Formal Verification of WebAssembly via LLVM IR

Abstract

This work proposes a verification pipeline for WebAssembly programs using compiler-based analysis and SAT/SMT solving techniques. The approach leverages aWsm to translate WebAssembly modules into LLVM Intermediate Representation (LLVM IR), enabling integration with existing formal verification frameworks. The generated LLVM IR is then analysed using SMACK, which translates the program into Boogie and generates verification conditions for the Z3 SMT solver.

The proposed framework aims to enable automated reasoning about properties such as memory safety, control-flow correctness, assertion validity, and reachability within WebAssembly programs. By leveraging established LLVM- and Boogie-based verification infrastructures, the approach extends existing verification capabilities to WebAssembly while preserving the low-level semantics necessary for rigorous analysis.

The work identifies key challenges in the translation and verification process, including semantic preservation between WebAssembly and LLVM IR, handling of linear memory, and scalability of generated constraints. WebAssembly's unique characteristics, including its stack-based execution model, sandboxed linear memory, and cross-platform portability guarantees, introduce non-trivial verification challenges that are not fully addressed by existing tooling. The project lays the foundation for a practical verification workflow for WebAssembly based on existing compiler and solver technologies.

2012 ACM Subject Classification Theory of computation → Program verification; Theory of computation → Automated reasoning; Software and its engineering → Formal software verification

Keywords and phrases WebAssembly, formal verification, LLVM IR, SMACK, Boogie, SMT solving, Z3

Acknowledgements This work was carried out as part of a research project on compiler-assisted verification of WebAssembly programs.

1 Introduction

WebAssembly has emerged as a portable, efficient compilation target increasingly deployed beyond web browsers into cloud platforms, edge computing, embedded systems, and serverless environments [7]. As these deployments grow in security-sensitivity and complexity, formal verification of WebAssembly programs has become an important research challenge.

Existing verification approaches for WebAssembly often rely on specialised semantics or custom tooling [12], limiting interoperability with mature verification ecosystems developed for established compiler infrastructures. This work investigates an alternative: a compiler-assisted pipeline that translates WebAssembly into LLVM intermediate representation (IR) using aWsm [4, 6], then applies SMACK to transform LLVM IR into Boogie and generate verification conditions, which are finally checked by the Z3 SMT solver [10, 9, 1, 5]. The central question is whether this pipeline can effectively verify desired properties of the original WebAssembly program, and whether the translations themselves are lossy in ways that matter. Each translation step risks discarding structural information useful for verification; in particular, the compilation of source code into WebAssembly may obscure high-level properties such as memory allocation patterns, making them harder to reason about downstream.

A big motivation is reuse. Rather than constructing a dedicated verifier over WebAssembly semantics, the proposed approach leverages existing LLVM-based tooling and the Boogie verification infrastructure, reducing implementation effort while benefiting from the maturity

and scalability of established solver technologies. A key challenge is semantic preservation: since verification operates on translated LLVM IR and Boogie programs rather than the original WebAssembly module, the pipeline's effectiveness depends on whether relevant behavioural properties survive both compilation and verification-condition generation [13].

Contributions. The contributions of this paper are:

- A proposed compiler-assisted verification pipeline for WebAssembly using LLVM, SMACK, Boogie, and Z3.
- An analysis of aWsm as a translation mechanism from WebAssembly to LLVM IR.
- A discussion of SMACK/Boogie-based reasoning applied to compiled WebAssembly programs.
- Identification of key challenges relating to semantic preservation, memory modelling, and verification scalability.

2 Background

2.1 WebAssembly

WebAssembly is a low-level binary instruction format designed as a portable compilation target for high-level languages [7]. It uses a stack-based virtual machine architecture, organising programs into modules containing functions, linear memory regions, tables, and globals. Its sandboxed execution model and compact binary representation have driven adoption across web and non-web environments alike, though its low-level semantics complicate direct formal analysis [12, 11]. In particular, structured control constructs, an implicit operand stack, and a linear memory model must be accounted for before most off-the-shelf verification tools can reason about WebAssembly programs.

