Verification of WebAssembly Programs

By

Diego Ignacio Ocampo Herrera
BSc Universidad Católica del Uruguay

A thesis submitted to Macquarie University
for the degree of
Master of Research
Department of Computing
May 2019

Typeset in LaTeX 2_ε.

Declaration

I certify that the work in this thesis entitled VERIFICATION OF WEBASSEMBLY PROGRAMS has not previously been submitted for a degree nor has it been submitted as part of the requirements for a degree to any other university or institution other than Macquarie University. I also certify that the thesis is an original piece of research and it has been written by me. Any help and assistance that I have received in my research work and the preparation of the thesis itself have been appropriately acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Diego Ignacio Ocampo Herrera

Abstract

WebAssembly is a new low-level language and compilation target mainly for the web that is already shipped in all major browsers in its minimum viable product version. The current version does not support exception handling, and therefore runtime errors cannot be handled inside the WebAssembly code.

Our main contribution of this research is the development of an approach that can detect runtime errors (traps) statically using Skink, a static analysis tool. To detect the possible traps, we: 1. translate WebAssembly (stack machine) into LLVM-IR (register machine), and 2. instrument the resulting code to reduce the problem of detecting traps to a reachability problem.

To test our solution, we use C/C++ benchmarks files from SV-COMP compiled into WebAssembly by Emscripten and compare the results against the standard verification process of C/C++ files by Skink. After successfully testing our approach, we apply our tool to verify programs that could abort execution due to runtime errors, detecting the conditions under which the error would occur.

Internet browsers could benefit from this approach in the future, as they will execute WebAssembly modules that originate from untrusted sources and possibly with malicious intentions. Our approach would then aid in the early detection of runtime errors of these WebAssembly modules.

Dedication

To Camila, without whom this journey wouldn't have been possible.

Acknowledgements

I would like to thank my supervisors, Franck Cassez and Anthony Sloane, for their guidance throughout this research project and for introducing me into the challenging world of software verification.

This research has been co-funded by International Macquarie University research Training Program Scholarship (CFMRTP) with ANII (The National Research and Innovation Agency of Uruguay), allocation 2017503.

Contents

List of Figures

List of Tables

1 Introduction

Web applications have become increasingly popular, replacing many of the traditional desktop applications such as e-mail clients, office suites, multimedia applications and other heavy computation applications. This massive exodus to the web is primarily due to the introduction of web standards such as HTML5 and increasing performance of JavaScript engines after the introduction of Just-In-Time (JIT) compilers. Consequently, web applications as online 3D software, online games and streaming platforms are becoming more sophisticated, demanding faster execution times and secure environments that can run untrusted code. However, as JavaScript is a dynamic language, its performance is nowhere close to native code execution performance [2].

Several attempts were made to enhance the performance of applications on the web, from early Java applets and ActiveX to the recent *asm.js* and Google's Native Client. Asm.js [3] is a specification of a strict subset of JavaScript, which resulted in more efficient JavaScript code when compiling C/C++ files with Emscripten [4]. This subset as a compilation target was promising, and even JIT compilers began to identify this code and optimise even further. However, JavaScript was never designed as a compilation target and still presents performance problems. Native client, on the other hand, was designed as a compilation target for the web and allowed the execution of native code. Nonetheless, it was only available for Google Chrome, and it has been discouraged in favour of WebAssembly due to its low adoption.

WebAssembly[1] is a new, standard and low-level programming language and compilation target for the web and has the aim of making the web faster, more reliable and secure. Currently, on its MVP version (Minimum Viable Product), it is already shipped in the major browsers, and several applications have already been ported to its platform, from game engines like Unreal Engine[2] to medical applications [5]. The WebAssembly programs that browsers run are from untrusted sources, and even though it runs sandboxed, these programs could have malicious intentions. Furthermore, in its MVP version, WebAssembly does not support exception handling, and therefore, every possible runtime error will abort the execution. In order to make WebAssembly programs more reliable and less

[1] https://webassembly.org/
[2] https://s3.amazonaws.com/mozilla-games/ZenGarden/EpicZenGarden.html

error-prone, they could be statically checked before they run in pursuance of finding and removing bugs or preventing malicious code from running. In this research project, we then deal with the verification of WebAssembly programs, either to determine if a specific runtime error would occur or if the algorithm violates some correctness specifications.

To achieve this, we first explored the practical feasibility of applying software verification on WebAssembly programs. Subsequently, we run the verification process over an instrumented intermediate representation in order to verify properties of the WebAssembly runtime and therefore, detecting possible runtime errors that would otherwise abort the execution.

1.1 Contributions

These are the main contributions of this research project are:

- We developed a Scala based WebAssembly parser: *ScalaWasm*[3]. This parser is generated using the *sbt-rats* parser generator [6] and is able to parse a comprehensive subset[4] of the WebAssembly language specification. This parser generates the Abstract Syntax Tree (AST) for a given WebAssembly module in its text format [5] based on the specification for version 1.0 or Minimum Viable Product (MVP)[6]. This parser is generic and does not add any metadata related to software verification and therefore can be used for any task that involves the AST of a WebAssembly module.

- We also designed a WebAssembly to Low Level Virtual Machine Intermediate Representation (LLVM-IR) translator, *Wasm2llvm*[7]. This translator uses *ScalaWasm* to obtain the AST of a WebAssembly module which is then processed, and an LLVM-IR file is generated. This tool is also capable of adding instrumentation code to the

[3] https://bitbucket.org/diegoocampohdr/scalawasm

[4] The set of covered instructions can be seen in the used sbt-rats grammar at: https://bitbucket.org/diegoocampohdr/scalawasm/src/default/src/main/scala/org/scalawasm/wasm/Wasm.syntax

[5] WebAssembly modules can be represented either in binary or text format

[6] https://webassembly.github.io/spec/core/bikeshed/index.html#text-format

[7] https://bitbucket.org/diegoocampohdr/wasm2llvm

output LLVM-IR in order to comply with the International Competition on Software Verification (SV-COMP[8]) specification and to verify runtime errors present in WebAssembly. This translator is described in detail in chapter 4.

- We have also developed techniques to detect specific types of runtime errors present in the WebAssembly runtime such as stack overflows, illegal memory access and conversion operations. By using this technique, web browsers could perform these kind of checks once an unsafe and untrusted WebAssembly module is transferred over the wire. In case the module presents errors, the execution could be prevented with a warning. This is covered in chapter 6.

- Overall we extended the capability of the static analysis tool Skink, providing two new frontends: WASM and EMCCWASM. The first allows Skink to run over WebAssembly modules in text format and the latter allows Skink to verify files that compile to WebAssembly using Emscripten.

1.2 Research aims

This research is focused on the applicability of software verification to WebAssembly programs and on verifying aspects that are particular to the WebAssembly language. By applying these techniques to WebAssembly programs, we are then able to prove that the verified programs are either correct or incorrect based on a specification. Correct programs are those in which the error state is not reachable while incorrect programs are those in which there are circumstances under which the error state is reachable. In case they are proven incorrect, a witness is provided specifying the conditions under which the program will fail. If the program is proven correct, we can then be sure that the program will not fail and that the assertions will not be violated under any circumstances.

In order to verify WebAssembly programs, several steps are involved. The main tool that we are using in this research is Skink [1], a static analysis tool that analyses LLVM-IR code to verify its correctness. As Skink performs its analysis on LLVM-IR, we

[8] https://sv-comp.sosy-lab.org

present two projects carried out in this research, introduced in section 1.1: 1. A Scala based WebAssembly parser, *ScalaWasm*. 2. A translator from WebAssembly into LLVM-IR, *Wasm2llvm*.

Once we can translate WebAssembly into LLVM-IR, we can then verify such programs using Skink. This is performed in two major and different steps:

1. Perform correctness analysis over the SV-COMP C files compiled into WebAssembly. As Skink can also analyse these C files by compiling them into LLVM, we can: (a) verify WebAssembly programs using Skink and the new developed frontends and, (b) by comparing the analyses results, detect problems in both the translation from WebAssembly into LLVM and possible bugs in Skink, as the generated LLVM-IR may differ from the original LLVM-IR from the C compilation while keeping the same semantics but comprised possibly of instructions of different complexity. The translation from WebAssembly into LLVM-IR can be formally verified but is not part of the scope of this research. These steps are described in the Methods section.

2. Instrument the resulting LLVM-IR from the translation in order to provide mechanisms to prove that certain errors that are not handled by the WebAssembly execution could be detected beforehand. As an example, these can be runtime errors that may occur in WebAssembly and for which there exists no mechanism to define exception handlers. WebAssembly does not support exception handling mechanisms so far [9].

The experiments that involve the validation of the translation and instrumentation and the verification applied to WebAssembly traps are organised as follows:

1. The extent of the covered WebAssembly specification and limitations and the translation into LLVM-IR and instrumentation are explained in chapter 4.

2. After the parser and translator are presented, the first experimental step is to validate the tool. This is developed in chapter 5.

3. The application of software verification to address the possible runtime errors of WebAssembly is developed in chapter 6.

[9] https://github.com/WebAssembly/proposals/issues/4

2 Literature review

2.1 WebAssembly

The web has become the most popular application platform and is growing in complexity, giving rise to increasingly demanding applications, such as video games and other 3D visualisation tools, high-quality video and audio streaming and cryptographic operations among others. With this in mind, efficiency, reliability, security and portability are of the utmost importance for the web to provide for these applications to fully flourish. However, the most common programming language of the web, not by convention but by historic choice, is the high-level JavaScript language, which cannot guarantee these requirements. In order to cope with these new challenges, *asm.js* [3] and *Native Client* [7] were introduced in the web without any standard and convention, leading to low adoption of the latter. This led to a significant consensus of the four web giants; Apple, Google, Microsoft and Mozilla to develop a standard approach to this problem: WebAssembly.

WebAssembly[1] is a new, standard and low-level binary instruction stack-based programming language for the web that is already shipped in the four major browsers: Firefox, Chrome, Safari and Edge [8]. Started in 2015, WebAssembly was designed to be the universal compilation target for the web to provide fast processing and near-native execution performance for web apps. WebAssembly provides a safe, fast, platform-independent and deterministic low-level language [2]. Even though the compiled WebAssembly modules are shipped in binary format, WebAssembly also supports text representation of the code that is easy to read, debug and process.

Using the Emscripten toolkit[2], C/C++/Rust programs can be compiled into WebAssembly together with the necessary environment for calling its functions from the JavaScript ecosystem. Consequently, those compiled functions will run at near-native speed. Compiled low-level code functions could be, for example, sorting algorithms and by using this technology, the performance is optimised. Notwithstanding being on the first release (cross-browser consensus in March 2017[3]), some applications are already using

[1] https://webassembly.org/
[2] https://github.com/kripken/emscripten
[3] https://webassembly.org/roadmap/

WebAssembly, such as a web-based cancer genome viewer [5], cryptography [9] and 3D visualizations [4]. Function calls can be made from JavaScript to WebAssembly and vice-versa, from the context of a Web application running on the client side (browser) or the server side (Node.js application). WebAssembly is, therefore, not replacing JavaScript but enhancing it by providing mechanisms of processing existing code bases at a faster speed than if those code bases were translated to JavaScript.

Previous attempts and solutions to provide a near-native code to the web environment such as *asm.js* and *Native Client* will be replaced by WebAssembly due to their lack of standards and, in the case of *Native Client*, low adoption. In contrast to its predecessors, WebAssembly was designed to be a compilation target for the web and to be embedded in a host environment [8]. JavaScript is a high-level language and was never designed to be a compilation target, i.e., to provide a safe and sound environment for compiling programs in C, for example, into JavaScript, to reuse existing low-level code on the web. *Asm.js* and *Native Client* among others provide such capabilities. *Asm.js* is the most used compilation target and a significant portion of JavaScript code on the web is generated by compiling C/C++ into *asm.js* [8]. *asm.js* is a subset of JavaScript which limits its features to a more optimizable language and therefore achieves better performance results than plain old JavaScript. Currently, C/C++ programs can be efficiently ported to the web using the Emscripten toolchain [5] to *asm.js*.

On the other hand, Google developed *Native Client*, which was released in September 2011. Native Client (NaCl) was the first sandboxed execution environment able to execute machine code on the web nearly as fast as native code [10]. This compilation target for the web did not receive as much adoption as *asm.js* and presented several difficulties to provide interoperability amongst all the browsers. Because of this scenario with multiple compilation targets, no standards, and the will of the major web companies, WebAssembly was designed to be a successor of the former.

[4] https://s3.amazonaws.com/mozilla-games/tmp/2017-02-21-SunTemple/SunTemple.html

[5] https://github.com/kripken/emscripten

2.2 Software Verification

Software verification was introduced by King [11] and Floyd [12]. The general idea and the foundations of the topic are going to be based on such references, the former one being a doctoral thesis and the latter a publication. Although these publications are quite old, they provide the pillars and the very foundation of the concepts in software verification. As of 2018, software is necessary and essential for modern societies. It is so ubiquitous that we can think of thousands of aspects of modern civilisation that are run by software, from controlling power-grids, public transport, communication to medical implants. Therefore, the level of dependency on such software is such that proving those programs correct has never been more critical.

Concerns about software being correct and being able to automatically prove it date back to 1970 with James King's doctoral thesis; "A program verifier". Software verification deals with proving that programs and algorithms will always execute as expected [11]. Automatic software verification is then the process of computers performing this kind of analysis and verification on programs, automatically. As software is more and more ubiquitous in our daily life, and some cases of mission-critical like pacemakers or insulin pumps, verifying unequivocally that such software will run correctly and as expected is critical. As a small example of a mission-critical system failing is NASA's Mars Climate Orbiter, which crashed in 1999 due to faulty software [13]. The error was introduced by using different unit systems and not performing the necessary checks, tests and verification of the algorithms, jeopardising a 327.6 million dollar mission. The US Department of Commerce estimated that the losses caused by avoidable errors in software oscillated around 20 and 60 billion USD annually, for the US economy only in 2002. A more recent report based on the impact for 2017 concluded that the financial losses are approximately 1.7 trillion dollars, worldwide, affecting more than 3.6 billion people [14].

In order to address the software verification problem, Antony Hoare, emeritus professor at Oxford University and a pioneer in the field, proposed the creation of a verifying compiler in the context of a "Grand Challenge for Computing Research" in 2003, planned to be achieved internationally and over the timespan of 15 years [15]. The proposed "verifying

compiler" is an enhanced compiler to which software verification capabilities are to be added in order to guarantee properties such as correctness, safety, termination and many other desirable properties. This was later presented at the conference on verified software, VSTTE (Verified Software: Theories, Tools, Experiments) [16], where the top experts in software verification participated.

2.2.1 Current approaches in automated program verification

This subsection will introduce modern advanced software verification techniques, software model-checking and tools, implementing the foundations presented in the previous subsection. For a more recent study and approaches into software verification, the survey from Silva [17] will be used to expand on the most modern techniques in the field. Beckert's survey [18] will also be used as a survey of existing tools used mostly in the industry, such as tools to verify real (not only experimental) Java programs and part of its API, for example. The current approaches in software verification can be categorised into interactive proof techniques and automated software verification. The first are the ones that need vast user interaction and effort to provide theorems and proofs of correctness or in some of the tools, annotate the source code heavily.

