightarrow S;\ F;\ v}$$

$$\frac{\begin{array}{ll} F.\text{inst.mems}[0] = j & \\ S.\text{mems}[j] = m & m' = m \text{ with data}[c_i..c_i + \text{size}(t)] := \text{bytes}(c) \\ c_i + \text{size}(t) \le \text{length}(m) & S' = S \text{ with mems}[j] := m' \end{array}}{S;\ F;\ (\textbf{i32.const}\ c_i)\ (t.\textbf{const}\ c)\ (t.\textbf{store}) \hookrightarrow S';\ F;\ \epsilon}$$

$$\frac{\begin{array}{l} F.\text{inst.mems}[0] = j \\ S.\text{mems}[j] = m \\ c_i + \text{size}(t) > \text{length}(m) \end{array}}{S;\ F;\ (\textbf{i32.const}\ c_i)\ (t.\textbf{const}\ c)\ (t.\textbf{store}) \hookrightarrow S;\ F;\ \textbf{trap}}$$

Fig. 3. Some individual reduction rules from Wasm's runtime semantics.

The rule for **local.get** i looks up the i-th local in the current frame and pushes it onto the stack. Note that no runtime bounds check is required on the static index i, as this has already been done during instantiation. Also note that locals are represented as lists within the frame. Both by the official OCaml reference interpreter and the verified interpreters of Watt et al. [2021] implement local access with a naïve linear-time walk of this list.

Finally, we show two rules for store. The rules first retrieve the index j of the module's memory from the module instance in the frame, given by $F.insts.mem[0]$ as modules have at most one memory, and then obtains the actual memory m from the global state. If the index is in bounds, the first rule produces the updated global state in the redex. Otherwise, the second rule traps. Again, naïve implementation of these rules involves several costly list-walks for indexing and updating, as exhibited by the verified interpreters of Watt et al. [2021]. The official OCaml reference interpreter deviates from the specification by representing memory using a mutable array.

2.4 Control Flow

Wasm is unusual as a compilation target in that it does not provide any mechanism for arbitrary `goto`, instead providing the structured **block**, **loop**, and **if** constructs, and the **br** "break" instruction. We now explain the key rules concerning Wasm's control flow, which are displayed in Fig 4.

2.4.1 The Label Construct. To give a unifying semantics to Wasm's control flow constructs, the specification extends Wasm's abstract syntax with the *labelled continuation* construct **label** (Fig. 4a). The expression **label** $n\ e^*_{cont}\ e^*$ consists of the output arity n, which tracks the number of values that will be pushed onto the stack when the label terminates; the continuation e^*_{cont}, which executes if the label is targetted by a **br** instruction (see below); and the body e^*, which will be executed until either a **br** is executed, or the body terminates normally (with values or a trap).

2.4.2 Label Evaluation Contexts. The rules in Fig. 4b specify how "local" execution rules, such as the ones from Fig 3, are generalized to a context of nested labels and additional values and instructions.

The first two rules define contexts of the form $L^k[e^*]$, which express that e^* is embedded inside k nested labels, along with a value stack suffix v^*_{ctx} and a tail of instructions e^*_{ctx}, neither of which are yet participating in reduction. The context is used to define a standard congruence rule which says that if e^* can execute one step, then $L^k[e^*]$ can execute one step. The last rule of Fig. 4b specifies execution of a **br** k instruction. This instruction breaks past k enclosing labels, and the $(k+1)$-th enclosing label is considered to be *targetted* by the **br**. Taking the targetted label's arity as n, all but n stack values are dropped, with the remainder serving as the value stack against which the label's continuation is executed. If the body of a label evaluates to completion (values or a **trap**) without executing a **br**, the result is propagated outwards without executing the continuation — we elide the rules for these last cases here but include them in the supplementary material [Watt et al. 2023].

2.4.3 Block and Loop. The semantics of Wasm's **block** and **loop** constructs are defined in terms of **label** (Fig. 4c). A **block** annotated with type $t^m \rightarrow t^n$ requires t^m stack values to execute its body, and immediately terminates when targetted by a **br** instruction, ensuring n values are kept on the stack with the remainder discarded. Note that Wasm's type system ensures that the n retained values have type t^n. Therefore **block** is modelled as a label with arity n and an empty continuation. A **loop** annotated with type $t^m \rightarrow t^n$ also requires t^m stack values to execute its body, but begins another iteration when targetted by a **br** instruction, retaining m values on the stack from its previous iteration. It is modelled by a **label** with arity m and the loop itself as continuation.

2.4.4 Function Calls. Entering a function call introduces a **frame** construct with similar congruence rules to **label**, which tracks the frame F of the newly-executing function. The function terminates when the body of the **frame** is executed to completion, or when a **return** instruction is executed. Formal details are elided here but are given in full in the supplementary material [Watt et al. 2023].

$$(\text{instructions})\ e ::= \ldots\ \mid \mathbf{label}\ \mathit{nat}\ e^*\ e^* \mid \mathbf{frame}\ \mathit{nat}\ F\ e^* \mid\ \ldots$$

(a) the label construct

$$\frac{L^0 = v^*_{ctx}\ [\text{<}\mathit{hole}\text{>}]\ e^*_{ctx}}{L^0[e^*] = v^*_{ctx}\ e^*\ e^*_{ctx}} \qquad \frac{L^{k+1} = v^*_{ctx}\ (\mathbf{label}\ n\ e^*_{cont}\ L^k)\ e^*_{ctx}}{L^{k+1}[e^*] = v^*_{ctx}\ (\mathbf{label}\ n\ e^*_{cont}\ (L^k[e^*]))\ e^*_{ctx}}$$

$$\frac{S;\ F;\ e^* \hookrightarrow S';\ F';\ e'^*}{S;\ F;\ L^k[e^*] \hookrightarrow S';\ F';\ L^k[e'^*]} \qquad \frac{}{S;\ F;\ \mathbf{label}\ n\ e^*_{cont}\ (L^k[v^n\ \mathbf{br}\ k]) \hookrightarrow S;\ F;\ v^n\ e^*_{cont}}$$

(b) L evaluation context definitions and selected label evaluation context reduction rules

$$\frac{\mathit{ft} = t^m \rightarrow t^n}{S;\ F;\ v^m\ \mathbf{block}\ \mathit{ft}\ e^* \hookrightarrow S;\ F;\ \mathbf{label}\ n\ []\ (v^m\ e^*)}$$

$$\frac{\mathit{ft} = t^m \rightarrow t^n}{S;\ F;\ v^m\ \mathbf{loop}\ \mathit{ft}\ e^* \hookrightarrow S;\ F;\ \mathbf{label}\ m\ [(\mathbf{loop}\ \mathit{ft}\ e^*)]\ (v^m\ e^*)}$$

(c) block and loop reduction rules

Fig. 4. Wasm's label and frame context reduction rules, used to define the behaviour of Wasm's control flow.

2.4.5 Comment. Wasm's definitions of **label** and **frame** evaluation contexts result in both the OCaml reference interpreter and the interpreters of Watt et al. [2021] exhibiting particular inefficiency. These interpreters iteratively apply a one-step function that implements a single reduction step of the semantics. In each iteration, this function has to recurse through the nested frames and labels in the configuration in order to determine the redex. This results in extremely poor performance for programs with deep recursion or deeply nested block expressions. WasmRef-Isabelle's representation of the evaluation context avoids these inefficiencies, as we discuss in §4.

3 MODEL EXTENSIONS

3.1 Integer Numerics

The WebAssembly specification formally defines the semantics of its 32 and 64 bit integer numeric operations in terms of underlying operations on mathematical integers [W3C 2019]. However, WasmCert-Isabelle previously did not formalise this part of the specification. Instead the representations and underlying semantics of Wasm's integer operations were underspecified in terms of uninterpreted types and functions. To make these definitions executable in OCaml code extracted from Isabelle, it was axiomatised that these operations could be implemented in terms of OCaml's integer types. This had two negative consequences. First, it was impossible to reason within Isabelle itself about the semantics of Wasm's integer operations. Second, the extraction process from Isabelle to OCaml was burdened with the additional necessity of trusting that WasmCert-Isabelle's bespoke code printing setup for integers was correct.