2.2 aWsm

aWsm is an ahead-of-time compiler and runtime for WebAssembly designed for execution outside browser environments [4, 6]. Rather than interpreting bytecode at runtime, aWsm translates WebAssembly modules into LLVM IR/bitcode, enabling native code generation and integration with LLVM-based analysis tools. It also incorporates runtime safety mechanisms intended to preserve WebAssembly's sandboxed execution guarantees, for example via inserted bounds checks and software-fault-isolation style instrumentation on memory accesses.

2.3 LLVM Intermediate Representation

LLVM IR is a language-independent intermediate representation used throughout the LLVM compiler framework [8]. Unlike WebAssembly's stack-based model, LLVM IR uses a register-based static single assignment (SSA) form, which simplifies data-flow analysis and symbolic reasoning. Its explicit control-flow, typed instructions, and widespread adoption make it a common target for verification tools and formal reasoning frameworks, including systems that check memory safety, arithmetic properties, and control-flow invariants.

2.4 SMACK and Boogie

SMACK is a verification toolchain for LLVM IR programs [10]. It translates LLVM programs into Boogie, an intermediate verification language designed as a common layer in program

verifiers [9, 1]. The Boogie tool then generates logical verification conditions in a first-order language with theories such as arithmetic and arrays, and invokes an SMT solver, typically Z3, to check their validity [3, 5]. By operating on LLVM IR rather than source languages directly, SMACK can support multiple compilation pipelines, making it well suited to the proposed approach in which aWsm supplies LLVM IR for WebAssembly modules.

2.5 SAT and SMT Solving

SAT solving determines whether a propositional formula can be satisfied by some variable assignment. SMT solving extends this by incorporating mathematical theories such as arithmetic, arrays, bitvectors, and memory models [2]. Program verification frameworks commonly rely on SMT solvers because software systems involve structured data and memory interactions that exceed the expressive power of propositional logic alone. Z3 [5] is among the most widely used SMT solvers and serves as the backend for Boogie in the proposed pipeline.

3 Motivation

WebAssembly's expanding deployment in security-critical environments creates a pressing need for formal verification support [13]. Yet verification tooling for WebAssembly lags behind that available for established compiler targets such as LLVM IR. Existing approaches frequently require dedicated semantics or custom frameworks [12], limiting their scalability and interoperability.

The LLVM ecosystem, by contrast, provides a mature collection of analysis and verification tools refined over many years [8]. Since aWsm already translates WebAssembly into LLVM IR, it creates a natural bridge: existing LLVM-based tools such as SMACK could potentially analyse WebAssembly programs indirectly through their compiled representations, without requiring a ground-up verification framework.

This approach may offer practical advantages in reduced implementation complexity, access to scalable solver technologies, and improved interoperability with existing program analysis workflows. The key open question is whether the pipeline can effectively verify desired properties end-to-end, given that each translation step risks discarding structural information relevant to verification. Direct verification over a bespoke formalisation of WebAssembly can in principle provide strong theoretical guarantees, but it often demands specialised tooling and may lag behind the rapidly evolving ecosystem of languages and runtimes targeting WebAssembly.

4 Proposed Architecture

The proposed pipeline translates WebAssembly programs into LLVM IR using aWsm, then applies SMACK to generate Boogie verification conditions resolved by Z3, reusing existing LLVM- and Boogie-based infrastructure throughout.