In this category, we can find deductive software verification, which is a formal technique for reasoning about program properties [18]. In this approach, the semantics of a program are thoroughly specified, from which later theorems are stated. The developer then has to provide a mathematical proof of such claims. These approaches are indeed effective but they require extensive user interaction and training. Among the most popular tools in this area are: Isabelle/HOL[6], ACL2[7] and Coq[8]. Model-Checking techniques do not require user interaction, and these are the kind of techniques that are going to be focused on in this research, and therefore the literature review will cover this category.

[6] https://isabelle.in.tum.de/
[7] http://www.cs.utexas.edu/users/moore/acl2/
[8] https://coq.inria.fr/

Static Analysis

The most common verification is static analysis. This kind of analysis is also done by compilers to perform several tasks such as dead code elimination and to prevent errors before compiling. Broadly, static analysis automatically computes information about the code and its behaviour without having to execute it [17]. Static analysis may not produce 100% accurate predictions or warnings about the code, but an approximation, as many of the properties that are to be verified can be undecidable or infeasible problems. Because of this approximate approach, some bugs can be missed (false negatives) and some spurious warnings (false positives) can be obtained as a result as well.

Concrete static analysis works by propagating the set of values of the variables until a fixpoint is reached. This is a theoretical aspect of accurate interpretation due to the large size value-sets that could be possibly infinite. These can grow fast, and a fixpoint may not be reached in a desirable amount of time. Therefore this technique does not scale well. This drawback of concrete interpretation led to the introduction of abstract interpretation by Cousot [19]. In the abstract interpretation of static analysis, the program's behaviour is analysed on an abstract domain (an approximation of concrete value-sets), and an abstract solution is obtained. In this regard, precision is replaced with efficiency. However, abstract interpretation cannot easily provide counterexamples or witnesses, and to remedy this problem, model checking is introduced by Clarke [20] and Queille [21].

Model checking

Model checking is a method for verifying that a system model satisfies a specification [20, 21]. The model consists of states and transitions. The properties can be safety properties such as assertion violations or buffer overflows. In comparison to static analysis, model checkers examine the reachable states of a program exhaustively and being aware of the path and program flow. The two approaches differ in their applicability in practice [17]. Model checkers were conceived to obtain precise results and provide a counterexample when a state that violates the correctness property is found reachable. In this case, the counterexample would be the trace that leads to such a state. This information is useful

for the developer, as the precise witness is indicating where the error would originate and the potential error can be solved.

This approach has the drawback that when the state space is large or possibly infinite, it is hard or infeasible to verify such models. Predicate abstraction was then introduced by Graf in [22]. With this technique, the state space is abstracted into a finite abstraction using a finite set of predicates. The predicate discovery process is where the complexity resides, as synthesising them defines the accuracy of the model checker. When an abstract counterexample is not realisable to the particular program, then the abstractions are refined to a more precise version. This process is called *counterexample-guided abstraction refinement* (CEGAR).

Model checkers use a range of solvers to analyse the feasibility of the predicates. These can be SAT solvers for boolean predicates, QBF (Quantified Boolean Formula) solvers and Satisfiability Modulo Theories (SMT) solvers. The latter is the most comprehensive and used as back-end in Skink [1], the tool that is used in this research to obtain the results.

SMT Solvers

SMT solvers are present in software verification and formal methods since 1980 and the interest in these has increased ever since to a point where they are largely present both in academia and in the industry [18]. These can be found in theorem provers such as HOL[9] and model checkers such as SLAM[10]. SMT solvers are currently an increasingly active area of software verification and instrumental in almost every tool in the SV-COMP (Software-Verification competition) [23]. For SMT and SAT solvers in particular, there are specific competitions to motivate research on these topics [11] [12].

SMT solvers work by analysing the satisfiability of predicates in a particular theory. These theories can be from set theories, lattices list or many other [18]. In the context of this research, SMT solvers are part of the tool that analyses if a program trace is satisfiable, or, in other words, if there exists a configuration of values for the variables that lead the

[9] https://hol-theorem-prover.org/
[10] https://www.microsoft.com/en-us/research/project/slam/
[11] http://www.satcompetition.org
[12] http://www.smtcomp.org

program to a state that violates the specification.

2.2.2 Trace abstraction refinement

Trace abstraction refinement is a new approach in software verification proposed by Heizmann [24]. In this new approach, the model checking is done using automata. In this automata, the set of statements or expressions of the program is the alphabet, and, each statement takes the place of a single letter, deprived of its meaning [25]. Different types of automata are to be used for different verification objectives, e.g., finite automata or nested word automata for recursion or Büchi automata for program termination [26]. In this approach, the first step is creating the automata based on a proof of infeasibility of a trace. For reachability cases, the automata would have a final error state. If there are traces that lead to such a state, the program would be incorrect. The initial abstraction is an automaton that accepts the language of all the possible sequence of statements of the program, given by the Control Flow Graph (CFG) extended with an error location.

In the next step, automata are generated for infeasible traces which are then removed from the initial trace abstraction using the techniques for regular languages subtraction defined in [24]. This is applied in an iterative CEGAR fashion. If by the end, no traces are left, it means that there are no feasible traces that lead to an error state and therefore the program is correct based on the proposed correctness model. Otherwise, if a feasible trace is found, then an error is found. As the problem is undecidable, this iterative process may never terminate and in practice, a timeout is set and once it is exceeded, an the result is inconclusive.

Applying trace abstraction refinement

The authors of the trace abstraction refinement techique have developed their own tool, UltimateAutomizer[13]. However, this tool analyses C programs and therefore we have chosen to use Skink [1], as it can analyse LLVM-IR and is developed by the Computing Department. As part of the review on tools that implement such theory, *Goanna* [27] is

[13] https://ultimate.informatik.uni-freiburg.de

also presented and compared with Skink, providing details of why Skink is more suitable for this current project.

Goanna, a commercial automated software verification tool is a software model checker based on trace abstraction refinement. Contrary to typical academic solutions, this tool can handle large and industrial sized C/C++ code bases for analysis. Goanna has been benchmarked by participating in the Software Assurance Metrics and Tool Evaluation [28], achieving interesting results: it detected the highest number of true weaknesses in security and quality metrics [27]. However, Goanna implements a simple trace abstraction refinement, while Skink performs a more elaborate technique.

Skink is the tool that will be used for the evaluation of the correctness of WebAssembly programs and gathering of the experimental results. Skink can process LLVM-IR programs through the analysis of their CFG, although it provides the option to be extended and perform the analysis on custom CFGs, say, from different languages. Skink is built using Mathias Heizmann's algorithm refinement of trace abstraction [29], which uses the CEGAR loop for refining the abstractions.

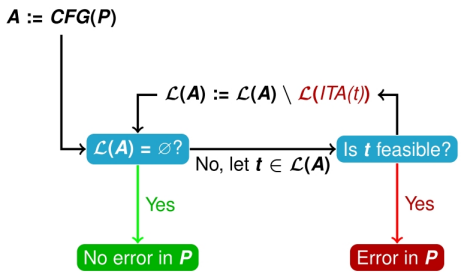

FIGURE 2.1: Refinement process. Credit [1].

The verification and refinement loop is shown in figure 2.1. The first step is obtaining the CFG of a program P and abstracting it into the automaton A. The loop is then iterated for all the traces generated by the automaton A, checking the feasibility of the traces. If the trace is feasible and leads to an error, then the code is declared incorrect, and a witness with the details of the configuration that leads to the error is provided. Infeasible traces that lead to an error state are removed from the automata abstraction using automata theoretic-operations. The feasibility of a trace is checked using SMT-solvers and in Skink using ScalaSMT [30], which supports popular solvers such as Z3 [31]. The loop may terminate with an automaton accepting the empty language $\emptyset$. Otherwise, it may end with the discovery of a feasible error trace, or might not come to a conclusion at all and halting because of the timeout, resulting in an "unknown" result.

3 Methods

3.1 Introduction

The primary methodology of this research is experimental; using WebAssembly programs and applying software verification techniques to prove our first claim: WebAssembly programs can be verified. The second objective of this research is to apply software verification to allow the detection of certain types of errors in WebAssembly. The focus of this research is not on testing the tool on large code bases but to try to verify as many runtime errors as possible. These are runtime errors that cause the execution to abort, named traps, are similar to exceptions in other programming languages. Exception handling is not a feature in WebAssembly, so far[1], meaning that the developer or the compiler cannot produce mechanisms of catching errors and taking a particular action on these events.

In order to run the verification tasks on WebAssembly programs, we used Skink. Skink provides a *C* frontend which uses *clang* to compile *C* programs into LLVM-IR programs and then uses the latter as the input. A diagram with the frontends is shown in figure 3.1 and explained later in that section. The resulting product of this research project is a set of tools that extend Skink and would aid in the validation phase of WebAssembly modules, preventing certain kind of runtime errors from occurring before execution.

The main decision concerning the methodology of this research was how the environment and tools, specially Skink, were to be used, set up and built. This is depicted in figure 3.1. The options were: 1. (left) Parse and translate WebAssembly into LLVM-IR, which is then used as an input for Skink using the existing machinery available for LLVM-IR. 2. (right) Modify Skink so that it also accepts WebAssembly direct input.

Implications of option 1: Skink already does the complicated processing of LLVM-IR and transformation into an automaton and further translation into SMT terms. In that

[1]Exceptions are part of the post MVP (Minimum Viable Product) features. This feature was assigned in October 2018, but no milestone or release has been planned. The issue is available at https://github.com/WebAssembly/proposals/issues/4.

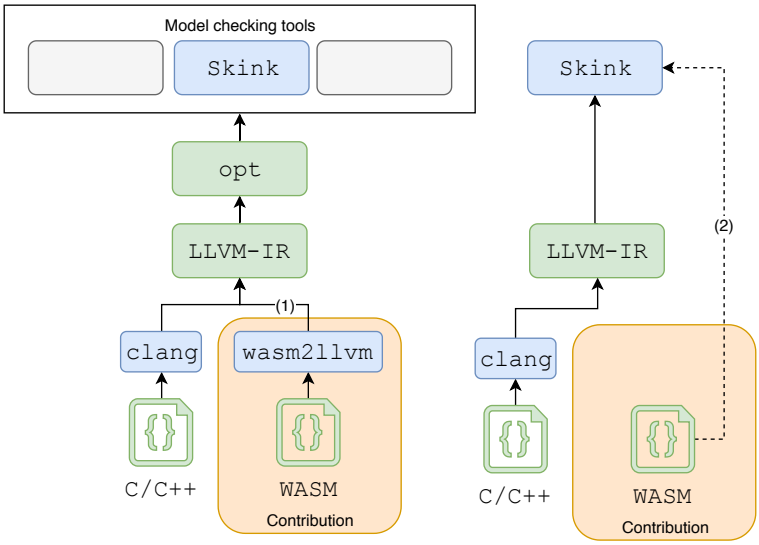

FIGURE 3.1: Tools and environment

regard, using the existing solution means that this challenging and time-consuming step is avoided. Skink has been tested extensively for reachability problems specifically and has participated in several instances of SV-COMP. Furthermore, the Skink project has been tested extensively, with a wide range of combinations of LLVM-IR instructions, from bit vector operations, floating point and loops to arrays. By taking this approach, we can build on top of the code that has been tested and used for years.

WebAssembly, as LLVM-IR, is an abstract low-level language. Almost every operation in WebAssembly can be directly translated into LLVM-IR. The main difference is that WebAssembly is a stack machine and LLVM-IR a register machine and the control flow is structured in WebAssembly where it is not in LLVM-IR. In this regard, the control flow in WebAssembly is a subset of that present in LLVM-IR and therefore it is translatable.

By translating WebAssembly into LLVM-IR, our tool is then independent of Skink, meaning that any static analysis tool that is capable of processing SV-COMP reachability problems in LLVM-IR would be able to use it and that there is no need to modify Skink. In this regard, Skink could still be used via the LLVM-IR frontend and using it as-is, without adding a frontend. Furthermore, using this approach, our solution would benefit from the improvements of the way Skink processes LLVM-IR and further changes in the LLVM specification that are incorporated into the tool.

With this approach, we can also benefit from the optimisations that the LLVM optimiser, *opt*, can apply to our resulting intermediate representation. These optimisations include function dead code elimination, loop unwinding and function inlining. Function inlining, which is the optimisation that replaces a function call by its body, is particularly useful in the task of verifying programs using Skink, as it is not capable of verifying multiple functions.

Implications of option 2: As mentioned in the implications of option 1, translating WebAssembly instructions into SMT terms and building the automata for further processing by Skink is not trivial. Firstly, the stack and the structured control flow labelling would have to be modelled into the SMT terms. Secondly, it requires an in-depth knowledge of the theories involved in the SMT terms. These theories can be the one that can represent floating point numbers and operations, for example.

By generating the terms from WebAssembly, this would be a new and untested code. Code that never participated in a software verification competition. Furthermore, that would require more time to develop a reliable tool, and given the limited timespan of this research, that would be out of scope.

Given the time constraint of this research, our emphasis on verifying aspects of WebAssembly and the benefits of option 1, we chose to parse and translate WebAssembly into LLVM-IR (*wasm2llvm*) to be further processed by software model checkers that can analyse LLVM-IR. No other options, such as verifying C source code were considered as both WASM and LLVM-IR are low level programming languages and with our approach we can also improve Skink. This is achieved by developing a WebAssembly parser (*scalaWasm*) and a tool that translates WebAssembly into LLVM-IR, *wasm2llvm*, that also performs modifications in order to transform instructions that cause possible traps in reachability problems. Table 3.1 shows a summary of the aspects to take into account for each option. In green are the positive aspects and in red the negatives.

Parameters	Option 1	Option 2
SMT terms semantics	Already done	Needs to be done
Reliability	Tested, participated in SV-COMPs	Needs to be tested from scratch
Independence from Skink	Yes	No
Access to optimizations	Yes	No

TABLE 3.1: Approach decision summary.