We extend WasmCert-Isabelle with a full mechanisation of the WebAssembly specification's formalisation of 32 and 64 bit integers, and moreover prove it equivalent to an existing mature Isabelle formalisation of 32 and 64 bit integers [Lochbihler 2018] which features a well-tested code generation setup for OCaml. In this way we improve the completeness and trustworthiness of the model and the generated OCaml code, without compromising on performance.

3.2 New Features

Since its original standardisation in 2019, the Wasm language has been extended with a number of features. We now describe extensions to WasmCert-Isabelle which are of lesser theoretical interest, but still represent necessary engineering work if the model is to keep pace with the evolving Wasm standard. In all cases, not only was the language model updated, but also all associated type soundness proofs and verified artifacts such as the type checker. As we report in §4, this work presents a new verified interpreter, WasmRef-Isabelle, which supports these features. We extend WasmCert-Isabelle with support for the following:

New Conversion Operators. Wasm 1.0 included instructions for converting from a floating point number to an integer, but these instructions would trap if the floating point number could not be represented in the target integer type's range. The *non-trapping float to int conversions* proposal [WebAssembly Community Group 2019] added a suite of instructions which perform the same conversions, except with a saturating semantics in the case that the value is out of range. It was simple to extend the model with these instructions by reusing WasmCert-Isabelle's existing infrastructure for handling float to int conversions.

Multi-Value Functions and Blocks. Recall that in Wasm, all functions and block-level control flow constructs are annotated with a type which describes how their execution will transform the value stack. Wasm 1.0 included a restriction that type-annotated blocks and loops were only permitted to have an empty input type. This restriction meant that blocks and loops could not manipulate existing values on the value stack directly, but had to use local variables. In addition, blocks, loops, and functions were only permitted to have an empty or unary output type. This restriction meant that executing these constructs to completion could only push at most a single new value onto the stack. Subsequently, the *multi-value* [WebAssembly Community Group 2020] proposal relaxed these restrictions, allowing blocks, loops, and functions to have arbitrary effects on the value stack.

Many internal details of WasmCert-Isabelle were already capable of handling multi-value blocks and functions, such that fully supporting the multi-value proposal was fairly straightforward.

SIMD. There has been a long-standing effort to expose hardware-level vector types and instructions directly on the Web, first as part of a stalled proposal for JavaScript [TC39 2018], and later as part of Wasm [WebAssembly Community Group 2021c]. This is a significant extension to the language which introduces a new type of *vector value*, v128, along with a suite of instructions for operating on it. Future proposals may add other vector widths to the language. Extending WasmCert-Isabelle with the v128 type and propagating this change throughout all definitions and existing proofs involved changes to around two thousand lines of existing Isabelle code. While the modifications were fairly mechanical, there was no clear way to automate this tedious process without embarking on a more fundamental restructuring of the model and proofs, which may be valuable future work if WasmCert-Isabelle is intended to keep pace with newly specified Wasm features. We implement the SIMD instructions using the same approach that WasmCert-Isabelle uses for existing floating-point instructions. Their semantics at the type system and reduction rule level are fully mechanised. However the value-level semantics are underspecified by way of uninterpreted functions, made executable by unverified extraction to the official OCaml implementation of Wasm. We leave the precise specification of these functions to future work.

4 A FASTER VERIFIED INTERPRETER

We present WasmRef-Isabelle, a verified interpreter for Wasm which has been adopted and deployed as a fuzzing oracle for Wasmtime, a widely used Wasm implementation. This interpreter makes two key optimisations in comparison to Wasm's specified operational semantics (and therefore in comparison also to the minimal verified interpreters previously presented by Watt et al. [Watt 2018; Watt et al. 2021]): first, it uses a different representation of control flow and evaluation contexts to avoid the inefficiencies discussed in §2.4.5; second, it uses arrays and monadic state in several places to replace the Wasm specification's inefficient list-based state representations (see §2.3).

Our interpreter is defined using Imperative HOL [Bulwahn et al. 2008], a library for defining and reasoning about imperative programs in Isabelle/HOL, which models their behaviour using a state monad. We prove the interpreter sound with respect to the mechanised operational semantics of WasmCert-Isabelle by way of a two step refinement process, and then use Isabelle/HOL's built-in *extraction* mechanism [Haftmann and Bulwahn 2021] to generate executable OCaml code. Note that we make minor presentational changes here to depicted Isabelle/HOL definitions for ease of reading. One may find the full definitions in our supplementary material [Watt et al. 2023].

4.1 Top-Level Structure

Fig. 5 depicts the structure of our Isabelle/HOL contributions. We distinguish the proof steps conducted with the aid of the Verification Condition Generator (VCG) of Sepref [Lammich 2015] to highlight where we are able to benefit from this previous work. We emphasise that the VCG only uses Isabelle/HOL's "raw proof" tactics, and thus is not part of the trusted computing base.

In the course of mechanising the language extensions described in §3.2, we made additions to the existing language semantics, executable type checker, and executable instantiation definition of Watt et al. [2021]. We additionally extend the mechanised language semantics with our mechanised definition of Wasm's integer semantics, as discussed in §3.1.

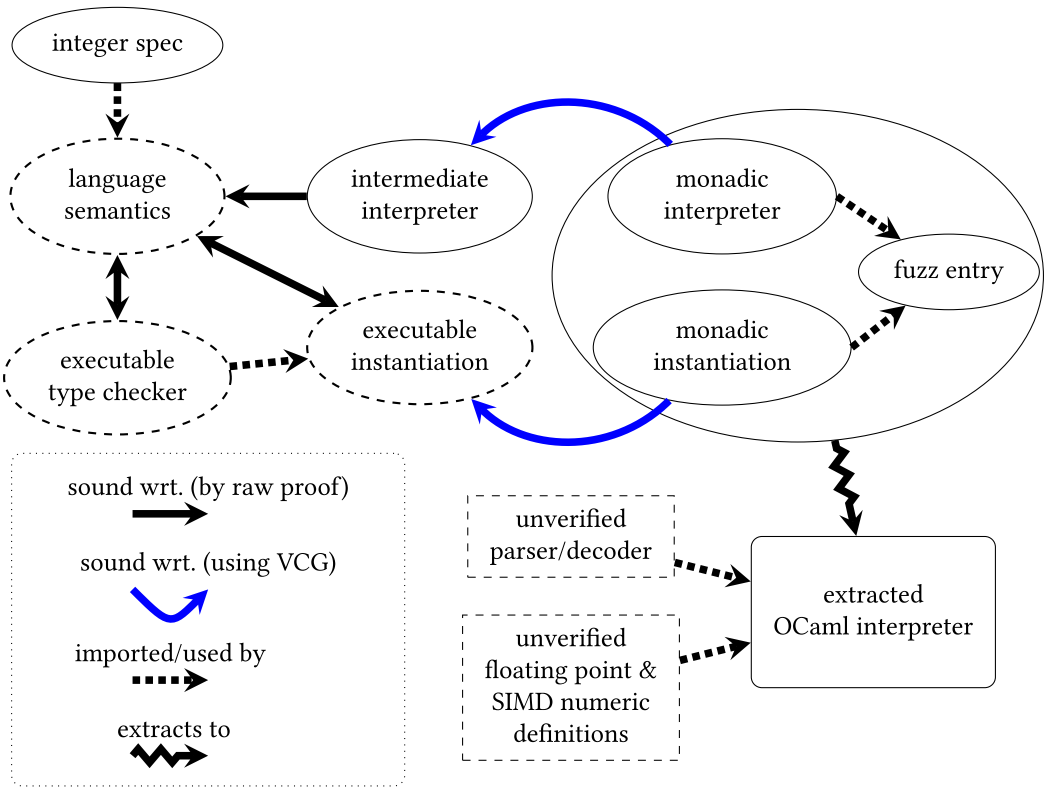

Fig. 5. A graphical depiction of the structure and associated proof guarantees of our work. Oval nodes represent Isabelle/HOL definitions and proofs, while rectangular nodes represent OCaml. Dash-bordered nodes represent definitions inherited from Watt et al. [2021] and extended as described in §3, while solid-bordered nodes represent entirely new contributions.