The pipeline consists of four primary stages:

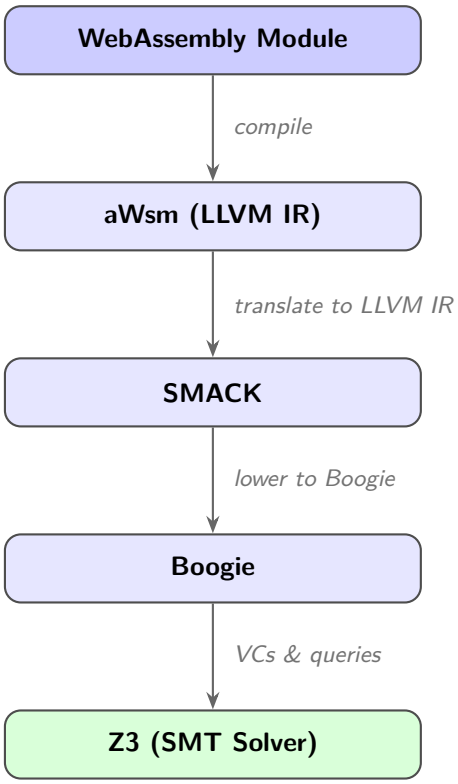

Figure 1 The four-stage verification pipeline.

A WebAssembly module is compiled by aWsm into LLVM IR via ahead-of-time translation. LLVM IR's explicit control-flow, typed instructions, and SSA form make it more amenable to symbolic reasoning than WebAssembly's stack-based format [8]. SMACK then consumes this IR and translates it into Boogie, an intermediate verification language tailored for expressing program semantics and verification conditions [10, 9, 1]. From the Boogie program, the Boogie tool generates logical constraints, which are finally analysed by the Z3 SMT solver to reason about properties including memory safety, assertion correctness, and control-flow reachability [2, 5].

Correctness depends on semantic preservation through compilation and translation: any divergence between WebAssembly, its aWsm-generated LLVM IR, the corresponding Boogie program, and the underlying logical model could yield verification results that do not reflect the original program's behaviour [13]. This work evaluates the feasibility of the pipeline and identifies the principal challenges to making it effective in practice.

4.1 Illustrative Example

To illustrate the proposed pipeline on a simple case, consider a WebAssembly function that returns the element at position i of an array stored in linear memory, provided that the index is in bounds. We model the array as a function

$$A : \{0, \ldots, m-1\} \to \mathbb{Z},$$

where $m \in \mathbb{N}$ is the array length, and the function takes an integer parameter $i \in \mathbb{Z}$. The intended behaviour is:

$$get(i) = \begin{cases} A(i) & \text{if } 0 \leq i < m, \\ \text{error} & \text{otherwise.} \end{cases}$$

The safety property of interest is that the function never performs an out-of-bounds read when called with a valid index.

In WebAssembly, this behaviour is implemented using the operand stack and a single access to linear memory [7]. The code computes the index i, checks the guard

$$0 \leq i < m,$$

and only then issues a load from linear memory at the computed address; if the guard fails, it triggers an assertion or jumps to an error-handling block. The informal safety requirement is: whenever a load is executed, its index i satisfies $0 \leq i < m$, so no out-of-bounds access occurs.

When the module is compiled by aWsm, this conditional load is translated into LLVM IR in static single assignment (SSA) form [8]. The implicit stack operations are replaced by explicit variables, and the bounds check becomes a branch on a Boolean condition such as

$$in\_bounds(i) \equiv 0 \leq i \wedge i < m.$$

If $in\_bounds(i)$ holds, the LLVM IR contains a load from an address derived from a base pointer p to the array and the index i; otherwise, control flows to an error block.

SMACK takes this LLVM IR and translates it into Boogie, preserving the structure of the conditional and the assertion [10, 9, 1]. From the Boogie program, the Boogie tool generates verification conditions that formalise the safety requirement. In simplified form, a verification condition capturing the absence of out-of-bounds accesses can be written as

$$\forall i.\ ExecLoad(i) \Longrightarrow 0 \leq i < m,$$

where $ExecLoad(i)$ is a predicate that holds exactly when the program executes a load from the array at index i on some execution. Together with a check that an assertion $\mathsf{assert}(0 \leq i < m)$ never fails at the load site, these conditions are passed to the SMT solver Z3, which attempts to prove them valid using theories of integers and memory structures [2, 5].