4 | Translating WebAssembly into LLVM

In the interest of being concise, a simplified version of the complete WebAssembly language specification used in the parser is presented in image 4.1. The complete supported syntax in *sbt-rats* format is available on the online repository[1]. Even though the parser is capable of parsing almost the entire WebAssembly specification, the translation to LLVM-IR does not cover the whole set of operations.

$$\begin{aligned}
\mathit{module} &::= \mathit{ftype}^* \ \mathit{import}^* \ \mathit{function}^+ \ \mathit{global}^* \ \mathit{export}^* \ \mathit{element}^* && (4.1)\\
\mathit{ftype} &::= \mathit{valueType}^* \rightarrow \mathit{valueType}^? && (4.2)\\
\mathit{resultType} &::= \textbf{result} \ \mathit{valueType} && (4.3)\\
\mathit{valueType} &::= \textbf{i32} \mid \textbf{i64} \mid \textbf{f32} \mid \textbf{f64} && (4.4)\\
\mathit{import} &::= \mathit{functionImport} \mid \mathit{globalImport} \mid \mathit{tableImport} \mid \mathit{memoryImport} && (4.5)\\
\mathit{function} &::= \mathit{ftype} \ \mathit{instr}^* && (4.6)\\
\mathit{instr} &::= \textbf{block} \ \mathit{resultType}^? \ \mathit{instr}^* \ \textbf{end} \mid \textbf{loop} \ \mathit{resultType}^? \ \mathit{instr}^* \ \textbf{end} && (4.7)\\
&\quad \mid \textbf{if} \ \mathit{resultType}^? \ \mathit{instr}^* \ (\textbf{else} \ \mathit{instr}^*)^? \ \textbf{end} && (4.8)\\
&\quad \mid \textbf{br} \ \mathit{index} \mid \textbf{br_if} \ \mathit{index} \mid \textbf{call} \ \mathit{index} \mid \textbf{call_indirect} \ \mathit{ftype} \mid \textbf{return} \mid \textbf{unreachable} && (4.9)\\
&\quad \mid \mathit{valueType}.\textbf{const} \mid \mathit{binOp} \mid \mathit{relOp} \mid \mathit{testOp} && (4.10)\\
&\quad \mid \mathit{valueType}.\textbf{load} \mid \mathit{valueType}.\textbf{store} \mid \textbf{memory.grow} \mid \textbf{memory.size} && (4.11)\\
&\quad \mid (\textbf{set} \mid \textbf{get} \mid \textbf{tee})\textbf{_local} \ \mathit{index} \mid (\textbf{set} \mid \textbf{get})\textbf{_global} \ \mathit{index} && (4.12)\\
&\quad \mid \textbf{drop} \mid \textbf{select} \mid \textbf{nop} && (4.13)\\
\mathit{binOp} &::= \textbf{i32.add} \mid \ldots \mid \textbf{f64.div} && (4.14)\\
\mathit{relOp} &::= \textbf{i32.eq} \mid \ldots \mid \textbf{f64.gt} && (4.15)\\
\mathit{testOp} &::= \textbf{i32.eqz} && (4.16)\\
\mathit{export} &::= \mathit{functionExport} \mid \mathit{globalExport} \mid \mathit{tableExport} \mid \mathit{memoryExport} && (4.17)\\
\mathit{index} &::= n \mid n \in \mathbb{N} \wedge n < 2^{32} && (4.18)
\end{aligned}$$

FIGURE 4.1: Simplified WebAssembly syntax.

WebAssembly programs are shipped in *modules* which can be in binary or text representation. Each file contains only one *module*, comprising several *functions*, metadata and data segments. Then, each function contains a sequence of instructions.

[1] https://bitbucket.org/diegoocampohdr/scalawasm/src/default/src/main/scala/org/scalawasm/wasm/Wasm.syntax

4.1 Overview

When performing the translation for WebAssembly functions, our goal is to translate each WebAssembly instruction into the LLVM-IR blocks that comprise the LLVM-IR function. In order to achieve this translation, we implement an intermediate representation with an inductive type we named Box, that models graph elements which have one entry and one exit. We implement different types of Box and each type expects different parameters, which are explained later in this chapter. Finally, we translate these boxes into LLVM-IR blocks. The first rule translates from a list of WebAssembly instructions into a Box:

$$C \vdash [\text{WebAssembly inst.}] \rightsquigarrow C' \vdash Box \tag{1}$$

The rule reads: given the list of WebAssembly instructions in the context C, we obtain the translation of such instructions in an instance of Box and a modified context C'. This rule applies to the whole set of instructions (see 4.1 lines 4.7 to 4.13), meaning that each instruction has a corresponding box. The context C stores information that is necessary to perform the translation, being the most important contents the operand stack (os) and the label stack (ls). As WebAssembly is a stack-based and LLVM-IR is register-based, we need to keep track of the WebAssembly stack and simulate it. The operations and contents of the context are described later in this chapter. We use a second main rule that translates from each Box into a list of LLVM-IR blocks:

$$lbl \vdash Box \Rightarrow [\text{LLVM-IR blocks}] \tag{2}$$

This second rule reads: given a Box, we generate a list of LLVM-IR blocks such that the last of such blocks jumps to the label lbl. This rule therefore defines the final step of the whole translation, generating and linking the LLVM-IR blocks that comprise each function. The type Box, defined in the next equation, can be composed of other boxes and a label (bl) or in the example of SimpleBox LLVM-IR instructions. The IfBox contains an inner Box for each possible path, which both exit to the final block of the IfBox. Each box then generates additional LLVM-IR blocks, defined by the previous rule and developed later.

$$
\begin{aligned}
Box ::= & \; SimpleBox(llvm, bl) \mid BrBox(llvm, bl) \mid RetBox(llvm, bl) \\
& \mid SeqBox(Box, Box, bl) \mid BlockBox(Box, Box, bl) \\
& \mid IfBox(c, ThenBox, ElseBox, bl)
\end{aligned} \tag{3}
$$

In figure 4.2, we show an example of the composition of boxes. The IfBox is composed of two inner boxes, one for each path and in the *then* path it contains a LoopBox and in the *else* a BlockBox. Loops are represented using BlockBoxes but we keep the name LoopBox to make the example clearer. The only difference between blocks and loops is how the branching behaves, which is discussed later. The IfBox then will be translated into an initial block that evaluates the conditional and performs a conditional branch to either the then box's first block or the else box's first block and a final block where both paths will merge.

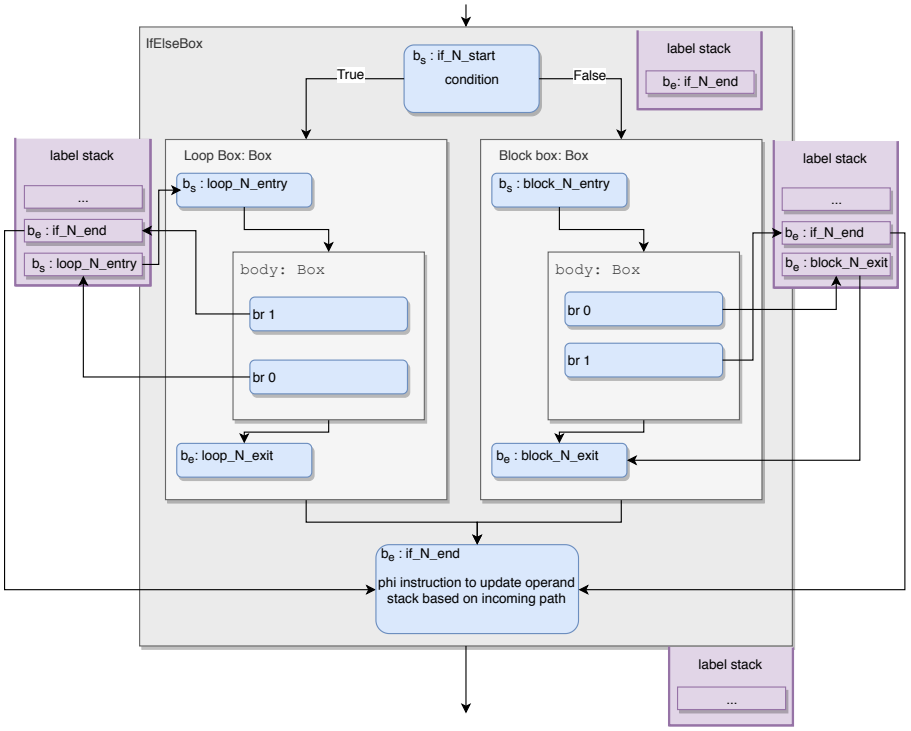

FIGURE 4.2: Box model and structured branching in the translation.

The image includes details about the labelling and the labels stack, that we explain later.

4.2 Details

The rule defined in equation 1 uses auxiliary rules, as boxes are built using LLVM-IR instructions. The next auxiliary rule translates a single WebAssembly instruction, given a context C, and obtains a list of LLVM-IR instructions and a modified context C'.

$$C \vdash \text{WebAssembly inst.} \rightarrow C' \vdash [\text{LLVM-IR inst.}] \tag{4}$$

The following image shows the specification of this rule for a variety of instructions. This rule is only applied for instructions that are not part of the structured control instructions. The latter are: `block`, `loop` and `if`.

$$\text{Context C} = (os, ls, b(\text{block label index}), funcs, locals, globals)$$

$$\overline{\text{'i32'} \rightarrow \text{'i32'}}\ \overline{\text{'i64'} \rightarrow \text{'i64'}}\ \overline{\text{'f32'} \rightarrow \text{'float'}}\ \overline{\text{'f64'} \rightarrow \text{'double'}}\ \overline{\text{'eq'} \rightarrow \text{'eq'}}\ \overline{\text{'ne'} \rightarrow \text{'ne'}}\ \overline{\text{'lt\_s'} \rightarrow \text{'slt'}}\ \overline{\text{'lt\_u'} \rightarrow \text{'ult'}}\ \overline{\text{'gt\_s'} \rightarrow \text{'sgt'}}$$

$$\overline{\text{'gt\_u'} \rightarrow \text{'ugt'}}\ \overline{\text{'le\_s'} \rightarrow \text{'sle'}}\ \overline{\text{'le\_u'} \rightarrow \text{'ule'}}\ \overline{\text{'ge\_s'} \rightarrow \text{'sge'}}\ \overline{\text{'ge\_u'} \rightarrow \text{'uge'}}$$

$$\frac{t \rightarrow t'\ \ C.os = [x]\ \ C' = C, os[y :: x]\ \ \text{fresh y}}{C \vdash t\textbf{.const}\ x \rightarrow C' \vdash [\text{'y = add } t'\text{ 0, x '}]} \qquad \frac{t \rightarrow t'\ \ irelop \rightarrow i\ \ C.os = [y :: x :: t]\ \ C' = C, os[z :: t]\ \ \text{fresh z}}{C \vdash t.irelop \rightarrow C' \vdash [\text{'z = icmp i } t'\text{ x, y'}]}$$

$$\frac{C.ls = [h :: x :: t]\ \ C' = C, ls[t]}{C \vdash \textbf{br}\ x \rightarrow C' \vdash [\text{'br label } x\text{'}]} \qquad \frac{C.os = [h :: t]\ \ C' = C, os[t]}{C \vdash \textbf{drop} \rightarrow C' \vdash []} \qquad \frac{}{C \vdash \textbf{nop} \rightarrow C \vdash [\text{'y = add } i\text{1 0, 0'}]}$$

$$\frac{C.os = []}{C \vdash \textbf{return} \rightarrow C \vdash [\text{'ret void'}]} \qquad \frac{C.os = [x]\ \ C' = C, os[])}{C \vdash \textbf{return} \rightarrow C' \vdash [\text{'ret x'}]} \qquad \frac{}{C \vdash \textbf{unreachable} \rightarrow C \vdash \left[\begin{array}{l}\text{'call void @\_\_VERIFIER\_error()'}\\ \text{'unreachable'}\end{array}\right]}$$

$$\frac{f = funcs(x)\ \ f.type = [] \rightarrow []}{C \vdash \textbf{call}\ x \rightarrow C \vdash [\text{'call void f.name()'}]} \qquad \frac{f = funcs(x)\ \ f.name = \text{'abortStackOverflow'}}{C \vdash \textbf{call}\ x \rightarrow C \vdash [\text{'call void @\_\_VERIFIER\_error()'}]}$$

$$\frac{l = locals(n)\ \ l.t \rightarrow t'\ C.os = [x :: t]\ \ C' = C, os[t]}{C \vdash \textbf{local.set}\ n \rightarrow C' \vdash [\text{'store } t'\text{ x, } t'\text{* } l.label\text{'}]} \qquad \frac{l = locals(n)\ \ l.t \rightarrow t'\ C.os = [t]\ \ C' = C, os[y :: t]\ \ \text{fresh y}}{C \vdash \textbf{local.get}\ n \rightarrow C' \vdash [\text{'y = load } t'\text{ x, } t'\text{* } l.label\text{'}]}$$

$$\frac{l = locals(n)\ \ l.t \rightarrow t'\ C.os = [x :: t])}{C \vdash \textbf{local.tee}\ n \rightarrow C \vdash [\text{'store } t'\text{ x, } t'\text{* } l.label\text{'}]} \qquad \frac{g = globals(n)\ \ g.t \rightarrow t'\ C.os = [x :: t]\ \ C' = C, os[t]}{C \vdash \textbf{global.set}\ n \rightarrow C' \vdash [\text{'store } t'\text{ x, } t'\text{* } l.label\text{'}]}$$

$$\frac{g = globals(n)\ \ g.t \rightarrow t'\ \ C.os = [t]\ \ C' = C, os[y :: t]\ \ \text{fresh y}}{C \vdash \textbf{global.get}\ n \rightarrow C' \vdash [\text{'y = load } t'\text{ x, } t'\text{* } g.label\text{'}]}$$

$$\frac{t \rightarrow t'\ \ C.os = [val :: addr :: t]\ \ C' = C, os[t]\ \ \text{fresh p1}\ \ \text{fresh p2}}{C \vdash t\textbf{.store}\ \rightarrow C' \vdash \left[\begin{array}{l}\text{'p1 = getelementptr inbounds [@mem.size x i8], [@mem.size x i8]* @mem, i32 0, i32 } addr\\ \text{'p2 = bitcast i8* } p1\text{ to } t'\text{*}\\ \text{'call void @\_\_VERIFIER\_memaccess(i32 } addr\text{, i32 } t.bitlength\text{) '}\\ \text{'store } t'\ val\text{, i8* } p2\end{array}\right]}$$

FIGURE 4.3: Implementation of rule defined in 4.

The first relations that we show in 4.3 are $\rightarrow$ and $\rightarrow$. The first is the relation between WebAssembly types and LLVM-IR types, such as `f64` in WebAssembly is represented as `double` in LLVM-IR. The second denotes the translation of the integer relational operation

keyword, which are quite similar, such as greater or equals signed `ge_s` maps to `sge`.

When referring to the operand stack, we use the notation $C.os[h :: t]$, meaning that the context C has an operand stack with h on its head and t on its tail, values that may be used in the translation. When modifying a specific part of the context, we use the notation $C' = C, os[t]$, meaning that we have a new context C' with the stack of C replaced by $[t]$ and keeping the rest of the values of C, to avoid explicitly repeating the rest of the values. The same notation applies for the modification of the label stack and rest of the values. We use the notation *fresh* y to indicate that a fresh variable y is created. As an example, the translation of the relational operations are defined with the rule:

$$\frac{t \rightharpoonup t' \;\; irelop \rightharpoonup i \;\; C.os = [y :: x :: t] \;\; C' = C, os[z :: t] \;\; \text{fresh } z}{C \vdash t.irelop \longrightarrow C' \vdash [\text{`}z = \text{icmp i } t' \text{ x, y'}]}$$

where the premises are indicating that the type t is translated into the LLVM-IR type t', then translate the keyword for the $irelop$ operation and given an operand stack with y and x on its head, we end up with the context C' which has the same contents as C but the stack is updated and now has z, a fresh variable, in its top. Finally, we build the LLVM-IR instruction using y, x, t', i and z.

The rule defined in equation 1 also uses the following auxiliary rule, which is similar to 1 and reads the same but applies to a single WebAssembly instruction and therefore the different arrow ($\rightsquigarrow_1$).

$$C \vdash \text{WebAssembly inst.} \rightsquigarrow_1 C' \vdash Box \tag{5}$$

This rule defines how each WebAssembly instruction is translated into a box, which may contain other boxes or just LLVM-IR instructions. The ones that will contain inner boxes are the structured control instructions, which will use rule 1 in its premises to perform the recursive instruction. In figure 4.4, we define relations 1 and 5.