The core of our extracted OCaml interpreter is the monadic interpreter written in Isabelle/HOL, which implements both the control flow and state representation optimisations sketched above. The interpreter is proven sound with respect to the mechanised Wasm semantics of Watt et al. [2021] (extended as described in §3) by way of a refinement process. First, we define an intermediate interpreter which implements *only* control flow optimisations without changing the representation of Wasm's runtime state. This is proven sound with respect to the language semantics by direct application of Isabelle/HOL's pre-defined proof tactics. Second, we define a monadic interpreter which maintains the control flow optimisations of the intermediate interpreter, and additionally uses Imperative HOL [Bulwahn et al. 2008] to replace a number of inefficient lists in the WebAssembly state with monadic arrays. This is proven sound with respect to the intermediate interpreter with the aid of Sepref's VCG [Lammich 2015], a separation logic framework for Isabelle/HOL intended to "greatly simplify reasoning about programs in Imperative HOL". By transitivity, we then obtain the final theorem that the monadic interpreter is sound with respect to the original mechanised language semantics.

Similarly, we define a monadic version of instantiation which produces and updates the interpreter's monadic state. In this case we skip the intermediate step, and directly prove our monadic instantiation sound with respect to the original executable definition of instantiation provided by Watt et al. [2021]. We then compose these two definitions to obtain an entrypoint for Wasmtime's fuzz testing setup which instantiates and executes a generated test, returning the result.

We now describe in more detail the definitions and proofs associated with the intermediate interpreter and monadic interpreter.

4.2 Intermediate Interpreter

This interpreter differs from the operational semantics only in its representation and handling of control flow. By first proving this interpreter sound with respect to the operational semantics, we isolate the verification burden incurred as a result of these modifications, which is orthogonal to the verification burden we will incur in §4.3 by introducing monadic state.

Note that Wasm's operational semantics underspecifies certain operations such as memory growth, allowing them to fail completely non-deterministically. In contrast to the operational semantics, but like both the official OCaml interpreter and the previous interpreters of Watt et al. [Watt 2018; Watt et al. 2021], this interpreter implements these operations deterministically.[4] Therefore this interpreter can only be sound with respect to Wasm's operational semantics, not complete.

As mentioned in §2.4.5, Wasm's specification of evaluation contexts means that a naïve stepwise interpreter must essentially re-derive the shape of the evaluation context in linear time with each step. In our intermediate interpreter, instead of representing labels and function frames as explicit nesting expressions, we maintain a stack of labels and frames off to the side, with the evaluation of **block**, **loop**, **if**, and function calls adding to these stacks instead of introducing a **label** or **frame** expression in the reduct, avoiding the need to recurse through these expressions in subsequent steps of the interpreter. This approach has the added benefit that identifying the jump target for a **br** becomes a matter of directly walking the label stack to the targetted index, rather than handling signalling results within a recursive call, while **return** can be implemented by simply discarding the top entry of the frame stack.

Fig. 6 shows the evaluation context representations used by the interpreter, as well as a family of functions given by $[\![\cdot]\!](e^*)$, each of which relates an interpreter evaluation context (with inner hole suitably filled by e^*) back to the corresponding unoptimised definition in the Wasm semantics. The Config object contains a global store S, an inner function frame context fc, which corresponds to the currently executing function, and a stack of outer frame contexts fc^*. It is the interpreter's representation of Wasm's $(S; F; e^*)$ runtime configuration as shown in §2, with F corresponding to the frame context at the *base* of the frame context stack of Config, and e^* derived from the rest of the frame context stack and the inner frame context. An individual frame context, aside from the base frame context, corresponds to the presence of a nested **frame** construct in e^*. Each frame context contains a stack of label contexts, lc^*, with each entry in the stack corresponding to a nested **label** within the **frame**, so that the stack as a whole corresponds to the L^k context used by the Wasm semantics as shown in Fig. 4. In addition, the frame context contains the inner redex rdx, made up of a value stack and an expression list, which corresponds to the code which will execute within the innermost **label** inside the frame once all inner frames have fully executed (i.e. once all functions for which this frame is a parent in the call stack have returned). This motivates the separation in Config of the inner frame context: at each step in the Wasm semantics, the redex to be reduced is present in the most deeply nested **label** of the most deeply nested **frame** present in e^*. While a naïve implementation must recurse through these contexts to identify the redex, the intermediate interpreter can do so far more efficiently, simply by looking directly at the redex component of the inner frame context of Config.

[4] Up to the abstraction provided by Isabelle/HOL's code extraction mechanism; see §4.4.

$$\begin{array}{lrl}
\text{(redexes)} & rdx ::= & \mathsf{Redex}\ v^*\ e^* \\
\text{(interpreter label contexts)} & lc ::= & \mathsf{Label\_ctx}\ v^*\ e^*\ n\ e^* \\
\text{(interpreter frame contexts)} & fc ::= & \mathsf{Frame\_ctx}\ rdx\ lc^*\ n\ F \\
\text{(interpreter configurations)} & cfg ::= & \mathsf{Config}\ S\ fc\ fc^*
\end{array}$$

$$\begin{array}{rcl}
[\![\mathsf{Redex}\ v^*_{ctx}\ e^*_{ctx}]\!](e^*) & \triangleq & (\text{rev}\ v^*_{ctx})\ e^*\ e^*_{ctx} \\[1ex]
[\![\mathsf{Label\_ctx}\ v^*_{ctx}\ e^*_{ctx}\ n\ e^*_{cont}]\!](e^*) & \triangleq & (\text{rev}\ v^*_{ctx})\ (\textbf{label}\ n\ e^*_{cont}\ e^*)\ e^*_{ctx} \\[1ex]
[\![lc \cdot lc^*]\!](e^*) & \triangleq & [\![lc^*]\!]([\![lc]\!](e^*)) \\
[\![\ []\ ]\!](e^*) & \triangleq & e^* \\[1ex]
[\![\mathsf{Frame\_ctx}\ rdx\ lc^*\ n\ F]\!](e^*) & \triangleq & \textbf{frame}\ n\ F\ ([\![lc^*]\!]([\![rdx]\!](e^*))) \\[1ex]
[\![\mathsf{Config}\ S\ fc\ (fc' \cdot fc'^*)]\!](e^*) & \triangleq & [\![\mathsf{Config}\ S\ fc'\ fc'^*]\!]([\![fc]\!](e^*)) \\
[\![\mathsf{Config}\ S\ (\mathsf{Frame\_ctx}\ rdx\ lc^*\ n\ F)\ ([])]\!](e^*) & \triangleq & (S; F; ([\![lc^*]\!]([\![rdx]\!](e^*))))
\end{array}$$

Fig. 6. Intermediate interpreter definitions, along with a recursively-defined relation back to the configurations of the operational semantics.

To define and verify the interpreter, we first define a one-step evaluation function for an individual expression and prove it sound with respect to Wasm's runtime semantics. The interpreter takes a Config, and returns a Config and a result enumeration which is either RS_step, indicating a normal evaluation step has taken place, RS_trap, which indicates that a runtime error has occurred such as division by zero, or RS_crash, which indicates that an invariant of the interpreter has been violated (such as trying to pop from an empty value stack, which should be prevented by the type system). Fig. 7 shows the relevant soundness lemma, established by direct proof. Note the use of $[\![cfg]\!]$ to relate the interpreter's configuration back to a configuration in the Wasm semantics.

lemma run_step_e_sound:
assumes "run_step_e e $cfg = (cfg', res_{step})$"
shows

$$\text{“}\left(\begin{array}{l} res_{step} = \mathsf{RS\_step}\ \wedge \\ [\![cfg]\!]([e]) \hookrightarrow [\![cfg']\!]([]) \end{array}\right) \vee$$

$$\left(\exists str.\ \begin{array}{l} res_{step} = \mathsf{RS\_trap}\ str\ \wedge \\ [\![cfg]\!]([e]) \hookrightarrow [\![cfg']\!]([\textbf{trap}]) \end{array}\right) \vee$$

$$(\exists str.\ res_{step} = \mathsf{RS\_crash}\ str)\text{”}$$

Fig. 7. Soundness of the intermediate interpreter's auxiliary one-step function.