If Z3 succeeds in proving the generated conditions, we obtain evidence that the LLVM/Boogie-based implementation of this simple array-access function never performs an out-of-bounds read for any admissible input i. Assuming that aWsm and SMACK preserve the relevant observable behaviour when compiling from WebAssembly to LLVM IR and then to Boogie, this guarantee lifts back to the original WebAssembly code, showing that the example *get* function is memory-safe.

This guarantee, however, addresses only part of the verification problem that arises in practice. Suppose the WebAssembly module was itself compiled from a higher-level language such as C or Rust, in which the programmer wrote an array access that should stay within a declared array of length m. When that source program is compiled to WebAssembly, the source-level array becomes a contiguous region within WebAssembly's linear memory, but linear memory as a whole is a much larger address space shared by all data in the module. A verified absence of out-of-bounds access at the WebAssembly level therefore guarantees only that the load does not escape linear memory entirely, which is weaker than the source-level requirement that the access stays within the subrange $[b, b + m)$ allocated to the original array.

5 Challenges and Limitations

5.1 Semantic Preservation

The most significant challenge is semantic preservation across the various compilation and translation stages. WebAssembly's stack-based execution model and LLVM IR's register-based SSA form are fundamentally different [7, 8], and any behavioural discrepancy introduced by aWsm — through stack-to-SSA translation, runtime instrumentation, or safety mechanisms — could invalidate verification results. Subsequent translation from LLVM IR into Boogie must likewise preserve control flow, data dependencies, and memory semantics to ensure that properties proved at the Boogie level reflect the original WebAssembly program [10, 1].

One particularly delicate aspect is the compilation of structured control constructs and indirect function calls. WebAssembly's structured blocks, loops, and if-expressions enforce well-formed control-flow by construction, whereas LLVM IR and Boogie may introduce additional basic blocks, temporaries, and helper procedures to implement the same behaviour. The verification pipeline must ensure that these compilation artefacts do not introduce new observable behaviours or mask behaviours present in the original module, especially around error handling and abrupt termination [13].

5.2 Memory Modelling

WebAssembly's linear memory provides contiguous, bounds-checked, sandboxed storage [7, 11]. Mapping this faithfully into LLVM-compatible constructs and then into Boogie's memory model is non-trivial, particularly when reasoning about pointer arithmetic, aliasing, or low-level memory operations. Rich memory models also tend to produce large logical constraints, potentially limiting scalability for memory-intensive programs [2].

A further complication is the choice between precise and approximate aliasing models. A highly precise encoding of linear memory and pointer relationships can reduce spurious counterexamples but may explode solver runtime, whereas a more coarse-grained model improves scalability at the cost of potential false alarms. Choosing an appropriate level of abstraction for WebAssembly's memory semantics will be crucial for making the proposed pipeline practical.

5.3 Scalability of Verification

SAT/SMT-based verification is inherently computationally expensive [2]. Verification conditions can grow significantly for programs with loops, complex branching, or extensive memory interactions. While SMACK and Boogie provide scalable infrastructures [10, 1], the practicality of applying them to large-scale WebAssembly applications remains an open question.

Deeply nested control-flow, unbounded loops requiring invariants, and intensive heap usage can all lead to verification conditions that are difficult for SMT solvers to discharge automatically. Techniques such as modular verification, abstraction, and user-supplied invariants may be necessary to scale the approach beyond small kernels and micro-benchmarks.

5.4 Scope and Limitations

This work evaluates the correctness of the WebAssembly-to-LLVM translation, the suitability of LLVM/Boogie verification tools for WebAssembly semantics, and the scalability of gener-

ated verification conditions [13]. The proposal also focuses solely on sequential programs, excluding concurrency, multithreading, and asynchronous execution [11].