When translating the inner instructions of structured control instructions, the operand stack needs to be empty, as instructions inside these constructs cannot access the stack elements pushed outside and a label has to be pushed to the label stack. To model this, we use $C' = C, os[], ls[bl_e :: C.ls]$, meaning that we perform the recursive translation

with a new context with an empty operand stack and a label appended to the top of the original $C.ls$.

$$\{SI\} = \text{Simple instructions, 4.10 to 4.13 in 4.7}$$

$$\frac{i = \textbf{br}\ x \quad C \vdash i \rightarrow C' \vdash i' \quad C', b(C.b+1)}{C \vdash i \rightsquigarrow_1 C' \vdash BrBox(i', C.b)} \qquad \frac{i = \textbf{return} \quad C \vdash i \rightarrow C' \vdash i' \quad C', b(C.b+1)}{C \vdash i \rightsquigarrow_1 C' \vdash RetBox(i', C.b)} \qquad \frac{i \in \{SI\} \quad C \vdash i \rightarrow C' \vdash i' \quad C', b(C.b+1)}{C \vdash i \rightsquigarrow_1 C' \vdash SimpleBox(i', C.b)}$$

$$\frac{bl = C.b \quad C.ls = [t] \quad C' = C, os[], ls[bl_e :: t], b(bl+1) \quad C' \vdash e^* \rightsquigarrow C'' \vdash bx \quad C''' = C, b(C''.b)}{C \vdash \textbf{block}\ e^*\ \textbf{end} \rightsquigarrow_1 C''' \vdash BlockBox(bx, bl)}$$

$$\frac{bl = C.b \quad C' = C, os[], ls[bl_s :: C.ls], b(bl+1) \quad C' \vdash e^* \rightsquigarrow C'' \vdash bx \quad C''' = C, b(C''.b), os[C''.os :: C.os]}{C \vdash \textbf{loop}\ rt\ e^*\ \textbf{end} \rightsquigarrow_1 C''' \vdash BlockBox(bx, bl)}$$

$$\frac{bl = C.b \quad C.os = [cond :: x] \quad C' = C, os[], ls[bl_e :: C.ls], b(bl+1) \quad C' \vdash e^* \rightsquigarrow C'' \vdash bx \quad C''' = C, b(C''.b)}{C \vdash \textbf{if}\ e^*\ \textbf{end} \rightsquigarrow_1 C''' \vdash IfBox(cond, bx, bl)}$$

$$\frac{\begin{array}{ccc} C' = C, os[], ls[bl_e :: C.ls], b(bl+1) & C' \vdash e_1^* \rightsquigarrow C'' \vdash ThenBox & C'' \vdash e_2^* \rightsquigarrow C''' \vdash ElseBox \\ bl = C.b & C.os = [cond :: x] & C'''' = C, b(C'''.b) \end{array}}{C \vdash \textbf{if}\ e_1^*\ \textbf{else}\ e_2^*\ \textbf{end} \rightsquigarrow_1 C'''' \vdash IfBox(cond, ThenBox, ElseBox, bl)}$$

$$\frac{C \vdash i \rightsquigarrow_1 C' \vdash box}{C \vdash [i] \rightsquigarrow C' \vdash box} \qquad \frac{C \vdash h \rightsquigarrow_1 C' \vdash hBox \quad C' \vdash t \rightsquigarrow C'' \vdash tBox}{C \vdash [h :: t] \rightsquigarrow C'' \vdash SeqBox(hBox, tBox, hbox.label)}$$

FIGURE 4.4: Implementation of rule defined in 1 and 5.

Control flow in WebAssembly is structured, meaning that branching can only jump to the enclosing constructs and not to arbitrary locations in the code, behaviour that is translated using the label stack. In the context of the recursive translation of the inner instructions of a `block`, we push the block's label to the labels' stack bl_e and on the case of the `loop` instruction, the subtle difference that we push bl_s. This is because when branching with index 0 inside a `block` or `if` in WebAssembly, a jump to the end of the `block` is performed, but on the case of a `loop`, a jump to the beginning of the `loop` is performed, in order to perform a new iteration. This is depicted in image 4.2, where `br n` is then translated into a `branch label` n', where n' is determined using the label stack maintained in the context.

In the case of a loop, the label that was pushed to the label stack indicates a backwards jump and in the case of a block a forward jump, based on the targets indicated by the labels stack. As we can see in the figure, we have BlockBox as one of the inner boxes of the IfBox, which is comprised of a start LLVM-IR block with the label b_s, then links to its inner Box with the translation of its instructions, which then jump to the block's exit llvm

block, with the b_e label.

The two rules at the end define the translation for a sequence of WebAssembly instructions. In the first case, when translating a list with a single element, $[i]$, we use the relation $\rightsquigarrow_1$. Otherwise, we use $\rightsquigarrow_1$ to translate the head of the list, obtaining the head box and recursively translating the tail with $\rightsquigarrow$ and at the end, a sequence of WebAssembly instructions gets translated into a single Box, which is a SeqBox.

The next figure implements the rule in equation 2. This rule has no context, as the values in the context are used in the creation of boxes.

$$\frac{lbl \vdash tBox \Rightarrow llt \qquad llt[0].label \vdash hBox \Rightarrow llh}{lbl \vdash SeqBox(hBox, tBox, label) \Rightarrow llh ++ llt} \qquad \frac{}{lbl \vdash SimpleBox(i, label) \Rightarrow LLBlock(label, i, \text{`br label lbl '})}$$

$$\frac{}{BrBox(i, label) \Rightarrow LLBlock(label, [], i)} \qquad \frac{}{RetBox(i, label) \Rightarrow LLBlock(label, [], i)}$$

$$\frac{\begin{array}{c} blockEnd = LLBlock(label_e, [], \text{`br label lbl '}) \qquad label_e \vdash innerBox \Rightarrow innerLlvm \\ \text{fresh label\_e label\_s} \qquad blockStart = LLBlock(label_s, [], \text{`br label innerLlvm.[0].label '}) \end{array}}{lbl \vdash BlockBox(innerBox, label) \Rightarrow [blockStart] ++ innerLlvm ++ [blockEnd]}$$

$$\frac{\begin{array}{c} label_e \vdash thenBox \Rightarrow thenLlvm \qquad label_e \vdash elseBox \Rightarrow elseLlvm \\ blockCond = LLBlock(label_s, [], \text{`tmp = icmp ne i32 c, 0'}, \text{`br i1 tmp, label thenLlvm.[0].label, label elseLlvm.[0].label '}), \\ blockEnd = LLBlock(label_e, [], \text{`z = phi [x, thenLlvm.last.label], [y, elseLlvm.last.label]'}, \text{`br label lbl '}) \qquad \text{fresh label\_e label\_s} \end{array}}{lbl \vdash IfBox(c, thenBox, elseBox, label) \Rightarrow [blockCond] ++ thenLlvm ++ elseLlvm ++ [blockEnd]}$$

FIGURE 4.5: Translation rules from boxes into final LLVM-IR.

This final rule builds the LLVM-IR blocks and links then. An LLVM-IR Block constructor, defined as $LLBlock(bl, phi^?, i^+, brI)$, expects a label (bl), an optional *phi* instruction, the instruction in the body i^+ and a final branching instruction. Phi instructions in LLVM-IR are a mechanism to load a variable depending on the incoming edge to the current block. When we have a conditional that opens two paths and then merges them at the end, we need to know what value to assign to value on top of the stack when the conditional yields a result. In such a case, it is required in WebAssembly that both paths return the same type. Therefore, the type of the new variable is resolved in the translation of the `if` instruction and the stack is modified in that occasion. At this point, we only need to build the phi instruction that will load such variable.

The lbl on the left-hand side of the rule is calculated recursively for the sequence of boxes in the rule in the top left, where lbl takes the value of the boxes' labels. We use the notation $llt[0].label$ to access the label of the LLVM-IR block at position 0 of the list of llt and in this way all the boxes in the sequence end up linked, and the last linked to lbl.

In the example of the IfBox, the inner boxes are translated and are both linked to the final block denoted with $label_e$. Then the conditional block is generated, which evaluates the conditional and performs a conditional branch to either the then box's first block or the else box's first block. Then a final block is created, that merges the two paths, and in the case that the conditional returns a value, a *phi* instruction is generated to choose the new operand at the top of the stack depending on the incoming edge. The phi instruction needs to know the label of each predecessor, and we use the notation $thenLlvm.last.label$ to access the label of the last LLVM-IR block of the *then* path.

The introduced rules deal with the translations of the sequence of instructions that comprise WebAssembly functions. The translation of the WebAssembly functions AST node into LLVM-IR functions is not done using the box model, as they do not follow the single-entry single-exit model. For the translation of a WebAssembly function into an LLVM-IR function, the inner sequence of instructions is translated into boxes and then into the final LLVM-IR blocks using the defined relations. The translation of a function will generate an extra initial and final block. The final block's label is used for the rule defined in figure 4.5, and is used in lbl. In this way, the last box will jump to the function's final block. Then, the first block with a unique label identifying the function, allocating the memory for local variables and branching to the first block of the translation of the inner box is generated. In the last block, the function then returns the value that is present on the top of the operand stack or returns void.

WebAssembly currently allows only one memory instance. Memory is translated into a byte array and initialised to zero, see line 1 of listing A.12 for an example. Global variables in WebAssembly are directly translated to LLVM-IR and placed in the top of the file, see the first lines of listing A.15 for examples. Both globals and memory can be initialised in the embedding environment, but as we only verify WebAssembly modules in isolation, when these values are imported, we initialise them to 0.

5 Validating the translation

5.1 Introduction

The approach we took to validate our translation with *wasm2llvm* tool is by using unit tests instead of formal proof. Establishing a proof of correctness of the translation *wasm2llvm* is hard and involves non-trivial techniques and, as mentioned in chapter 3, given the time constraints of this research, the viable approach is using test cases instead. For the test cases, files obtained from previous instances of SV-COMP were used. The SV-COMP setting and definitions are defined in the following section. In this chapter, we present the results of this validation phase and a discussion of such results.

5.2 SV-COMP setting

SV-COMP is the International Competition on Software Verification that has recently run its eighth event at Prague, Czechia. This competition defines benchmarks and verification tasks in order to assess a wide range of state-of-the-art verification tools [32]. A verification task is defined as a non-interactive C file which is purely computational and therefore has no side-effects and its *main* function has no parameters. In the context of SV-COMP, these C files are not intended to be executed but to be verified. The C file includes a correctness specification. For the category that we deal with, *reachability*, the specification consists of error states determined by calls to a special function called `__VERIFIER_error()`.

In reachability tasks, the verification tool has to determine if the error state(s) in the program can be reached, and in the case that the reachability depends on a specific value, such value would be the violation witness for specification provided in the scenario of an incorrect program. The code in listing 5.1 combines an example of reachability that depend on external values. Calls to `__VERIFIER_error()` are in essence assertions of reachability. In a real C file, an assertion would have no effect other than debugging purposes. As defined by SV-COMP, the call to `__VERIFIER_error()` aborts the execution and the control flow never returns. In this regard, when computed the CFG of the program, the call to the error state would be a final error state.

```
1 extern void __VERIFIER_error() __attribute__ ((__noreturn__));
2 int __VERIFIER_nondet_int();
3 int main (){
4         int m = 0;
5         int i = __VERIFIER_nondet_int()
6         int j = __VERIFIER_nondet_int();
7         if (i > j)
8                 if (i > j) __VERIFIER_error();
9 }
```

Listing 5.1: Sample incorrect program from SV-COMP

Some verification tasks, as 5.1, include function calls to obtain nondeterministic values that may alter the control flow. These functions have the type `__VERIFIER_nondet_X()` where X is the type of the value that the function returns. The *X* type can be an integer, a floating point values, among others. In the cases such as listing 5.1, the verification consists of determining the conditions under which the `call void __VERIFIER_error()` is reachable, based on the possible values that variables *i* and *j* can take. When Skink verifies this file as *INCORRECT*, it also provides the witness, specifying each of the values that `__VERIFIER_nondet_int()` needs to return for the program to fail. The test cases used in this validation phase involve files SV-COMP that cover different language constructions, while always addressing reachability problems. These constructions cover loops, (nested) conditionals, arrays, strings, floats/doubles, integers and arithmetic operations.

5.3 Validation

The validation process is depicted in figure 5.1. In the *current Skink workflow*, the process that Skink uses to verify *C* programs and produce a result is shown. In this process, C files are compiled and optimised with *clang* and then the resulting LLVM-IR is processed by Skink to analyse its correctness. This set of files are the ones currently used to test Skink extensively. The same set of files is then applied to our workflow, the one beneath the *current Skink workflow*. Our workflow then compiles the C files into WebAssembly using *Emscripten (emcc)*[1]. Then the WebAssembly file gets translated and instrumented by

[1] http://kripken.github.io/emscripten-site/

wasm2llvm into LLVM-IR, which is further optimised with *opt* in order to inline functions and optimise the code.

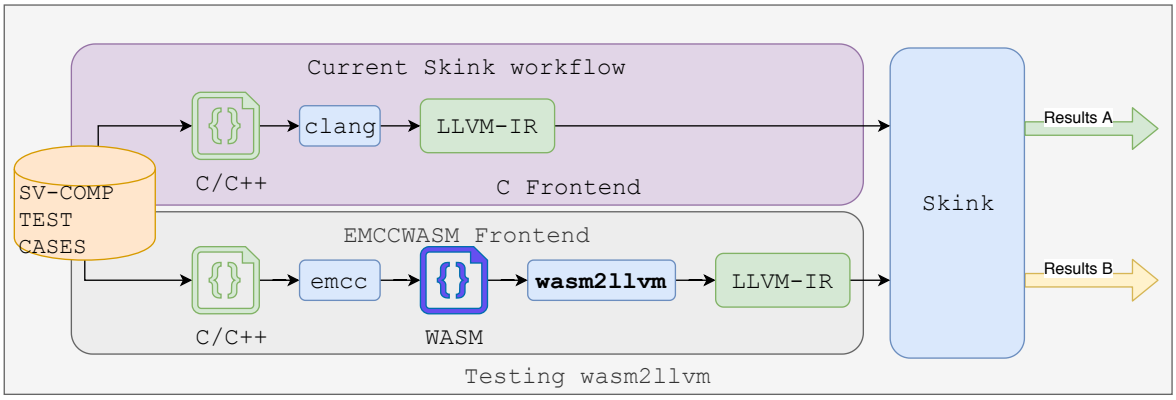

FIGURE 5.1: Validating the translation.

In our tests we use different levels of optimisations. The C frontend relies on the optimisation done by *clang* and in the case of our translation, we use *opt*. In EMCCWASM, additionally, default optimisation is included in the *emcc* step as a means to get rid of WebAssembly runtime code that is not used. Following the execution of both workflows, we then compare the results.

Skink produces three possible results: • *TRUE*: The program is correct and `__VERIFIER_error()` is not reachable. In some cases, the optimizer can get rid of unreachable/dead code that contains the error state, and in these cases, Skink concludes the correctness without effort. It is in these cases that the optimisation takes less time than the actual verification and that is reflected in the results and discussed later in this chapter. • *FALSE*: The program is incorrect, `__VERIFIER_error()` is reachable, and a witness for the values that may produce the error is produced as well. This occurs when the program involves a call to a function that returns a value which is then used in the control flow of the program. The witness reveals the sequence of possible values returned by the non-det-calls. • *UNKNOWN*: Skink could not determine the correctness status of the program.