Then, we define the run_iter function which repeatedly extracts an expression from the current redex and calls run_step_e. The function's correctness theorem is shown in Fig. 8, relating its result to the transitive closure $\hookrightarrow^*$ of the Wasm semantics' reduction relation $\hookrightarrow$. Because all functions in Isabelle must terminate, the function takes an integer *fuel* parameter which bounds the number of iterations. It can return one of three results: RValue, which carries with it a list of values and indicates normal termination with a result, RTrap, which indicates that a Wasm **trap** runtime error has terminated execution, or RCrash, which indicates that either an invariant of the interpreter has been violated or the execution has run out of *fuel*.

$$\text{computes\_to } cfg\ (cfg', res) \triangleq \begin{array}{l} \left(\exists S'\ F'\ v^*.\ \begin{array}{c} res = \text{RValue } v^* \wedge \\ [\![cfg]\!]([\,]) \hookrightarrow^* [\![cfg']\!]([\,]) \wedge \\ [\![cfg']\!]([\,]) = (S'; F'; (\text{rev}(v^*)) \end{array}\right) \vee \\ \left(\exists str\ S'\ F'\ e'^*.\ \begin{array}{c} res = \text{RTrap } str \wedge \\ [\![cfg]\!]([\,]) \hookrightarrow^* (S'; F'; [\textbf{trap}]) \wedge \\ [\![cfg']\!]([\textbf{trap}]) = (S'; F'; e'^*) \end{array}\right) \vee \\ (\exists str.\ res = \text{RCrash } str) \end{array}$$

theorem run_iter_sound:
  assumes "run_iter *fuel cfg* = (*cfg'*, *res*)"
  shows "computes_to *cfg* (*cfg'*, *res*)"

Fig. 8. Soundness of the intermediate interpreter's top level iteration.

4.3 Monadic Interpreter

This interpreter maintains the control flow optimisations of §4.2, but acts on a monadic heap, and we replace the Wasm semantics' list-based representations of state with monadic arrays.

To verify the correctness of this interpreter, we prove that it refines our intermediate interpreter, and thus is sound with respect to the mechanised operational semantics of WasmCert-Isabelle. We conduct this proof with the aid of Sepref's VCG [Lammich 2015]. Because the intermediate interpreter already implements and verifies our control flow optimisations, this refinement proof can focus on verifying the correctness of the monadic state representation changes.

4.3.1 State Representation. Imperative HOL [Bulwahn et al. 2008] is a library providing an Isabelle/HOL model of references to allocations in a heap. It provides monadic operations which model the allocation and access of references, and in particular contains a suite of operations which support arrays. We briefly show the relevant definitions in Fig. 9. The underlying representations of the *heap* and *array* types are essentially treated as opaque, while the monadic array operations act over the *Heap* monad, which represents the result of executing an operation, along with the effect that executing the operation has on the heap (for example, if an array is modified by reference). Imperative HOL contains a code generation setup to extract such monadic operations to in-place operations over OCaml's native `Array` type, meaning that our extracted interpreter performs constant-time random access and stateful update in many situations where the official reference interpreter and the interpreters of Watt et al. [2021] perform linear-time list walks and copies.

$$\begin{array}{rcl} heap & ::= & \ldots\text{treated as opaque} \\ 'a\ array & ::= & \ldots\text{treated as opaque} \\ 'a\ Heap & ::= & \text{Heap } (heap \Rightarrow ('a \times heap)\ option) \\ \\ \text{Array.of\_list} & :: & 'a\ list \Rightarrow ('a\ array)\ Heap \\ \text{Array.nth} & :: & nat \Rightarrow\ 'a\ array \Rightarrow\ 'a\ Heap \\ \text{Array.upd} & :: & nat \Rightarrow\ 'a \Rightarrow\ 'a\ array \Rightarrow\ 'a\ Heap \\ \text{Array.len} & :: & 'a\ array \Rightarrow nat\ Heap \end{array}$$

Fig. 9. Some Imperative HOL definitions

With reference to Fig. 2, many components of Wasm's runtime state are specified as lists. Most egregiously, the memory is represented as a list of bytes, making access and update highly inefficient to execute. We define an alternative representation of WebAssembly's state that replaces the lists of Fig. 2 with Imperative HOL's *array* type, and give the main definitions in Fig. 10. Our monadic interpreter operates over this state, and is modelled in Imperative HOL's *Heap* monad. This is significantly more efficient, although Wasm's **memory.grow** instruction must be implemented by re-allocating and copying the memory *array*. Note that in components of the state where a list is used in a stack-like fashion, such as the frame list fc_m^* of Frame_ctx_m, we retain the list representation instead of using an *array*.

$$\text{(monadic stores) } S\_m ::= \{\ \text{funcs} :: \mathit{finst\ array},\ \text{globs} :: \mathit{ginst\ array},\ \text{mems} :: \mathit{minst\_m\ array},\ \text{tabs} :: \mathit{tinst\_m\ array}\ \}$$

$$\begin{array}{rcll} \text{(monadic memory instances)} & \mathit{minst\_m} & ::= & \{\ \text{data} :: \mathit{byte\ array},\ \text{max} :: \mathit{nat}\ \} \\ \text{(monadic frames)} & F\_m & ::= & \{\ \text{locals} :: \mathit{v\ array},\ \text{inst} :: \mathit{inst\_m}\ \} \end{array}$$

$$\text{(monadic module instances) } \mathit{inst\_m} ::= \{\ \text{types} :: \mathit{ft\ array},\ \text{funcs} :: \mathit{i\ array},\ \text{globs} :: \mathit{i\ array},\ \text{mems} :: \mathit{i\ array},\ \text{tabs} :: \mathit{i\ array}\ \}$$

$$\text{(monadic interpreter frame contexts) } \mathit{fc\_m} ::= \text{Frame\_ctx\_m}\ \mathit{rdx}\ \mathit{lc}^*\ n\ F\_m$$

$$\text{(monadic interpreter configurations) } \mathit{cfg\_m} ::= \text{Config\_m}\ S\_m\ \mathit{fc\_m}\ \mathit{fc\_m}^*$$

$$\text{run\_iter\_m} :: \mathit{fuel} \Rightarrow \mathit{cfg\_m} \Rightarrow (\mathit{cfg\_m} \times \mathit{res})\ \mathit{Heap}$$

Fig. 10. Monadic interpreter definitions and top-level type signature.

4.3.2 Verification. Sepref's VCG [Lammich 2015] allows us to state and prove Hoare triples of the form $\{P\}\ C\ \{\lambda x.\ Q\ x\}$. This triple states that for all heaps that satisfy the precondition P, and on which the monadic program fragment C returns a result x, the new heap will satisfy the postcondition $Q\ x$. P and Q are expressed in separation logic [Reynolds 2002], allowing modular reasoning over the heap.

While the original library provided by Sepref proves total correctness, we modify the library to prove *partial* correctness, so that a Hoare triple holds if the program C diverges or raises an error. This is because our refinement from the intermediate interpreter to the monadic interpreter can be expressed unconditionally as a partial correctness Hoare triple, while the version requiring total correctness only holds assuming the input program is well-typed. Partial correctness is sufficient for our purposes however, since our use of the interpreter as a fuzzing oracle only requires that non-crash outputs can be trusted (although any observed crashes would represent a bug in our interpreter which should be fixed). Therefore we avoid the additional complications that would arise from introducing assumptions about the type system into our proofs.

We show relevant lemmas in Fig 11. First, we show the Hoare triple for Array.nth (see Fig. 9), which is already provided by the separation logic library. Using a standard "points-to" relation $\mapsto_a$, it associates an array reference with a list representing the values in the allocated array in the heap.

The next lemma, run_step_e_m_triple, connects the behaviours of the single-step function of our monadic interpreter, run_step_e_m, and of our intermediate interpreter, run_step_e. The cfg_m_assn *cfg cfg_m* assertion captures that the array references within *cfg_m* do not alias and point to arrays equivalent to the lists of *cfg*. The non-aliasing property, which is naturally expressed in separation logic, allows us to reason about the update-by-reference of one array within the interpreter state, without having to explicitly reason that other arrays are not affected. The lemmas are proved using Sepref's VCG, which automates Hoare-logic and frame inference.