Moreover, the current scope omits interaction with host environments and system interfaces such as WASI, which are central to many real-world WebAssembly deployments. Precisely modelling host functions, sandbox boundaries, and isolation guarantees will be essential for future work that aims to reason about end-to-end security and correctness properties of deployed systems rather than standalone modules

6 Future Work and Evaluation Plan

Future work will develop a prototype pipeline integrating aWsm with SMACK and Boogie, compiling WebAssembly modules into LLVM IR, translating them to Boogie, and processing the result through Boogie and Z3 for verification condition generation and SMT-based analysis [10, 1]. A key area of investigation is semantic correspondence between the original WebAssembly program and the generated LLVM IR and Boogie programs, with particular attention to:

- Stack-to-SSA translation.
- Representation of linear memory.
- Control-flow preservation.
- Runtime safety checks introduced during compilation.

Evaluation will use representative WebAssembly benchmarks, such as arithmetic kernels, control-flow intensive programs, and memory manipulation examples compiled from C or Rust, to investigate whether existing LLVM/Boogie-based tools can successfully analyse compiled WebAssembly, which properties can be verified effectively, and how WebAssembly-specific features affect solver performance [13]. Beyond qualitative case studies, the evaluation should consider quantitative metrics such as solver runtime, the size and complexity of generated verification conditions, the proportion of properties proved versus those left unknown, and the frequency of spurious counterexamples [2].

Future work may also compare the proposed approach against direct WebAssembly verification techniques to assess practical advantages in scalability or implementation complexity. Extensions could include support for concurrent execution, WASI environments, host function modelling, and runtime isolation guarantees. Formal reasoning about translation correctness may also become necessary, potentially involving translation validation techniques between WebAssembly, LLVM IR, and Boogie [13].

7 Conclusion

This work has proposed a compiler-assisted verification pipeline for WebAssembly programs, combining aWsm ahead-of-time compilation with SMACK-based translation to Boogie, verification condition generation, and Z3 SMT solving. The approach is motivated by the growing deployment of WebAssembly in security-critical environments and the opportunity to reuse mature LLVM- and Boogie-based verification infrastructure rather than building a dedicated framework from scratch.

Key challenges include semantic preservation between WebAssembly, LLVM IR, and Boogie, linear memory modelling, and solver scalability, all of which bear directly on the soundness of the pipeline. While the work remains theoretical, it establishes a concrete foundation for future implementation and evaluation, and contributes toward more practical verification methodologies for WebAssembly software.

References

1 The boogie intermediate verification language. https://boogie-docs.readthedocs.io. Online documentation.

2 Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018. URL: http://theory.stanford.edu/~barrett/pubs/BT18.pdf, doi:10.1007/978-3-319-10575-8_11.

3 Boogie Contributors. Boogie. https://github.com/boogie-org/boogie, 2015. GitHub repository.

4 James Cadden et al. Making FaaS aWsm: An efficient serverless Wasm runtime for the edge. *Arm Developer Blog*, 2021. Describes the aWsm LLVM-based ahead-of-time WebAssembly compiler and runtime. URL: https://developer.arm.com/community/arm-research/b/articles/posts/making-faas-awsm-an-efficient-serverless-wasm-runtime-for-the-edge.

5 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

6 GW Systems. aWsm: An awesome Wasm compiler and runtime. https://github.com/gwsystems/aWsm, 2018. WebAssembly ahead-of-time compiler and runtime targeting LLVM bytecode.

7 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 185–200. ACM, 2017. doi:10.1145/3062341.3062363.

8 Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 75–86. IEEE, 2004. doi:10.1109/CGO.2004.1281665.

9 K. Rustan M. Leino et al. Boogie: An intermediate verification language. https://www.microsoft.com/en-us/research/project/boogie-an-intermediate-verification-language/, 2019. Microsoft Research.

10 Zvonimir Rakamaric and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014. doi:10.1007/978-3-319-08867-9_7.

11 Andreas Rossberg. WebAssembly core specification, version 2.0. Technical report, World Wide Web Consortium (W3C), 2022. URL: https://www.w3.org/TR/wasm-core-2/.

12 Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pages 53–65. ACM, 2018. doi:10.1145/3167082.

13 Hao Wen et al. Towards automated verification of WebAssembly programs. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023. doi:10.1109/ASE56229.2023.00188.