In our evaluation we then categorise a test case as Passed if the tool produces the expected SV-COMP verification result for the file (correct or incorrect), Failed if the tool produces the wrong result and Unknown if the tool cannot decide. The results of the

validation are shown in table 5.1. The column *C Frontend* correspond to the *current Skink workflow* in figure 5.1 whereas the results under the column *EMCC Frontend* correspond to the second workflow. Verification tasks Nº 41, 42 and 43 are only run on -02 as originally they contain multiple functions, which Skink cannot process. The tests were conducted with the host configuration and tools listed in table D.1.

5.4 Discussion

The main result to highlight is that no false positives or false negatives result from the execution on both frontends on different optimisation levels. Therefore, no program was incorrectly categorised. Table 5.2 shows the distribution of the percentages for correctly verified programs (**% Correct**) vs those that resulted in *unknown* (**% Unknown**).

As presented in table 5.2, we obtained less cases of *unknown* by using the new frontend, EMCCWASM. With EMCCWASM only 3.51% resulted in *unknown* while the remaining 96.49% where correctly verified. The percentage of unknowns is 0% when not optimising and 6.67% when optimising the LLVM-IR. With regards to the C frontend, it resulted in almost five times the amount of unknowns than our frontend, reaching a 17.54% of unknowns. However, with the optimizer at *-O2*, only 3.33% resulted in *unknown*.

With regards to the average execution times, shown in milliseconds per frontend and optimisation level, it is important to highlight that when the optimiser is on, the average time is lower than when it is off for both frontends. This indicates that the optimiser improvement of the code, making it simpler and in some cases getting rid of the unreachable error states takes less time than it would take for Skink to complete the task. This fact highlights how beneficial can be the optimiser in the execution of the verification task and seems to indicate that our choice for re-using the LLVM-IR with the optimiser, discusses earlier in 3.

				C Frontend		EMCCWASM Frontend	
#	**File**	**Desc.**	**Opt.**	**Time (ms.)**	**Result**	**Time (ms.)**	**Result**
1	array-hard_true-unreach-call.c	Arrays	-O0	1,787	Passed	7,286	Passed
2	-	-	-O2	53	Passed	6,231	Passed
3	array-sequence_true-unreach-call.c	Arrays	-O0	86	Passed	5,324	Passed
4	-	-	-O2	46	Passed	5,139	Passed
5	count-up-down_false-unreach-call.c	While loop	-O0	5,027	Passed	6,524	Passed
6	-	-	-O2	1,046	Passed	6,518	Passed
7	count-up-down_true-unreach-call.c	While loop	-O0	N/A	Unknown	5,881	Passed
8	-	-	-O2	77	Passed	5,376	Passed
9	eca-like_false-unreach-call.c	ECA	-O0	988	Passed	6,058	Passed
10	-	-	-O2	927	Passed	5,906	Passed
11	float_false-unreach-call.c	Floating point	-O0	1,371	Passed	8,316	Passed
12	-	-	-O2	1,314	Passed	7,332	Passed
13	fpspecial_true-unreach-call.c	Floating point	-O0	12,136	Passed	7,106	Passed
14	-	-	-O2	2,581	Passed	6,885	Passed
15	getelementptr_true-unreach-call.c	Structs	-O0	N/A	Unknown	11,051	Passed
16	-	-	-O2	107	Passed	13,330	Passed
17	getelementptr1_false-unreach-call.c	Structs	-O0	N/A	Unknown	6,573	Passed
18	-	-	-O2	1,154	Passed	7,695	Passed
19	getelementptr2_false-unreach-call.c	Structs	-O0	N/A	Unknown	6,279	Passed
20	-	-	-O2	954	Passed	6,211	Passed
21	getelementptr3_false-unreach-call.c	Structs	-O0	N/A	Unknown	6,202	Passed
22	-	-	-O2	947	Passed	6,191	Passed
23	getelementptr4_false-unreach-call.c	Structs	-O0	N/A	Unknown	9,635	Passed
24	-	-	-O2	986	Passed	7,753	Passed
25	getelementptr5_false-unreach-call.c	Structs	-O0	N/A	Unknown	7,707	Passed
26	-	-	-O2	1,168	Passed	7,601	Passed
27	global_init_true-unreach-call.c	Arrays/Strings	-O0	4,563	Passed	6,126	Passed
28	-	-	-O2	1,957	Passed	N/A	Unknown
29	implicitunsignedconversion_false-unreach-call.c	Unsigned Int	-O0	1,941	Passed	8,613	Passed
30	-	-	-O2	1,356	Passed	8,744	Passed
31	implicitunsignedconversion_true-unreach-call.c	Unsigned Int	-O0	88	Passed	5,187	Passed
32	-	-	-O2	48	Passed	5,236	Passed
33	multi_float_false-unreach-call.c	Floating point	-O0	1,177	Passed	7,322	Passed
34	-	-	-O2	995	Passed	7,114	Passed
35	multi_int_false-unreach-call.c	Unsigned Int	-O0	1,021	Passed	5,970	Passed
36	-	-	-O2	901	Passed	6,150	Passed
37	multi_true-unreach-call.c	Int multiplication	-O0	228	Passed	6,400	Passed
38	-	-	-O2	63	Passed	5,738	Passed
39	multiple-error-calls_false-unreach-call.c	Conditionals	-O0	1,312	Passed	6,458	Passed
40	-	-	-O2	954	Passed	6,505	Passed
41	newton_1_1_true-unreach-call.c	Polynomials	-O2	N/A	Unknown	N/A	Unknown
42	simple-function_false-unreach-call.c	Function calls	-O2	1,145	Passed	7,315	Passed
43	simple-function_true-unreach-call.c	Function calls	-O2	73	Passed	6,289	Passed
44	simple-if_false-unreach-call.c	Nested conditionals	-O0	1,038	Passed	6,085	Passed
45	-	-	-O2	949	Passed	6,037	Passed
46	simple-if_true-unreach-call.c	Nested conditionals	-O0	315	Passed	5,161	Passed
47	-	-	-O2	55	Passed	5,232	Passed
48	simple-loop_false-unreach-call.c	While loop	-O0	N/A	Unknown	7,088	Passed
49	-	-	-O2	1,149	Passed	7,364	Passed
50	simple-loop_true-unreach-call.c	While loop	-O0	N/A	Unknown	5,754	Passed
51	-	-	-O2	66	Passed	6,293	Passed
52	simple-loop-array_true-unreach-call.c	While loop/Arrays	-O0	3,791	Passed	7,012	Passed
53	-	-	-O2	49	Passed	6,270	Passed
54	test-interpolant-franck_true-unreach-call.c	Interpolant	-O0	4,321	Passed	5,777	Passed
55	-	-	-O2	65	Passed	6,259	Passed
56	unsigned_true-unreach-call.c	Unsigned Ints	-O0	83	Passed	5,169	Passed
57	-	-	-O2	50	Passed	5,075	Passed

TABLE 5.1: Validation of the EMCCWASM frontend against the C frontend

The set of files that produced *unknown* by EMCCWASM is almost a subset of the set of the files that produced *unknown* by the C frontend, except for test case N°28. For those

Frontend	Opt.	Correct	Total	% Unknown	% Correct	Time (ms.) avg.
C	-O0	18	27	33.33%	66.67%	2,293
	-O2	29	30	3.33%	96.67%	732
C Total		47	57	17.54%	82.46%	1,330
EMCCWASM	-O0	27	27	0.00%	100.00%	6,743
	-O2	28	30	6.67%	93.33%	6,707
EMCCWASM Total		55	57	3.51%	96.49%	6,725
Grand Total		102	114	10.53%	89.47%	4,027

TABLE 5.2: Distribution of percentages for results and execution times.

cases that result in *unknown* in the C frontend but are valid in EMCCWASM, it seems that a small optimisation is done by default in every case by *emcc*, resulting in a less complex program, to begin with, making the verification task easier and thus not exceeding the timeout. Moreover, as WebAssembly is low-level code, we generate LLVM-IR with simple instructions, and this also contributes to the reason why our frontend performs best.

For the particular case of N°28, it is failing when the LLVM-IR is optimised. This is because the program in the file has two arrays, which in WebAssembly are compiled as data initialisation and further in the LLVM-IR translation produces a sequence of pairs of instructions to initialise the memory present in the WebAssembly module.

```
1 %temp_N = getelementptr inbounds [128 x i8], [128 x i8]* @mem, ↩
    i8 0, i8 %addr
2 store i8 %value, i8* %temp_N
3
4 store i8 %value, i8* getelementptr inbounds ([128 x i8], [128 x↩
     i8]* @mem, i32 0, i32
```

Listing 5.2: Instructions for data initialization

The first line gets the pointer to address specified by `%addr` while the second line stores the value held by `\%value` at the previously retrieved pointer. Skink can process this pair of instructions, but when the LLVM-IR is optimised, the two instructions transform into the instruction in line 4, which is more complex and Skink does not currently handle this case. Adding the capability to process this particular construction of *store* and also *load* will solve this problem. All in all, our frontend performs better when using these benchmark files, but it takes five times longer than the C frontend, on average.

6 Verifying WebAssembly

6.1 Introduction

In this chapter, we cover the application of software verification to WebAssembly sources in order to address possible runtime errors called traps. To achieve this, we turn the possible traps into reachability problems. This is done after the WebAssembly source file is translated into LLVM-IR, where the instructions from the source that would produce a trap are instrumented and modified so they can be transformed into a reachability problem. Although the semantics from the original source program are kept, the resulting instrumented LLVM-IR is not intended to be executed but to be analysed.

The first and most important aspect of this verification is that we are then able to verify if C programs will run without errors in the WebAssembly runtime environment. The main difference is that in Skink's normal verification process, the LLVM-IR is processed as-is, without making any modifications to verify reachability problems on the C/C++ source code. With our approach, we are also able to check reachability problems of C/C++ programs but also the potential problems that may occur running in the WebAssembly VM.

6.2 Traps

Traps in WebAssembly are runtime exceptions that abort the execution [33]. Some instructions unconditionally produce a trap and others may produce a trap depending on the operands involved in the instruction. As WebAssembly does not support exception handling, these traps are reported to the embedding environment, where they can be handled. The embedding environment can be the web browser, a *node.js* environment or native execution such as Lucet[1]. In the case of the browser, for example, WebAssembly traps would have to be handled in the JavaScript code to inform the user of the error. In any case, these traps abort the execution, and there are no mechanisms like *try/catch*, and therefore traps must be prevented, as they could cause serious problems

[1] https://github.com/fastly/lucet

6.2.1 Scenarios that cause traps

The *unreachable* instruction: This instruction produces an unconditional trap. This is covered in section 6.3.

Illegal memory access: When an instruction that reads or writes to memory tries to access out of bounds addresses, a trap is produced. This is not checked in the validation phase as that phase is mainly concerned about types and the address operand is not static but popped from the stack and therefore can be a result from a function call. This is covered in section 6.4.

Operations: Some operations, such as conversions or binary operations, may cause traps. Examples of binary operations producing traps are divisions by 0 and remainder by 0. Conversions can be instructions such as `i32.trunc_f32_s`, that converts a floating point value into an integer representation, based on the IEEE 754-2008 standard *convertToIntegerTowardZero* operation. When the operand does not have an integer representation, such as *NaN* or *Infinity*, then the instruction produces a trap. This is covered in section 6.5

Call stack overflow: As specified in the official documentation, the allocated space for the call stack is unspecified in the module, depends on the embedding environment and it is a source of nondeterminism [34]. The program before being executed has no means of checking the allowed call stack space in order to prevent stack overflows. This is covered in section 6.6

Indirect calls: WebAssembly supports the *call_indirect* instruction, which is an indirect call to a function, based on the stack operand that points to a function reference on a table. A trap will occur if the index provided for the function call exceeds the table size, resulting in an out of bounds table access or if the table is not initialised in the provided index. This is covered in section 6.7

6.3 Detecting unreachable code

One of the WebAssembly instructions that unconditionally produces a trap is *unreachable*. The validation phase of WebAssembly does not check whether the unreachable instructions

are actually unreachable. Instead, this instruction has the role of asserting that the execution should never reach the blocks that begin with the *unreachable* instruction. These instructions can be introduced by the compiler, such as *Emscripten*, as assertions that such instructions should not be reached.

Therefore, if an *unreachable* instruction is reachable then either the compiler is faulty, or in the case of a manually crafted WebAssembly module, the module has bugs introduced by the developer. In the first case, the errors present in the compiler led to the incorrect assumption that a particular block of code is not reachable and in this regard, our approach could detect bugs in compilers. In the second case, although WebAssembly is designed as a compilation target, modules can still be manually crafted, a process which is error-prone and could be exploited by malicious attackers. In either case, there are no checks or validation performed by the WebAssembly validator before the execution that would inform the developer that a specific module will fail and therefore abort execution.

In this section, we present the methods applied to this particular instruction that enable us to statically verify that the *unreachable* instruction is indeed reachable or not and therefore enhancing the validation phase of WebAssembly.

Listing 6.1 shows a simple function that always returns 10.

```
1 (module
2   (func $main (result i32)
3     block
4       i32.const 10
5       return
6     end
7     unreachable ))
```

Listing 6.1: Unreachable instruction

The instruction at line 4 is pushing 10 to the top of the stack. The *return* instruction inside the block will end the function execution and return the value that is on the top of the stack, as the execution will unconditionally enter the block. Therefore, whatever code is after the block, after line 6, should never be executed or the execution would present a runtime error. In order to verify that *unreachable* instructions will never be executed without executing the code, we apply software verification with the following steps. Each occurrence of *unreachable* is translated as two LLVM-IR instructions: 1. `call void @__VERIFIER_error()`. 2. `unreachable`.

In this regard, even though the LLVM-IR code is not to be executed but to be analysed, the original semantics are kept: if the *unreachable* WebAssembly instruction is

reached, then the program fails. The translation to LLVM-IR and instrumentation of the WebAssembly is given in listing A.1 and the corresponding CFG in figure 6.1.

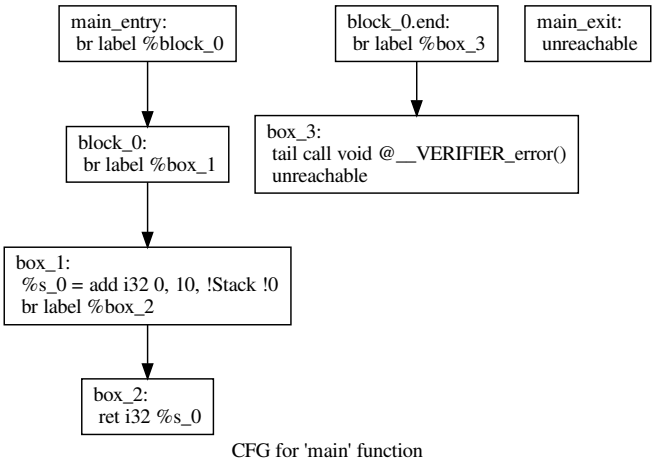

FIGURE 6.1: CFG for the translation of 6.1.

We can see that the call instruction to `__VERIFIER_error` is on block with label *box_3*. As the entry point of the function is the *main_entry* block and the only possible ending being the block *box_2*, there are no possible paths that lead to the execution of instructions in *box_3*. Therefore, the error is not reachable under any circumstances, and we can conclude that this function is correct.