We then use run_step_m_triple to prove a further triple connecting the behaviour of our monadic interpreter's main loop with the behaviour of the same in our intermediate interpreter. Finally, we can use this and the result of Fig. 8 to prove a theorem expressing the behaviour of our monadic interpreter directly in terms of WasmCert-Isabelle's operational semantics. Note that the proof of correctness for our monadic definition of instantiation proceeds similarly, but we elide further description here for space reasons.

lemma nth_rule:
shows "$\{ arr \mapsto_a x^* \}$ Array.nth n arr $\{\lambda res.\ arr \mapsto_a x^* \ast res \doteq x^*[n]\}$"

lemma run_step_e_m_triple:
shows "$\{$ cfg_m_assn cfg cfg_m $\}$

run_step_e_m e cfg_m

$$\left\{\lambda(cfg\_m', res).\ \exists cfg'.\ \begin{array}{l}\text{cfg\_m\_assn}\ cfg'\ cfg\_m'\ \ast \\ \text{run\_step\_e}\ e\ cfg = (cfg', res)\end{array}\right\}\text{"}$$

theorem run_iter_m_sound:
shows "$\{$ cfg_m_assn cfg cfg_m $\}$

run_iter_m $fuel$ cfg_m

$$\left\{\lambda(cfg\_m', res).\ \exists cfg'.\ \begin{array}{l}\text{cfg\_m\_assn}\ cfg'\ cfg\_m'\ \ast \\ \text{computes\_to}\ cfg\ (cfg', res)\end{array}\right\}\text{"}$$

Fig. 11. Hoare triples involved in the monadic interpreter's proof of partial correctness.

4.4 Trusted Computing Base

Our work stems from the Isabelle/HOL Wasm model of Watt et al. [2021], which is intended to have a line-by-line correspondence to the pen-and-paper formal model present in the official Wasm specification, and we must trust this translation (although we can validate it to some extent through experimental testing). Our chain of proof from the mechanised semantics to our interpreter relies only on the correctness and interpretability of our theorems as proven in Isabelle/HOL (see Pollack [1996] for a discussion of concerns here).

In producing the executable WasmRef-Isabelle interpreter, we must trust Isabelle/HOL's code extraction to OCaml, and the OCaml toolchain itself. A particular hazard here is the use of `code_printing` equations which force Isabelle/HOL to extract individual types or functions as unchecked user-provided strings. These are occasionally necessary as a way of directing the extraction to implement certain Isabelle/HOL types such as *uint32* directly in terms of their OCaml counterparts. We only introduce such equations where necessary to connect the Isabelle/HOL definitions to their OCaml counterparts. The only exception is that we define an optional `code_printing` setup which allows the Isabelle/HOL-defined *byte array* type and associated operations to be extracted to the more efficient OCaml `bytes` type rather than Isabelle/HOL's default target, the less efficient `char array` type. Since the extracted interpreter with this setup enabled passes all reference tests and did not report any divergences in behaviour during fuzzing, we believe our handwritten `code_printing` setup is correct, although it can be disabled if desired.

Some aspects of our generated interpreter are supplied directly at the OCaml level as unverified code, namely the top-level command line interface logic, the parser and binary decoder, the implementation of floating point arithmetic, and the implementation of vector (SIMD) numeric operations. At the Isabelle level, as discussed in §3.1, it is essentially axiomatised that the relevant OCaml-level types and functions exist and can be used to implement the uninterpreted definitions of the model. In all of these cases almost all of the supplied OCaml code comes directly from the official Wasm reference interpreter, with a small amount of handwritten OCaml code to complete the interface between the reference interpreter's code and the generated code of WasmRef-Isabelle.

5 EVALUATION

We evaluate WasmRef-Isabelle's performance both as a stand-alone interpreter, and as a component of Wasmtime's fuzzing infrastructure. On the strength of these results, WasmRef-Isabelle has been adopted and deployed as a fuzzing oracle into Wasmtime's fuzzing and continuous integration infrastructure, including its OSS-Fuzz instance [WasmCert 2023].

5.1 Performance

In this section we investigate the performance of our interpreter. All benchmarks are executed on a Lenovo T590 laptop with a i5-8265U processor and 8GB RAM. All numbers reported below are the average of 10 repetitions unless otherwise stated.

Before we go further, it is worth emphasising that it is easy to manufacture pathological programs for which the OCaml reference interpreter and the interpreter of Watt et al. [2021] have arbitrarily poor performance in comparison to WasmRef-Isabelle, since our optimisations improve the time complexity of certain interpreter steps in comparison to these interpreters. The fuzzing infrastructure of Wasmtime generated several tests which run particularly slowly in the OCaml reference interpreter, including one which executes over 4000 times faster in WasmRef-Isabelle [Crichton 2021]. To briefly illustrate this, Fig. 12 graphs the execution time of the simple recursive function $f(0) \triangleq 0; f(n+1) \triangleq f(n) + n^5$ for different values of n (with the exponentiation to the fifth power implemented via repeated multiplication), executing in the interpreter of Watt et al. [2021], in the official OCaml reference interpreter, and in WasmRef-Isabelle. The execution times of both these existing interpreters scale super-linearly with n due to their inefficient handling of function frames, while WasmRef-Isabelle's execution time in comparison barely doubles across the input values considered.

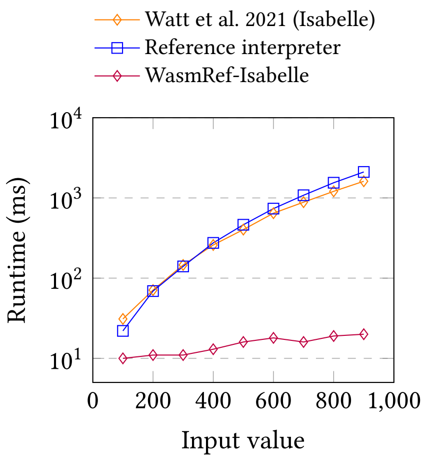

Fig. 12. Graphing execution times for the function $f(0) \triangleq 0; f(n+1) \triangleq f(n) + n^5$ (note the log scale for runtime).

The following benchmarks compare WasmRef-Isabelle to the official OCaml reference interpreter [WebAssembly Community Group 2022b], and the previous Isabelle/HOL-extracted interpreter of Watt et al. [2021], as well as two efficient industry implementations written in Rust: Wasmi [Parity Technologies 2022] an industry Wasm interpreter used in several blockchain-related applications and also deployed as a fuzzing oracle for Wasmtime; and Wasmtime [Bytecode Alliance 2022b] itself, the Wasm engine which now uses WasmRef-Isabelle as a fuzzing oracle. We emphasise that we do not expect the performance of WasmRef-Isabelle to be in any way competitive with the optimised release builds either of these implementations, although our experiments suggest that WasmRef-Isabelle does have roughly comparable performance to a debug build of Wasmi.

We also measure the throughput of Wasmtime's fuzzing infrastructure when using WasmRef-Isabelle as an oracle, and contrast it with the throughputs measured when using either the OCaml reference interpreter or Wasmi as an oracle.

Table 1. CoreMark 1.0 (Wasm port) results.

implementation	raw score
Watt et al. 2021 (Isabelle)	0.046172101
Reference interpreter	1.4857737
WasmRef-Isabelle	9.5817566
Wasmi (dev)	10.886131
Wasmi (release)	205.78702
Wasmtime (JIT)	12704.866

Table 2. Recursive fibonacci (input: 35).

implementation	execution time (s)	ratio
Watt et al. 2021 (Isabelle)	1090	30
Reference interpreter	750	21
Wasmi (dev)	72.8	2.0
WasmRef-Isabelle	36.1	1
Wasmi (release)	2.76	0.076
Wasmtime (JIT)	0.076	0.0021

5.1.1 CoreMark. CoreMark [Gal-On and Levy 2009] is an industry-standard benchmark written in C, which performs a suite of low-level data-processing operations such as list sort and matrix multiplication, outputting a single score upon completion which can be used to comparatively rank CPUs and compilers. A port of CoreMark 1.0 to Wasm with stripped-down system dependencies has been widely used to compare the performance of different Wasm implementations [wasm3 2021]. Table 1 shows the observed performance score for the various Wasm implementations we have considered. Both the reference interpreter and the interpreter of Watt et al. [2021] score extremely poorly in comparison to WasmRef-Isabelle, which itself has a score slightly below the debug build of Wasmi. As expected, Wasmi release and Wasmtime exhibit significantly better performance.

The poor performances of the reference interpreter and the interpreter of Watt et al. [2021] are likely due to a number of inefficiencies which are avoided by WasmRef-Isabelle. First, the core computations of the benchmark take place inside several nested blocks, exercising the implementation inefficiencies mentioned in §2.4. Moreover, the list-based representation both of these implementations use for local variables also degrades performance, as each local variable access takes linear time in the number of declared local variables. Finally, the list-based representation of memory used by the interpreter of Watt et al. [2021] is a significant source of inefficiency, and is likely the primary cause of the gap in score between it and the reference interpreter.

5.1.2 Recursive Fibonacci. Table 2 reports the time taken in seconds for the various Wasm implementations to execute a naïve recursive fibonacci calculation for a fixed input of 35. This micro-benchmark stresses the ability of the Wasm implementations to efficiently handle deeply-nested function calls. Each function call introduces an additional **frame**, and so we use this micro-benchmark to highlight the previously-mentioned massive control flow inefficiencies in both the official reference interpreter and the Isabelle/HOL interpreter of Watt et al. [2021], which become even greater as the depth of the call stack grows (a graph for different input values is included in the supplementary material [Watt et al. 2023]). This benchmark also represents WasmRef-Isabelle's strongest performance against Wasmi, maintaining nearly twice the speed of Wasmi's debug build, although it still falls far short of the optimised release build's performance.

5.1.3 Iterative Fibonacci and Memory Walk. Fig. 13 shows the time taken for the various implementations to execute two simple iterative algorithms which are based on conformance tests present in Wasm's official test suite. The first calculates the 10^8th fibonacci number through bottom-up iteration, while the second iterates over 10^8 newly created memory cells and checks that each is zero-initialised. Since the iterative fibonacci test only executes a small amount of arithmetic in a tight loop, it does not exercise any of the previously-discussed design-level inefficiencies present in either the reference interpreter or the interpreter of Watt et al. [2021]. In effect, it represents the best chance for these interpreters to perform well in comparison to WasmRef-Isabelle. Nevertheless, WasmRef-Isabelle still outperforms them: its runtime is around 80% that of the reference interpreter and around 40% that of Watt et al. [2021]. It is worth noting however that the industry implementations handle such tight loops particularly efficiently in comparison. While WasmRef-Isabelle was just under 13 times slower than Wasmi's release build for recursive fibonacci, here it is just under 47 times slower. Similarly, the memory walk test should not expose inherent issues in the reference interpreter, but should expose the limitations inherent in the list-based memory of the interpreter of Watt et al. [2021]. As expected, we see that WasmRef-Isabelle outperforms the reference interpeter by roughly the same ratio as observed in the iterative fibonacci micro-benchmark, while the interpreter of Watt et al. [2021] fails to terminate within eight hours.

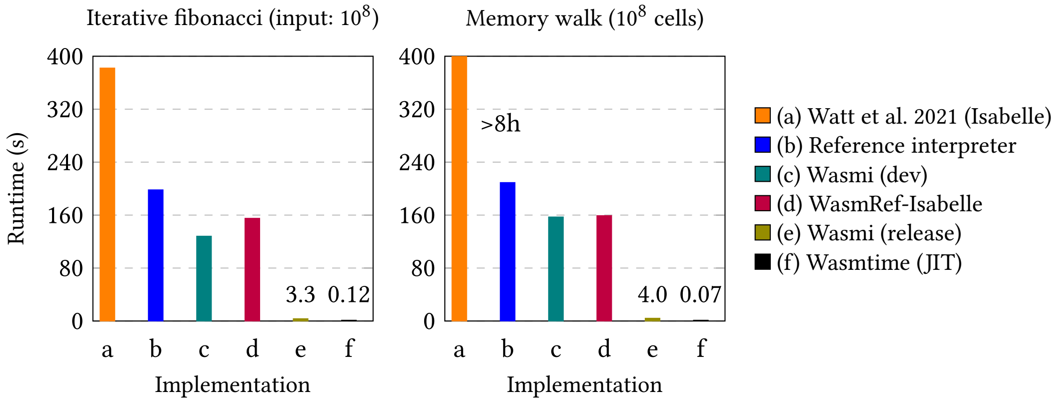

Fig. 13. Iterative fibonacci and memory walk benchmarks.

5.1.4 LibFuzzer Throughput. Wasmtime's fuzzing infrastructure makes use of the LibFuzzer [LLVM Progect 2022] tool using either WasmRef-Isabelle, Wasmi, or Google's V8 JIT as an oracle. The configuration used by WasmRef-Isabelle was originally developed by Wasmtime's maintainers for fuzzing against the official OCaml reference interpreter, before this was abandoned due to its poor performance. We first benchmark the WasmRef-Isabelle configuration against a version of the same configuration that restores the OCaml reference interpreter, and an analogous version using Wasmi. V8's configuration uses a different feature-set and is benchmarked separately below. Measuring the throughput of each of these three configurations requires care: LibFuzzer uses coverage metrics and the corpus of previously-generated tests to guide the generation of subsequent, more complex tests. We begin with an empty test corpus and execute LibFuzzer in its default configuration for eight hours with fixed initial seed 3142680329. Results can be found below:

oracle	total tests	slowest test (secs)
Reference Interpreter	1900091	48
WasmRef-Isabelle	2740908	<1
Wasmi (release)	2793736	<1

Despite the release build of Wasmi consistently outperforming WasmRef-Isabelle by at least an order of magnitude in previous tests, their reported fuzzing throughputs are similar, while the fuzzing output of the official reference interpreter is noticeably lower. These results suggest that performance of the coverage-based fuzzing used by Wasmtime is not dominated by the performance of the oracle unless the oracle is egregiously slow — note that all generated fuzz tests took less than a second to execute in both WasmRef-Isabelle and Wasmi, (the minimum granularity that LibFuzzer reports), while the reference interpreter required 48 seconds to execute its longest-running test.

Note also that the reference interpreter's poor performance disproportionately affects tests with deeply-nested blocks, distorting the distribution of executed tests and occasionally causing spurious timeouts (such as with the "4000x" example mentioned above).

The test configurations above generate tests exercising only the features of Wasm 1.0. As , we can enable this feature during test generation. Previously, Wasmtime could only perform fuzzing involving SIMD instructions by using a special configuration with Google's V8 JavaScript engine as an oracle; Wasmi has no plans to support this feature. We compare the throughput of WasmRef-Isabelle, which supports SIMD vector instructions through scalar emulation, against V8 as an oracle for this fuzzing configuration. Again, we report the results of an eight hour run with initial seed 3142680329.

oracle	total tests	slowest test (secs)
WasmRef-Isabelle	2923403	<1
V8	2517064	<1

These numbers contain two surprises. First, the throughput of the WasmRef-Isabelle fuzzing configuration is somewhat higher with SIMD enabled. We speculate that the test generator may spend a greater proportion of the eight hour runtime generating trivial (even single-instruction) tests due to the large number of new instructions introduced by the SIMD proposal. Second, the V8-based configuration is surprisingly slow in comparison. We believe that Wasmtime's fuzzing setup currently interfaces with V8 in a particularly inefficient way: for each generated Wasm test, a small JavaScript program wrapping the test is dynamically produced for V8 to execute.

5.1.5 Conclusions. These results demonstrate that WasmRef-Isabelle significantly outperforms both the previous work of Watt et al. [2021], and the unverified reference interpreter. Its performance is competitive with the unoptimised developer build of Wasmi, a widely used industry Wasm interpeter. Note though that Wasmi is written in Rust, which is known to have a larger gap between the performances of its different optimisation levels than C/C++ – for example the release build of Wasmi shows significantly better performance than WasmRef-Isabelle: around 20x in CoreMark. However, we observe that both WasmRef-Isabelle and the release build of Wasmi exhibit similar throughput as fuzzing oracles for Wasmtime, suggesting that WasmRef-Isabelle's performance is sufficient for the fuzzing throughput to be dominated by other factors such as test generation.

5.2 Bug Finding

To execute Wasm's official test suite, we must discard tests which involve the upcoming Reference Types [WebAssembly Community Group 2021b] and Bulk Memory [WebAssembly Community Group 2021a] features, which are not yet supported. We successfully pass all remaining tests.

Since its deployment in March 2022, the fuzzing configuration with WasmRef-Isabelle as an oracle has identified a single bug in Wasmtime [ClusterFuzz 2022]. This was a crash bug which did not depend on comparing functional behaviour. It is worth noting that Wasmtime has already been extensively fuzzed against other industry implementations using the same test generation strategy, and so it is likely that the main value of fuzzing against WasmRef-Isabelle will be in discovering bugs in future changes to the codebase, rather than in revealing extant bugs. Moreover, contributors are encouraged to fuzz their changes locally, reducing the chances that the continuous integration infrastructure discovers bugs. In the same time period, the publicly-available bug tracker indicates that no other part of Wasmtime's fuzzing continuous integration infrastructure, which also includes differential configurations fuzzing against Wasmi and V8, reported a semantic bug in Wasmtime.