The blocks that are unreachable from *main_entry*, being the main path the one that goes from *main_entry* to *box_2*, are unlinked because of the semantics of the *return* instruction in line 5 in listing 6.1. Therefore, *block_0.end*, *box_3* and *main_exit* represent dead code and the optimizer can get rid of them. Before the code is optimised, we run Skink over this WebAssembly file (listing 6.1) with the *WASM* frontend and it reports that the program is correct, meaning that there are no failure traces. The command line execution of Skink and its resulting log is available at B.1.

When we run the LLVM optimizer, dead code is eliminated, including traces to possible error states that are not reachable, and the function just returns 0, meaning that the verification of this simple scenario by Skink was accurate. In this case, the optimiser could get rid of the dead code, including the error states. However, this is not always possible, especially when entering into a state depends on an external value.

6.3.1 Detecting a reachable *unreachable* WebAssembly instruction

In the following scenario, we can detect if a *switch* statement incorrectly ends up in the default case. The example *switch* in C++ in listing 6.2 could be compiled into WebAssembly and generate a code similar to the one in listing 6.3. WebAssembly does not have a *switch*

expression but has table branching. Table branching specifies a vector of labels to jump to, depending on the value that is on the top of the stack.

```
1 switch(x) {
2   case 0: return 10;
3   case 1: return 11;
4   default:
5     assert(0);
6  }
```

FIGURE 6.2: Switch in C++

In the case of listing 6.3, the table branch depends on the value yielded by `call 0`. The resulting *i32* is a zero-based index that indicates to which of the labels in the labels vector to jump. In the case of line 9 of listing 6.3, `br_table 0 1 2`, if the result points to the first label to jump to is 0, the second is 1 and the last, 2, indicates the default jump.

As the control flow in WebAssembly is structured, jumps to arbitrary parts of the code are not allowed. Instead, jumps are only legal to enclosing blocks in the form of *blocks*, *loops* and *ifs*. The index of the branch targets depends on the nesting depth of the enclosing construction. This means that index 0 targets the innermost control instruction and as it increases it targets those farther out [33]. This is the reason why the *switch* statement is represented with nested blocks in WebAssembly.

```
1  (module
2    (type (;0;) (func (result i32)))
3    (import "env" "___VERIFIER_nondet_int" (func (;0;) (type 0)))
4    (func $main (;0;) (type 0)
5      block                 (;block 0;)
6        block               (;block 1;)
7          block             (;block 2;)
8            call 0
9            br_table 0 1 2
10         end
11         i32.const 10      (; x == 0;)
12         return
13       end
14       i32.const 11        (; x == 1;)
15       return
16     end
17     unreachable           (; default case;)
18   )
19 )
```

FIGURE 6.3: Switch in WebAssembly

Therefore, if the selected index is 0, then the execution will jump to the end of *block 2* as it is the first enclosing structure, pushing 10 to the stack and returning it. If the selected index is 1, then the execution will jump to the second outermost enclosing structure, jumping to the end of *block 1*, pushing 11 to the stack and returning it. In any other scenario, then the execution will jump to the 3rd outermost structure, jumping to the end of *block 0*, executing *unreachable* and therefore producing a trap. The arrows in the image indicate where each value from the vector of labels jumps to.

The optimised CFG of the translated and instrumented LLVM-IR is shown in figure 6.4.

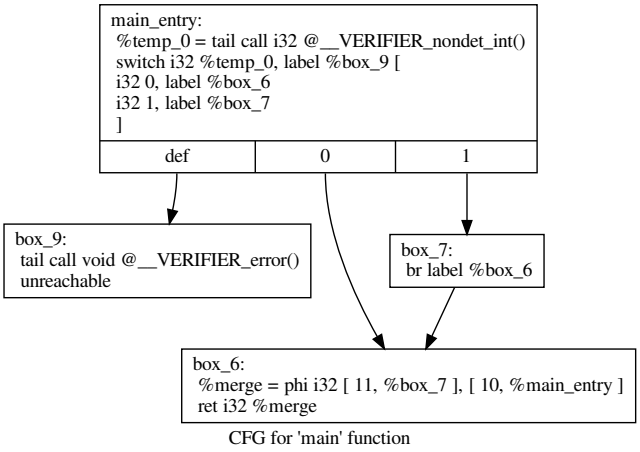

FIGURE 6.4: CFG for reachable *unreachable* in a switch statement.

In the interest of keeping the example concise, the corresponding translation to LLVM-IR is available at appendix A.1 and the complete version of the CFG at appendix C.1. Each WebAssembly `block` is translated into LLVM-IR as a pair of blocks: the starting block and the closing block. All the instructions that are inside each WebAssembly `block` are then enclosed within the translated starting and closing block. This can be appreciated in A.1: for each block opening as `block_0:`,`block_1:`,`block_2:`, there is `block_2.end:`, `block_1.end:`, `block_0.end:`. The `br_table` is then translated into the innermost block, `table_branch_4:`, where the `switch` is generating, which then performs jumps to `block_N.end:`, being N the given value to jump.

The block *main_entry* is where the switch decision is made, and the *def* (default) path is taking place whenever the function called yields a value other than 0 and 1. After running the verification process on this file, Skink indicates that the program is incorrect, meaning that the only present assertion can be violated. The resulting log is available at B.2. A witness is generated by Skink that indicates the conditions under which the error state is reachable. In these simple scenarios, Skink is reporting that if the called function returns 2, then the program will fail. The complete witness is available at A.3.

Although this is a simple scenario, this kind of verification could be applied to ensure that a function's contract is respected over time and that further changes have not affected the possible results. Moreover, the complexity of control flow of a function could increase with time, and the error will not always be visible as this case, and our approach can deal with this situation and in that regard preventing an *unreachable* instruction from aborting the execution. Since these are synthetic and straightforward cases to demonstrate the capabilities of the verification on WebAssembly developed in this research, more complex and realistic cases are covered in the validation section 5.3.

6.3.2 *unreachable* as a wildcard

The validation phase of WebAssembly has its limitations, and in some cases, modules are validated where *unreachable* instructions are indeed reachable. Even though that is not the responsibility of the validator, currently there are no tools to detect these scenarios and prevent a trap from happening.

Listings 6.2 and 6.3 are manually crafted modules. In the example in listing 6.2, the *return* inside the block is returning control to the caller of the current function and returning the expected value. By returning control to the caller at that specific line, the code that follows the block is unreachable by the nature of the *return* instruction.

```
1 (module
2   (func (result i32)
3   block
4     i32.const 10
5     return
6   end
7
8   ))
```

Listing 6.2: Invalid function

```
1 (module
2   (func (result i32)
3   block
4     i32.const 10
5     return
6   end
7   i32.const -1
8   ))
```

Listing 6.3: Valid function

The function in listing 6.2 is declared invalid by the WebAssembly validator, generating the following output B.3. The validator is detecting that the function should return an `i32`, but the stack is empty. This is because instructions inside a *block* cannot access the stack values pushed before entering the *block* and the instructions that follow the block do not have access to values pushed into the stack inside the *block*. In this example the *block* is void, but the *block* could yield a result, making it available to the outer section of the block. Therefore, as the stack is empty at line 6 and the function type indicates that an *i32* must be returned, the instructions in the function fail to satisfy the type constraint.

The problem is that the *return* in line 5 is returning an $i32 = 10$ that is on top of stack pushed by the instruction `i32.const 10`. The function is finishing its execution returning that value, meaning that the rest of the code is unreachable. As a result of this simple scenario, the validation phase does not consider the *return* instructions inside the blocks, and therefore this is a case of a false positive of non-valid WebAssembly module for the validator. One workaround to bypass this validator error is to push a dummy value at the end, as shown in line 7 of listing 6.3. This last instruction may confuse someone reading

the code, and even the translation to machine code will keep that instruction.

According to the WebAssembly specification, the *unreachable* instruction is valid with type $[t_1^*] \rightarrow [t_2^*]$, this means that accepts any types in the stack and produces values of any type, making it a stack-polymorphic instruction [33]. This unreachable "wildcard" is tricking the validator into thinking that the last instruction indeed returns an `i32` and the validation phase succeeds for this program. When the last instruction is executed, the stack would have the required value for the function to be valid, which will never happen.

```
(module
  (func (result i32)
    block
      i32.const 1
      drop
    end
    unreachable ))
```

Listing 6.4: Reachable unreachable

This has the drawback that when function is expected to return a value but the instructions in the function fail to do so, then the validation will succeed if there is an *unreachable* instruction at the end. This is shown in listing 6.4. When the code in listing 6.4 is executed, a runtime error occurs, as expected and error is available in B.1. This means that the use of *unreachable* in these cases could potentially cause a runtime error if false assumptions are in place.

As Hass points out in [2] regarding the instructions that perform unconditional control transfer such as *unreachable*, *return* and others, the control never continues to the instructions that follow them, making that code dead code. Therefore, as the next instructions would never be executed, there are no requirements for the types that the stack requires as input or output and therefore, every type or sequence of types is valid.

This is expectable of any sound compiler that has no errors, but as mentioned before, there is no restriction that WebAssembly could be altered manually and the developer can introduce errors. Moreover, the compiler could have bugs itself as this is the first version of WebAssembly and it is a relatively new technology. Our tool can aid in the validation process in the sense that it can detect if the blocks marked by *unreachable* are truly unreachable and a trap by that condition will never happen.

[1]For this purpose, a validator and executor for simple WebAssembly modules was implemented in https://bitbucket.org/diegoocampohdr/wasm2llvm-testfiles/src/default/wasmExecutor.html

6.4 Detecting illegal memory access

In WebAssembly, memory is modelled as a single array of bytes. In this regard, it does not have information of the bounds of memory allocated to structs or variables inside the code and in this way it is very simple. As the memory address operand is an unsigned int, the only check that we can perform is an out-of-bounds one. Instructions that access the memory must specify the address, which is a zero-based index of the array, and optionally an offset. Depending on the size of the value to load, the number of consecutive bytes that are to be read starting from the given address plus the optional offset. The main operations that deal with memory are *t*`.load` and *t*`.store`, where *t* is the type of the loaded value into the stack when reading and the type of the value to be written, present on the top-of-stack when writing. The condition for the trap is the same for both the *store* and *load*, therefore, we refer to the memory instruction as *meminstr*:

1. Given the memory instruction *t*`.meminstr offset`, where offset is an unsigned 32 bits integer.
2. Let i be the `i32.const` value on top-of-stack, which indicates the memory address.
3. Let $ea = i +$ offset be the effective address.
4. Let N be the bit width of the value to write/read.
5. If $ea + N/8 > mem.length$ then trap.

```
define void @__VERIFIER_memaccess(i32 %addr, i32 %bits) #0  {
    %b = sdiv i32 %bits, 8
    %eab = add i32 %addr, %b
    %memsize = load i32, i32* @__verif_memory_size_bytes
    %if = icmp sgt i32 %eab, %memsize
    br i1 %if, label %box_error, label %exit

  box_error:
    tail call void @__VERIFIER_error()
    unreachable

  exit:
    ret void
}
```

Listing 6.5: Memory bounds check

Given these conditions for the trap, we then instrument the LLVM-IR in order to check if these conditions are met on each memory instruction. The function in listing 6.5 is the

central part of the instrumentation.

This function expects the memory address, which already includes the optional offset and the byte width of the target in bits. This function then compares the intended target address, which also includes the size in bytes (see line 3, where the byte width is added) and compares it against the memory size in bytes.

In order to be able to do so, a global variable has to be defined previously in the LLVM-IR program, `__verif_memory_size_bytes`, that stores the current memory size. This global variable is then modified each time the memory is incremented, with the `memory.grow` instruction, which increments its size by 65536 bytes. Therefore, potential memory size increments are considered in this check.

By calling this function before a memory store or load is executed and in the case that the address would exceed the memory's limits, this function will call the `__VERIFIER_error` function, turning this problem into a reachability problem. Skink can, therefore, detect if this error call is reachable and in such case, a memory access violation will be detected. This function receives two parameters: address (`%adds`) and bit width (`%bits`). The address parameter already includes the offset as that is part of the translation of the memory instruction that calculates the effective address (see line 11 in A.4). The bit width indicates the intended number of bits to be read/written from memory. For the case of truncated memory instructions, such as `i32.store8`, where the `i32` value to be stored is wrapped to 8 bits, the value for bit width that is passed for the (`%bits`) is then 8. A complete example of truncated store in WebAssembly and its translation into LLVM-IR is available at A.11 and A.12. This solution supports the full set of memory instructions.

In the example in the following WebAssembly listing, memory is imported with a minimum and maximum size of 1 page. In order to keep the translation simple and to keep the SMT terms generation to a minimum for these case studies, the page size is reduced to 128 bytes in the translation and instrumentation tool. This value is configurable[2].

In listing 6.6, the initial address provided in the stack is 500, plus an offset of 50, trying to store the `i32.const 10` value. As the memory size is 128 bytes and therefore it

[2] https://bitbucket.org/diegoocampohdr/wasm2llvm/src/default/src/main/scala/org/bitbucket/diegoocampohdr/wasm2llvm/ConfigParams.scala

is modelled into an array of bytes of length 128, the intended target address will cause an illegal memory access trap and this runtime error is not detected by the validation phase.

```
(module
  (type (;0;) (func (result i32)))
  (import "env" "memory" (memory (;0;) 1 1))
  (func $_main (type 0)(result i32)
    i32.const 500          (; addr ;)
    i32.const 10           (; value to store ;)
    i32.store offset=50    (; stores 4 bytes ;)
    i32.const 0
  )
)
```

Listing 6.6: Memory trap example

When running Skink with the WASM frontend, Skink concludes that the program is *INCORRECT* and a witness provided, specifying that the error state is reachable. The witness is available in appendix B.3. The produced LLVM-IR is available in appendix A.4 and the CFG in C.2. This LLVM-IR contains the code corresponding to the translation of the WebAssembly instruction plus the necessary instructions to check the illegal memory access and the global variable declaration to keep track of the current memory size.

Our tool is adding the function presented in listing 6.5 to the end of the generated LLVM-IR file and called before either a memory store or load. As the function is relatively short, the LLVM optimiser is capable of inlining the blocks and instructions into the function that it is being called from and the resulting inlined code is the presented at A.4.

In summary, with this approach, we are able to detect all possible illegal memory accesses, either in a `store` or `load` and taking into account the possible offset provided in the instruction and truncated memory operations.

6.5 Detecting illegal operations

Several binary or unary operations can produce traps when the result is undefined. Examples of undefined results are divisions by 0, remainder by 0 or illegal conversions such as trying to convert *Infinity* into a signed integer representation. In this section we explain the design involved in detecting these scenarios and an implemented case.

In figure 6.5, we model on the left how the translation of an operation is performed and on the right, how a translation with the instrumentation necessary to verify illegal

operations is done. The diagram of the left shows a simple sequence of blocks showing a binary operation, where the two operands (`%s_x` and `%s_x+1`) are calculated and kept in the stack on previous blocks and then the operation is translated.

To instrument the CFG on the left to be able to verify if the conditions of an illegal operation are met while at the same time keeping the semantics of the original instruction, we generate the blocks that are modelled on the right. Instead of a single block for the instruction, block_*N*, we need to generate 3 blocks in total. In block_*N*_entry, we generate the code that is responsible for checking if the operands, or at least one of them, is going to cause a trap given the operation that is being translated. The boolean result (`True` for violation and `False` otherwise) is then stored in the variable `%temp_check`, which is used then used for the condition of the conditional branch to either the error block or performing the operation.

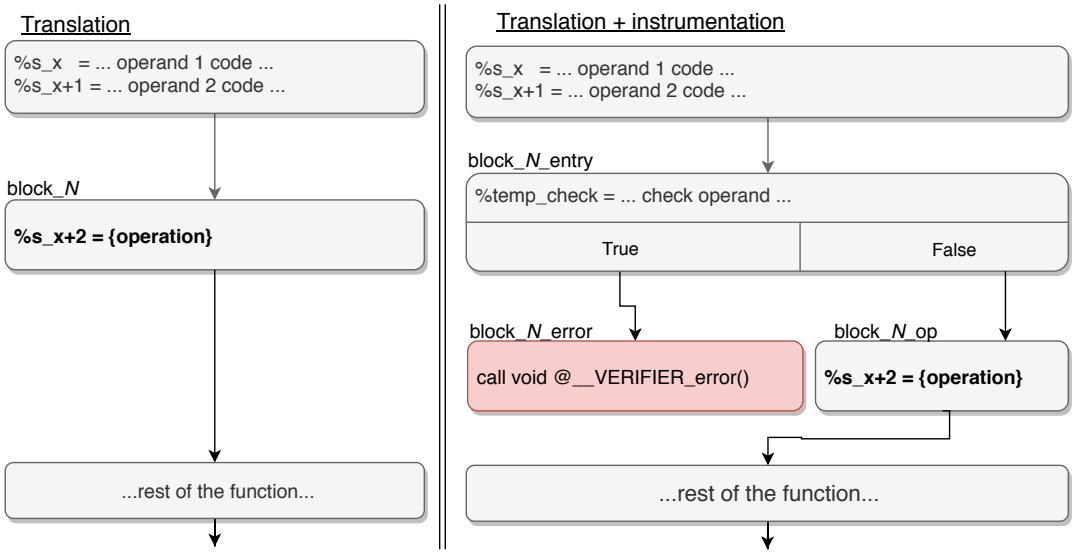

FIGURE 6.5: Verifying illegal operations.

If the operation is illegal, then the conditional branch will jump to block_*N*_error and call `__VERIFIER_error`, reaching an error state. Otherwise, the operation is executed as expected in block_*N*_op the same way as in block_*N*. In this way, the semantics are kept but the trap is modelled into a detectable state, that Skink can figure out if the conditions to reach such state are feasible, whereas in the simple translation (left), an illegal operation is not detectable and will result in a runtime error. When verifying the

example in listing 6.7 with Skink and the WASM frontend, Skink concluded that the program is INCORRECT.

```
(module
  (func (result i32)
    i32.const 10
    i32.const 0
    i32.div_u
  ))
```

Listing 6.7: Illegal operation example

The generated LLVM-IR code, resulting from the execution of Skink on the previous WebAssembly file and presented in A.13, contains the blocks mentioned in the model design. In this case, $N = 2$, and we can appreciate the operand error for the unsigned division: `icmp eq i32 %s_1, 0` in line 11, checking if the divisor is 0; the possible path to the error state in block_2_box_error and the actual execution of the operation in box_2_op, `udiv i32 %s_0, %s_1`, performing the unsigned division.

Wasm2llvm only supports this case, but implementing the checks for the rest of the possible undefined results is trivial. With this approach we can then verify if a WebAssembly program will produce runtime traps originating from illegal operations.

6.6 Detecting stack overflows

Call stack overflows are a source of nondeterminism in WebAssembly [34]. As of the first version of WebAssembly, it is not possible to access the allocated space for the call stack. Furthermore, the first version of WebAssembly does not support tail call, and therefore simple recursive functions could overflow the stack easily. This feature is planned to be added as part of the post MVP features[3]. Therefore, it is not possible to reason about the limits of the call stack space allocated by the embedding environment and to predict if it is going to cause an overflow or not. Emscripten, when executed with the `-s ASSERTIONS=2 -s TOTAL_STACK=1024` options, can generate assertions to detect the stack overflow when the stack space is specified, limiting the stack space to 1024 bytes, in this case. This value is independent of the space that the embedding environment will allocate and should only be used to verify if a specific function will cause this kind of runtime error given a particular stack space size.

When Emscripten generates the HTML+JS for the WebAssembly module, the function in listing A.19 is included in the code and imported into the module. This function is

[3] https://github.com/WebAssembly/proposals/issues/17

invoked from the WebAssembly function once the 1024 stack size limit has been exceeded, aborting the execution and displaying an error message.

In order to keep track of the stack size, Emscripten also generates another function into the WebAssembly module, `__post_instantiate`, where the call stack size is stored into a global variable. This function type is void and does not accept any parameters. This function has to be executed before executing the main function, in order to initialise the mentioned global variable. This is not done inside the module and is the responsibility of the embedding environment, such as a web page, to invoke such function before the execution of the main function. Given the fact that we analyse isolated WebAssembly modules, there is no embedding environment in our translation and verification process and therefore, the call to `__post_instantiate` is included in the translation of the *main* function, after the memory allocation for the local variables.

In the figure 6.6, we show on the left an example WebAssembly function for calculating the factorial of a number, without the generated assertions by Emscripten. On the right, we present the same factorial code but with the added instructions that check if the stack is being overflown. On each call, the global variable that stores the actual stack size is increased by a number of bytes, depending on factors such as the local variables used in the function. At the end of the recursive call, the original size of the call stack is restored.

Our approach is then to rewrite the function invocation to *abortStackOverflow* that Emscripten generates into a call to `__VERIFIER_error`. The latter does not expect parameters while the first expects a number, indicating the amount of bytes that were to be allocated in the stack and failed. Therefore, we translate this call as: dropping the top of the stack and then calling `__VERIFIER_error`. Furthermore, we call `__post_instantiate` in the first part of the *main* function, to make sure the global variables are properly initialized.

6.6.1 First attempt

On our first attempt, we compiled the factorial in C A.20 with Emscripten, then run our tool, *wasm2llvm* and finally optimised the result in order to try to inline the factorial recursion into the main function. This is because Skink does not support multiple functions

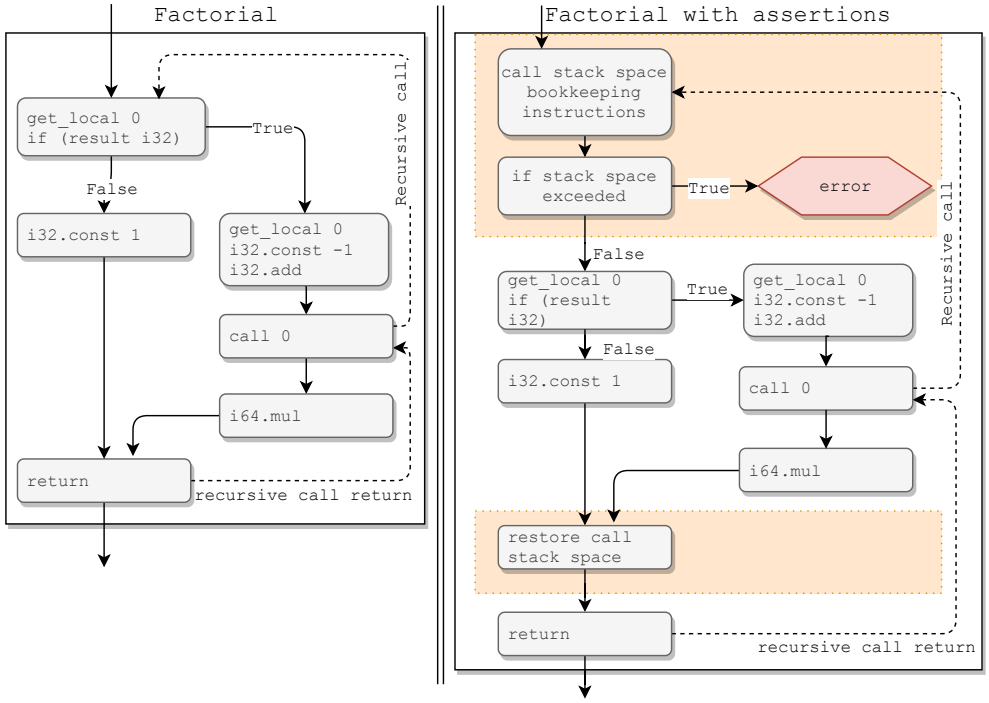

FIGURE 6.6: Left: Factorial without assertions. Right: Factorial algorithm with call stack overflow control instructions that lead to an error state.

and in order to be able to verify this code, it has to be inlined.

In order to keep the assertions generated by Emscripten, no optimisations on the Emscripten level can be used. Otherwise, Emscripten gets rid of all the assertions. In this regard, by using the parameter *-O0*, the resulting WebAssembly code is significantly larger than the optimised version.

The WebAssembly module generated by Emscripten is available A.14 and our translation to LLVM-IR in A.15. The resulting LLVM-IR inlines the `__post_instantiate` function in the first block of *main*, setting the stack size to 1024 bytes. However, the *factorial* function is not inlined by the optimizer and therefore this program is not verifiable by Skink, being this one of the most significant limitations in our verification of WebAssembly: if the LLVM optimiser cannot inline the functions called by *main*, then the *main* function is not verifiable. Even though the optimizer does not inline the functions in this program, our tool is generating the necessary code in LLVM-IR that would make it possible for Skink to verify it if it was able to process function calls, which may be introduced in the future. Moreover, as our tool is not tied to Skink, other LLVM-IR based verification tools that do

not have such limitation will be able to verify stack overflows in WebAssembly using our tool, *wasm2llvm*.

6.6.2 Manual attempt

Unfortunately, there is no possible configuration on Emscripten to compile a C program, perform some optimisations but at the same time keep the generated assertions. In order to reduce the size of the generated program, we manually created a factorial program in WebAssembly, based on the Emscripten output for the stack control when using *-O0* and the compilation of the factorial when using *-O2* flags. We then merged the optimised version of the factorial with the stack control instructions, which store, increment and restore the stack space usage. The WebAssembly code for this file, available in A.16 and its translation, instrumentation and further optimization available in A.17.

When running the optimizer over the translation generated by *wasm2llvm*, the code for the *factorial* function gets inlined into the *main* function, simulating recursion with a loop. In each iteration, 16 bytes are "allocated" to the stack and checked upon the maximum stack size, 1024. When running Skink, it determines that the program is *INCORRECT*, providing the witness A.18, indicating that if we calculate the factorial of 7, the program will fail because of a stack overflow error.

Even though we were able to craft a WebAssembly recursive program that when translated into LLVM-IR the compiler was able to transform the recursion into a loop, that will not always be the case. Therefore, for the cases that the LLVM-IR file, after optimisation, keeps the recursion with function calls, our approach will not be able to verify it, not at least with Skink.

6.7 Detecting illegal indirect calls

Indirect function calls can result in traps when the table is not initialised in the targeted index or if the index is out of bounds. These two cases are slightly different; the table metadata describes its minimum, and maximum size and the original size of the table is based on this minimum. Based on this size is that the out of bounds can occur. A table is

then initialised by the *element* tag, initialising each of the elements of the table and based on this initialisation the first mentioned trap can occur.

The indirect call instruction has the form `call_indirect X`, where X indicates the type of the function to be called, indicating its parameter types and result type. The validation phase of WebAssembly validates that the type specified by X is present in the type definitions of the module, but, as the function to be called is determined dynamically during runtime, the validation is not able to check that the type use specified by X corresponds to the type of the targeted function. This possible type mismatch is another cause of a trap which is discussed later, and an example is provided in A.3.1.

The instruction also expects that a value of type `i32` is present on the top-of-stack, which is the function index for the table. This is asserted in the validation phase, and therefore this cannot be violated during runtime. Currently, WebAssembly only supports the declaration and initialisation of a single table, and therefore no index for the table needs to be provided. The steps that would cause a trap on runtime, given a table *tab*, are then:

1. Let i be the `i32` value popped from the stack.
2. If $i >= tab.length$, trap
3. If $tab[i]$ is not initialized, trap.
4. If 2 and 3 do not trap, then let ft be the function type of the function at $tab[i]$.
5. If $ft \neq X$, then there is a type mismatch trap.

Where $tab.length$ indicates the number of elements in the table and $tab[i]$ is the i^{th} element present on the table, on a zero-based index.

Function pointers, such as the ones present in C, are emulated in WebAssembly by using a table and indirect calls [2]. When function pointers are stored in an array in a C program, *Emscripten* compiles that code into WebAssembly and simulates the array of function pointers with a table and its initialisation, as the example included in A.3. The corresponding WebAssembly file for this C program is available online[4]. For the cases that function pointers are not stored in an array but declared separately, *Emscripten*

[4] https://bitbucket.org/diegoocampohdr/wasm2llvm-testfiles/src/default/wasm/parsing/indirect/indirect-Os.wat

is smart enough to determine the target of the function pointers and generate simple WebAssembly function calls. An example of this case is available at A.6 and its compilation into WebAssembly A.7. Our tool and Skink supports the verification of these cases.

Function pointers can be translated into LLVM-IR in order to be verified, but this presents several challenges. Firstly, the LLVM optimiser does not inline functions that are dynamically referenced. An example of a C file that uses function references and its compilation into LLVM-IR is available at A.3, to which after running the optimiser, the code for the referenced functions is not inlined. Skink does not support LLVM-IR code that is not inlined and therefore, this kind of dynamic referencing is not verifiable by the tool. The reason why this is not supported by Skink yet is that processing a CFG for dynamic referencing and transforming from dynamic code to precomputed static, with dynamic GOTOs, is a complex problem.

Secondly, WebAssembly adds the complexity that tables are heterogeneous, allowing function pointers that reference to functions of different types. In the example compiled LLVM in A.3, it can be appreciated that the function pointers are stored in an array, because they share the same type. This is not possible when translating from WebAssembly into LLVM-IR and therefore a different approach to an array of pointers has to be taken. Due to this complexity and the fact that Skink does not support indirect calls that are not inlined, the implementation of this verification is future work. However, in the next section, we sketch the solution on how to instrument this kind of calls in order to transform these runtime errors into reachability problems.

6.7.1 Modelling illegal indirect calls verification

The diagram on image 6.7 depicts one possible approach to translating WebAssembly tables and indirect calls into LLVM-IR and its further instrumentation. The boxes represent blocks, and the indirect call is translated as a switch statement that jumps to different blocks, depending on the value of the function index. Each possible function call is then translated into a separate block, where the function pointer is retrieved and called. In the case of the block *block_func1*, a type mismatch detection is modelled, discussed later.