Our fuzzing configuration did also reveal one deficiency in WasmRef-Isabelle by generating a test containing over 450 repeated applications of the bitwise **i64.popcnt** instruction, which WasmRef-Isabelle took abnormally long to execute (over a minute). It was discovered that the way we were modelling **popcnt** in Isabelle/HOL caused particularly inefficient code to be generated for this case in the interpreter. To fix this, we defined a more efficient implementation for **popcnt**, and thanks to our new semantic model of Wasm's integer operations, we were able to prove our efficient implementation equivalent to the previous one, enabling Isabelle/HOL's code generator to use it instead. Once this was accomplished, the test case executed in just under a second. Fuzzing has not yet revealed any other deficiencies, semantic or otherwise, in WasmRef-Isabelle.

To validate the fuzzing configuration, we locally introduced some simple bugs to Wasmtime of a sort suggested by its maintainers: permuting the order of operands for individual numeric instructions, and flipping individual bits in the constant masks used in implementing vector instructions. Each of our introduced bugs was discovered within a few minutes of fuzzing. We should emphasise that so long as our interpreter can execute generated tests with reasonable speed, the question of whether or not an extant bug is discovered is mainly down to the effectiveness of the fuzzing infrastructure's coverage-guided test generation. Aside from the test involving **popcnt** mentioned above, we have no evidence that WasmRef-Isabelle has been a fuzzing bottleneck for Wasmtime's setup in comparison to unverified oracles such as Wasmi or V8.

6 RELATED WORK

Refinement is a natural approach to reduce the complexity of both implementations and correctness proofs, by separating different aspects. Early formal treatments are by Hoare [1972] and Back and von Wright [1998], and early tools include the VDM [Alagar and Periyasamy 1998] and B-Method [Abrial 1996]. Refinement is also used in more recent verification projects like the SeL4 kernel [Klein et al. 2014] or the verified CompCert compiler [Blazy and Leroy 2009]. Closest to our approach are the verified UNSAT checkers by Heule et al. [2017] and Lammich [2020], that use multiple refinement steps, ultimately introducing imperative arrays.

Refinement techniques also differ on the level of automation: in our approach, we use plain Isabelle/HOL and a standard verification condition generator [Lammich and Meis 2012]. This requires some programming and proving discipline to not accidentally break the abstraction barriers and get overly complicated proofs. A similar approach is used by Heule et al. [2017]. On the other hand, there are frameworks like Fiat [Pit-Claudel et al. 2020], CoqEAL [Cohen et al. 2013], Autoref [Lammich 2013], and Sepref [Lammich 2015] that assist with data refinement, and synthesize the refined program and proof (semi-)automatically. The memoization framework of Wimmer et al. [2018] is also highly automated but restricted to dynamic programming.

In our work, we could not take advantage of the more automated approaches, as they are not (yet) general enough to handle certain subtleties with aliasing in our monadic state. We believe that our work provides a valuable target against which to develop more general automation.

While the possibility of using a verified programming language implementation as a fuzzing oracle to test industry language implementations features prominently in the "folklore" of the field, we have found a comparative paucity of published examples. The most prominent work in this vein is undoubtedly that of Yang et al. [2011], in which their Csmith fuzz test generation tool for C is used to pit the verified CompCert C compiler [Leroy 2009] against a suite of industry C implementations. Chapman et al. [2019] briefly describe testing their verified interpreter for the smart contract language Plutus Core against a production implementation. Watt [2018] briefly describes testing an early version of the Wasm interpreter of Watt et al. [2021] against industry implementations, using Csmith-generated programs compiled to Wasm. Schumi and Sun [2021] use K-Java [Bogdanas and Roşu 2015] as an oracle in fuzzing against the Java compiler, and KSolidity [Jiao et al. 2020] as an oracle in fuzzing against the Solidity compiler. While it is impossible to make an apples-to-apples comparison, the authors describe running ~30,000 Java tests over ~3 weeks, and ~50,000 Solidity tests over ~2 weeks, suggesting a fuzzing throughput that is several orders of magnitude below that of our setup (see §5.1.4) — this is not necessarily surprising since the oracles are naïvely derived directly from the relevant K framework semantic definitions [Roșu and Șerbănută 2010]. The authors also suggest generating tests to achieve coverage of the oracle rather than of the target implementation, which may be interesting future work in our context.

More broadly, there are many examples of formal techniques being used to test an implementation against the specification it is intended to follow. Barr et al. [2015] and Hierons et al. [2009] both survey a variety of approaches for specifying and obtaining oracles for software testing. SibylFS [Ridge et al. 2015] is an oracular formal model and test suite for file system APIs. He and Turner [1999] describe the generation of tests for digital circuits from a formal model. Siegl et al. [2011] describe the use of formal models to generate tests for safety systems in automobiles.

Formalizing semantics of programming languages in theorem provers has a long tradition – examples include Java [Klein and Nipkow 2006], JavaScript [Bodin et al. 2014; Guha et al. 2010], Standard ML [Kumar et al. 2014; Lee et al. 2007], C [Krebbers 2015; Leroy 2009; Norrish 1998], and Rust [Jung et al. 2017]. Other formalizations [Bogdanas and Roşu 2015; Ellison and Rosu 2012; Park et al. 2015] use specialized semantic frameworks like K [Roșu and Șerbănută 2010].

7 CONCLUSION AND FUTURE WORK

We have presented WasmRef-Isabelle, a monadic Wasm interpreter with a mechanised proof of correctness in Isabelle/HOL, which has been deployed as a fuzzing oracle in a major industrial Wasm implementation's continuous integration infrastructure. We have demonstrated that in this capacity WasmRef-Isabelle's throughput is sufficient to be competitive with unverified implementations. It is exceptionally rare for a mechanised and verified programming language implementation to achieve this level of industry adoption, and we believe WasmRef-Isabelle's success here is thanks to the maturity of verification libraries such as Imperative HOL and Sepref.

There are three main axes along which WasmRef-Isabelle can be directly improved. First, there is still significant room to increase its performance by investigating further optimisations, efficient data structure representations, and so on. While we have demonstrated that WasmRef-Isabelle's performance is sufficient for use as a fuzzing oracle, it is still significantly slower than optimised Wasm interpreters, and as such it is likely not yet suitable for use in more general contexts (see Titzer [2022] for a broader discussion of the state-of-the-art in industrial Wasm interpreters). Second, we believe there are opportunities to improve the automation of its proof of correctness by developing the capabilities of proof assistance tools. Third, we can expand WasmCert-Isabelle and WasmRef-Isabelle to support further upcoming Wasm features. The second edition of the Wasm standard (Wasm 2.0) is currently being drafted, and thanks to our extensions WasmCert-Isabelle supports all of its planned features except for reference types [WebAssembly Community Group 2021b] and bulk memory operations [WebAssembly Community Group 2021a]. We consider it a high priority to extend the model and interpreter with these remaining features in order to keep pace with the Wasm standard and maintain WasmRef-Isabelle's usefulness as a fuzzing oracle. Planned extensions beyond Wasm 2.0 such as threads [WebAssembly Community Group 2021d] and garbage-collected types [WebAssembly Community Group 2022a] would be much more challenging to faithfully mechanise. Still, some Wasm implementations not associated with Web browsers have signalled that they have no plans to support these more ambitious extensions [Parity Technologies 2022], and so the feature-set supported by WasmRef-Isabelle will likely remain relevant for some time.

ACKNOWLEDGMENTS

We thank the members of the Bytecode Alliance for their advice and help during this work; in particular Andrew Brown, Alex Crichton, Chris Fallin, and Nick Fitzgerald. We also thank Tobias Nipkow, Simon Roßkopf, and Simon Wimmer for their advice and support.

REFERENCES

Jean-Raymond Abrial. 1996. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press.

V. S. Alagar and K. Periyasamy. 1998. *Vienna Development Method.* Springer New York, New York, NY, 219–279. https://doi.org/10.1007/978-1-4757-2920-7_9

Ralph-Johan Back and Joakim von Wright. 1998. *Refinement Calculus — A Systematic Introduction.*

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288. http://xavierleroy.org/publi/Clight.pdf

Martin Bodin, Arthur Chargueraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. 2014. A Trusted Mechanised JavaScript Specification. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14).* Association for Computing Machinery, New York, NY, USA, 87–100. https://doi.org/10.1145/2535838.2535876

Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15).* Association for Computing Machinery, New York, NY, USA, 12 pages. https://doi.org/10.1145/2676726.2676982

Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. 2008. Imperative Functional Programming with Isabelle/HOL. In *TPHOLs 2008 (LNCS, Vol. 5170)*, Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). Springer, 134–149.

Bytecode Alliance. 2022a. Bytecode Alliance. https://bytecodealliance.org/.

Bytecode Alliance. 2022b. wasmtime. https://github.com/bytecodealliance/wasmtime.

James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction: 13th International Conference, MPC 2019, Porto, Portugal, 2019, Proceedings* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 43 pages.

ClusterFuzz. 2022. wasmtime:differential_spec: ASSERT: assertion failed. https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=47918.

Cyril Cohen, Maxime Dénès, and Anders Mörtberg. 2013. Refinements for Free!. In *CPP 2013 (LNCS, Vol. 8307)*, Georges Gonthier and Michael Norrish (Eds.). Springer, 147–162.

Alex Crichton. 2021. Timeouts in spec interpreter fuzzing. https://github.com/bytecodealliance/wasmtime/issues/3186.

Chucky Ellison and Grigore Rosu. 2012. An Executable Formal Semantics of C with Applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12).* Association for Computing Machinery, New York, NY, USA, 533–544. https://doi.org/10.1145/2103656.2103719

Chris Fallin. 2021. [interpreter] Fix quadratic behavior when stepping in deeply-nested scopes. https://github.com/WebAssembly/spec/pull/1354.

Shay Gal-On and Markus Levy. 2009. CoreMark. https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf.

James Gosling. 1995. Java Intermediate Bytecodes: ACM SIGPLAN Workshop on Intermediate Representations (IR'95). In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, California, USA) *(IR '95).* Association for Computing Machinery, New York, NY, USA, 111–118. https://doi.org/10.1145/202529.202541

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The Essence of Javascript. In *Proceedings of the 24th European Conference on Object-Oriented Programming* (Maribor, Slovenia) *(ECOOP'10).* Springer-Verlag, Berlin, Heidelberg, 126–150.

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* (Barcelona, Spain) *(PLDI 2017).* Association for Computing Machinery, New York, NY, USA, 16 pages. https://doi.org/10.1145/3062341.3062363

Florian Haftmann and Lukas Bulwahn. 2021. Code generation from Isabelle/HOL theories. https://isabelle.in.tum.de/doc/codegen.pdf.

Ji He and Kenneth J. Turner. 1999. Protocol-Inspired Hardware Testing. In *IWTCS.*

Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. 2017. Efficient, verified checking of propositional proofs. In *International Conference on Interactive Theorem Proving*. Springer, 269–284.

Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. 2009. Using Formal Specifications to Support Testing. *ACM Comput. Surv.* 41, 2, Article 9 (feb 2009), 76 pages. https://doi.org/10.1145/1459352.1459354

C. A. R. Hoare. 1972. Proof of correctness of data representations. *Acta Informatica* 1 (1972), 271–281. Issue 4.

Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2020. Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *2020 IEEE Symposium on Security and Privacy (SP).* 1695–1712. https://doi.org/10.1109/SP40000.2020.00066

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (dec 2017), 34 pages. https://doi.org/10.1145/3158154

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.* 32, 1 (2014), 2:1–2:70. https://doi.org/10.1145/2560537

Gerwin Klein and Tobias Nipkow. 2006. A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.* 28, 4 (jul 2006), 619–695. https://doi.org/10.1145/1146809.1146811

Robbert Krebbers. 2015. *The C standard formalized in Coq.* Ph. D. Dissertation. Radboud University Nijmegen.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 179–191. https://doi.org/10.1145/2535838.2535841

Peter Lammich. 2013. Automatic Data Refinement. In *ITP*. LNCS, Vol. 7998. Springer, 84–99.

Peter Lammich. 2015. Refinement to Imperative/HOL. In *ITP*. LNCS, Vol. 9236. Springer, 253–269.

Peter Lammich. 2020. Efficient Verified (UN)SAT Certificate Checking. *J. Autom. Reason.* 64, 3 (2020), 513–532. https://doi.org/10.1007/s10817-019-09525-z

Peter Lammich and Rene Meis. 2012. A Separation Logic Framework for Imperative HOL. *Archive of Formal Proofs* (November 2012). https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development.

Daniel K. Lee, Karl Crary, and Robert Harper. 2007. Towards a Mechanized Metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/1190216.1190245

Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 9 pages. https://doi.org/10.1145/1538788.1538814

LLVM Progect. 2022. LibFuzzer. https://llvm.org/docs/LibFuzzer.html.

Andreas Lochbihler. 2018. Fast Machine Words in Isabelle/HOL. In *Interactive Theorem Proving*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer International Publishing, Cham, 388–410.

Michael Norrish. 1998. *C formalised in HOL.* Technical Report.

Parity Technologies. 2022. wasmi. https://github.com/paritytech/wasmi.

Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 346–356. https://doi.org/10.1145/2737924.2737991

Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *Automated Reasoning*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer International Publishing, Cham, 119–137.

Robert H. Pollack. 1996. How to Believe a Machine-Checked Proof 1.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc of. Logic in Computer Science (LICS)*. IEEE, 55–74.

Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: Formal Specification and Oracle-Based Testing for POSIX and Real-World File Systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 38–53. https://doi.org/10.1145/2815400.2815411

Grigore Roșu and Traian Florin Șerbănută. 2010. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* 79, 6 (2010). https://doi.org/10.1016/j.jlap.2010.03.012 Membrane computing and programming.

Richard Schumi and Jun Sun. 2021. SpecTest: Specification-Based Compiler Testing. *Fundamental Approaches to Software Engineering* 12649 (2021).

Sebastian Siegl, Kai-Steffen Hielscher, Reinhard German, and Christian Berger. 2011. Formal specification and systematic model-driven testing of embedded automotive systems. In *2011 Design, Automation & Test in Europe*. 1–6. https://doi.org/10.1109/DATE.2011.5763028

TC39. 2018. SIMD.js. https://github.com/tc39/ecmascript_simd.

Ben L. Titzer. 2022. A Fast In-Place Interpreter for WebAssembly. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 148 (oct 2022), 27 pages. https://doi.org/10.1145/3563311

W3C. 2019. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-1/.

wasm3. 2021. was-coremark. https://github.com/wasm3/wasm-coremark.

WasmCert. 2023. WasmRef-Isabelle. https://github.com/WasmCert/WasmCert-Isabelle.

Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing

Machinery, New York, NY, USA, 13 pages. https://doi.org/10.1145/3167082

Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Formal Methods*, Marieke Huisman, Corina Păsăreanu, and Naijun Zhan (Eds.). Springer International Publishing, Cham, 61–79.

Conrad Watt, Maja Trela, Peter Lammich, and Florian Märkl. 2023. Supplementary material for WasmRef-Isabelle. https://doi.org/10.5281/zenodo.7815663

WebAssembly Community Group. 2019. nontrapping-float-to-int-conversions. https://github.com/WebAssembly/nontrapping-float-to-int-conversions.

WebAssembly Community Group. 2020. multi-value. https://github.com/WebAssembly/multi-value.

WebAssembly Community Group. 2021a. bulk memory. https://github.com/WebAssembly/bulk-memory-operations.

WebAssembly Community Group. 2021b. reference types. https://github.com/WebAssembly/reference-types.

WebAssembly Community Group. 2021c. simd. https://github.com/WebAssembly/simd.

WebAssembly Community Group. 2021d. threads. https://github.com/WebAssembly/threads.

WebAssembly Community Group. 2022a. gc. https://github.com/WebAssembly/gc.

WebAssembly Community Group. 2022b. WebAssembly/spec/interpreter . https://github.com/WebAssembly/spec/tree/main/interpreter.

Simon Wimmer, Shuwei Hu, and Tobias Nipkow. 2018. Verified Memoization and Dynamic Programming. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10895)*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer, 579–596. https://doi.org/10.1007/978-3-319-94821-8_34

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 12 pages. https://doi.org/10.1145/1993498.1993532

Received 2022-11-10; accepted 2023-03-31