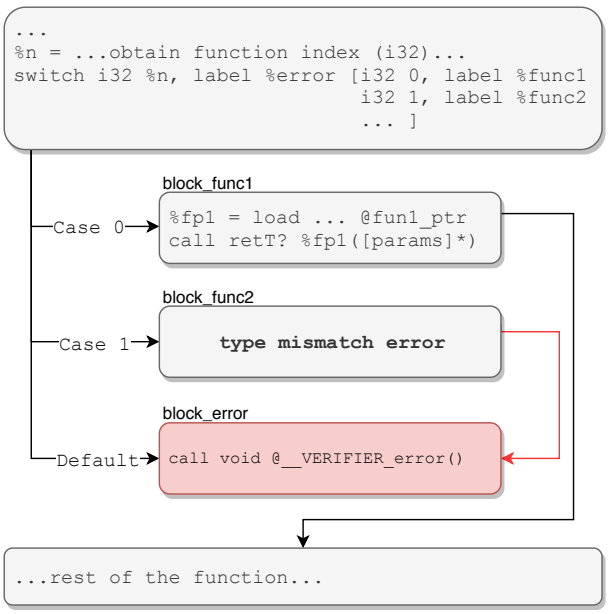

FIGURE 6.7: Targeting the function pointer traps

A final block is modelled as the error state, where we would call `__VERIFIER_error`. A possible translation of WebAssembly tables into LLVM-IR following this model is available at A.8. This allows us to verify that the function index is not out of bounds: if an invalid index is provided, the control flow will jump to this error block, which is the default case for the switch. This would cover uninitialized tables and index out of bounds traps.

In order to detect if the type mismatch is reachable, we analyse the target function type that is part of the indirect call instruction. The target type indicated in indirect calls, such as `call_indirect (type 0)`, is static and therefore we can use that information in our verification instrumentation. When generating each of the blocks that retrieve the function pointers and call them, such as *block_func1*, we know the information of the intended target type (such as `(type 0)`) and the type of the actually selected target function, as such information is stored in the context of our translation, and at this point we know the value of the function index.

These function types are part of the context of the translation and are needed when processing `call $label` instructions from WebAssembly, as function calls in LLVM-IR need to specify the function signature. It is at this point, therefore, that we can be sure if an indirect call results in a type mismatch error and instead of generating the instructions to retrieve the function from its pointer and calling it, we jump to the error block.

With this approach, we would be able to verify all the possible traps produced by the indirect call instruction, but, due to the mentioned limitations, the implementation of this verification is part of the future work.

7 Conclusions

In this research, we developed a general purpose Scala WebAssembly parser *scalaWasm*, a translator from WebAssembly to LLVM-IR, *wasm2llvm*, that instruments the code to perform validations and included two new frontends for Skink: *EMCCWASM* and *WASM*. The first frontend, which handles C/C++ files compiled to WebAssembly using Emscripten, was developed for the validation of our translation. The second frontend, *WASM*, uses the tool developed in this research, *wasm2llvm*, and enables Skink to verify WebAssembly programs and, by extension, any language that compiles to WebAssembly, provided the resulting program is within the limitations of Skink and our translation.

We successfully tested our tool using the *EMCCWASM* frontend with benchmarks from SV-COMP, comparing the results against the C path, scoring better than the C frontend. In this regard, one of the contributions of this research is that our tool could be used not only to verify that the added assertions to the source file remain true but also to verify that a C/C++ source file will run properly under the WebAssembly runtime environment.

After checking that our tool could be used to verify WebAssembly modules to a reasonable degree of confidence, we used the tool to address traps in WebAssembly. We identified all the types of instructions that cause traps, based on the official WebAssembly specification [33] and covered the mechanisms to verify a large subset of the scenarios that produce traps in WebAssembly. We were able to develop the instrumentation code for almost every type of instruction that can lead to a runtime error. This is therefore, the second main contribution of this research: we were able to successfully apply the translation with instrumentation and further verification by Skink to detect incorrect WebAssembly modules, where the assertions were automatically added by our solution to check if a trap would occur. In the cases that traps would occur depending on stack state, we provide a witness that specifies the conditions under which the WebAssembly module would abort. The toolchain we have developed is available online at `bitbucket.org/diegoocampohdr/wasm2llvm`.

Future work needs to be done on our approach to be able to verify larger and more complex WebAssembly modules. In cases such as 5.2, the translation with optimisations resulted in an instruction that was not processable by Skink because the translation of

a specific LLVM-IR instruction into SMT terms is not available yet. With regards to the covered instructions that produce trap, implementing the indirect calls is part of the future work. This particular instruction, as explained in 6.7, is not part of the translation and instrumentation capabilities and is not covered in the run experiments but the approach was designed, due to the Skink limitation that does not support the verification of function calls in LLVM-IR and even though we would have implemented it, we would not be able to carry out the experiments. To add this capability to our approach and using it with Skink, Skink needs to be modified first to be able to process function calls. However, the translation of indirect calls could also be developed to be used with any other verification tool that processes LLVM-IR and the SV-COMP setting.

Furthermore, we currently verify single WebAssembly modules in isolation, without input and that are purely computational, without side effects. By single we mean that we do not support the verification of multiple modules that are linked, share function pointers and can call each other's functions. This means that we do not take into account the values that are instantiated in the embedding environment and then imported into the module, which could be global variables, chunks of the memory byte array, functions from other WebAssembly modules or function tables. Further expansion of this research could involve the verification of WebAssembly programs within an embedding environment, such as web browsers and bringing into the equation the initialised values in JavaScript.

With the current limitations, we believe that web browsers could still benefit from this tool and this provides a solid basis to build on. Web browsers run WebAssembly modules that are untrusted and originates from unknown sources, and even though the code runs sandboxed, they could benefit from our approach by leveraging the validation phase of WebAssembly and preventing runtime errors from occurring and possibly malicious code from running. Furthermore, compilers, such as Emscripten, could benefit from our tool to check that C/C++ files that are correct are compiled into WebAssembly and that they will not produce any of the runtime errors such as illegal memory access or unreachable code actually being run. In this regard, compilers could also simulate the configuration parameters such as the call stack size and memory size limits of a particular target with our approach and test the compilation in a particular environment configuration.

References

[1] F. Cassez, A. M. Sloane, M. Roberts, M. Pigram, P. Suvanpong, and P. G. de Aledo. *Skink: Static analysis of programs in LLVM intermediate representation*. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10206 LNCS, pp. 380–384 (2017). xv, 3, 10, 11, 12

[2] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. *Bringing the web up to speed with WebAssembly*. ACM SIGPLAN Notices **52**(6), 185 (2017). URL http://dl.acm.org/citation.cfm?doid=3140587.3062363. 1, 5, 37, 46

[3] D. Herman, L. Wagner, and A. Zakai. *asm.js*. URL http://asmjs.org/spec/latest/. 1, 5

[4] A. Zakai. *Emscripten: an LLVM-to-JavaScript compiler*. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '11*, p. 301 (ACM Press, New York, New York, USA, 2011). URL http://dl.acm.org/citation.cfm?doid=2048147.2048224. 1

[5] R. Finney and D. Meerzaman. *Chromatic: WebAssembly-Based Cancer Genome Viewer*. Cancer Informatics **17**, 117693511877197 (2018). URL http://journals.sagepub.com/doi/10.1177/1176935118771972. 1, 6

[6] A. M. Sloane, F. Cassez, and S. Buckley. *The sbt-rats parser generator plugin for Scala (tool paper)*. Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala - SCALA 2016 pp. 110–113 (2016). URL http://dl.acm.org/citation.cfm?doid=2998392.3001580. 2

[7] Google. *Native Client*. URL https://developer.chrome.com/native-client/overview. 5

[8] Watt and Conrad. *Mechanising and verifying the WebAssembly specification*. Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs pp. 53–65 (2018). URL https://dl.acm.org/citation.cfm?id=3167082&CFID=851114064&CFTOKEN=83204809. 5, 6

[9] N. Attrapadung. *Efficient Two-level Homomorphic Encryption in Prime-order Bilinear Groups and A Fast Implementation in WebAssembly* pp. 685–697 (2018). 6

[10] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. *Native client: A sandbox for portable, untrusted x86 native code*. Proceedings - IEEE Symposium on Security and Privacy pp. 79–93 (2009). 6

[11] J. King. *A program verifier*. Ph.D. thesis (1969). URL http://search.proquest.com/docview/302239822/citation/9F7E6667508A4F0DPQ/3?accountid=28962. 7

[12] R. W. Floyd. *Assigning Meanings to Programs* pp. 65–81 (1993). URL http://www.springerlink.com/index/10.1007/978-94-011-1793-7_4. 7

[13] D. Isbell and D. Savage. *Mars Climate Orbiter Failure Board Releases Report*. URL https://mars.jpl.nasa.gov/msp98/news/mco991110.html. 7

[14] Trucentis.com. *Software Fail Watch*. URL https://www.tricentis.com/software-fail-watch/. 7

[15] T. Hoare. *The Verifying Compiler : A Grand Challenge for Computing Research*. Computer pp. 1–12 (2003). 7

[16] D. Hutchison and J. C. Mitchell. *Verified Software : Theories , Tools , Experiments* (2005). 8

[17] V. D. Silva, D. Kroening, and G. Weissenbacher. *Keynote Paper Formal Software Verification* **27**(7), 1165 (2008). 8, 9

[18] B. Beckert. *Intelligent Systems and Formal Methods in Software Engineering*. IEEE Intelligent Systems **21**, 71 (2006). URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4042539. 8, 10

[19] P. Cousot and R. Cousot. *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 238–252 (ACM, 1977). 9

[20] E. M. Clarke and E. A. Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic* . 9

[21] J.-P. Queille and J. Sifakis. *Specification and verification of concurrent systems in CESAR*. In *International Symposium on programming*, pp. 337–351 (Springer, 1982). 9

[22] S. Graf and H. Saidi. *Construction of abstract state graphs with PVS* . 10

[23] D. Beyer. *Software Verification with Validation of Results BT - Tools and Algorithms for the Construction and Analysis of Systems*. pp. 331–349 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2017). 10

[24] M. Heizmann. *Traces, Interpolants, and Automata: A New Approach to Automatic Software Verification*. Ph.D. thesis (2015). 11

[25] M. Heizmann, J. Hoenicke, and A. Podelski. *Software Model Checking for People who Love Automata* URL www.springerlink.com/index/10.1007/978-3-642-39799-8_2. 11

[26] A. Farzan, M. Heizmann, J. Hoenicke, Z. Kincaid, and A. Podelski. *Automated program verification*. In *International Conference on Language and Automata Theory and Applications*, pp. 25–46 (Springer, 2015). 11

[27] M. Bradley, F. Cassez, A. Fehnker, T. Given-wilson, R. Huuck, and N. South. *High Performance Static Analysis for Industry*. Electronic Notes in Theoretical Computer Science **289**, 3 (2012). URL http://dx.doi.org/10.1016/j.entcs.2012.11.002. 11, 12

[28] NIST. *SAMATE - Software Assurance Metrics And Tool Evaluation project main page*. URL https://samate.nist.gov/Main_Page.html. 12

[29] M. Heizmann, J. Hoenicke, and A. Podelski. *Refinement of trace abstraction*. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **5673 LNCS**(Sas), 69 (2009). 12

[30] F. Cassez and A. Sloane. *ScalaSMT: Satisfiability Modulo Theory in Scala (tool paper)*. SCALA 2017 - Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, co-located with SPLASH 2017 (2017). 12

[31] Microsoft Corporation. *The Z3 theorem prover* (2017). URL https://github.com/Z3Prover/z3. 12

[32] D. Beyer. *SV-COMP 2015 - 4th International Competition on Software Verification* (2015). URL http://sv-comp.sosy-lab.org/2015/. 24

[33] W. C. Group. *WebAssembly Specification* (2017). 30, 34, 37, 49

[34] *WebAssembly Semantics*. URL https://webassembly.org/docs/semantics/. 31, 42

A Code listings

A.1 Unreachable

```
define i32 @main() #0  {
  main_entry:
    br label %block_0

  block_0:
    br label %box_1
    %s_0 = add i32 0, 10
    ret i32 %s_0

  block_0.end:
    br label %box_3

  box_3:
    tail call void @__VERIFIER_error()
    unreachable

  main_exit:
    unreachable
}
declare void @__VERIFIER_error()
```

Listing A.1: Intermediate representation of the *unreachable* instruction

```
define i32 @main() #0  {

  main_entry:
    br label %block_0

  block_0:
    br label %block_1

  block_1:
    br label %block_2

  block_2:
    br label %box_3

  box_3:
    %temp_0 = call i32 @__VERIFIER_nondet_int()
    br label %tableBranch_4

  tableBranch_4:
    switch i32 %temp_0, label %block_0.end [
      i32 0, label %block_2.end
      i32 1, label %block_1.end
    ]

  block_2.end:
    br label %box_5

  box_5:
    %s_0 = add i32 0, 10
    br label %box_6

  box_6:
    ret i32 %s_0

  block_1.end:
    br label %box_7

  box_7:
```

```
39      %s_1 = add i32 0, 11
40      br label %box_8
41
42    box_8:
43      ret i32 %s_1
44
45    block_0.end:
46      br label %box_9
47
48    box_9:
49      tail call void @__VERIFIER_error()
50      unreachable
51
52    main_exit:
53      unreachable
54
55  }
56  declare i32 @__VERIFIER_nondet_int()
57  declare void @__VERIFIER_error()
```

Listing A.2: Translation and instrumentation of switch statement with default as unreachable.

A.2 Failure witness in switch

```
1  <graphml xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
2           xmlns="http://graphml.graphdrawing.org/xmlns"
3  >...
4    <graph edgedefault="directed">
5      <data key="witness-type">violation_witness</data>
6      <data key="sourcecodelang">C</data>
7      <data key="producer">skink</data>
8      <data key="specification">CHECK( init(main()), LTL(G ! call(↩
           __VERIFIER_error())) )</data>
9      <data key="programfile">/home/diego/uni/repos/thesisOverleafRepo/5↩
           c6d2cebd1ba83158575460d/code/unreachable/reachable/reachableUnreachable↩
           -switch.wat</data>
10     <data key="programhash">9d2543a231fc765de8b227d9a7ecf81f5660470b</data>
11     <data key="memorymodel">simple</data>
12     <data key="architecture">32bit</data>
13     <node id="N0">
14       <data key="entry">true</data>
15     </node>
16     <edge id="E0" source="N0" target="N1">
17       <data key="assumption">\result == 2;</data>
18       <data key="assumption.note">hex: 2</data>
19       <data key="assumption.scope">main</data>
20       <data key="assumption.resultfunction">__VERIFIER_nondet_int</data>
21     </edge>
22     <node id="N1">
23       <data key="violation">true</data>
24     </node>
25   </graph>
26 </graphml>
```

Listing A.3: Switch failure witness [1].

[1] Listing A.3 at lines 40-45 shows the most important part of the witness that indicates the values that make the trace to the error state feasible and where it is coming from. This witness is specifying that in the context (*assumption.scope*) of the *main* function, if the call to the function *__VERIFIER_nondet_int* (*assumption.resultfunction*) results in 2 (*assumption*), then an error state is reachable.

A.3 Function pointers

```c
unsigned int __VERIFIER_nondet_uint();
int add2(int a) {
    return a+2;
}
int multiply(int a){
    return a*2;
}
int main() {
    unsigned int n = __VERIFIER_nondet_uint();
    int (*fun_ptr_arr[])(int) = {add2, multiply};
    return (*fun_ptr_arr[n])(n);
}
```

Listing A.4: C file with function pointers stored in array.

```
@main.fun_ptr_arr = private unnamed_addr constant [2 x i32 (i32)*] [i32 (i32)* @add2, i32 (i32)* @multiply], align 16
define i32 @add2(i32) #0 {
  %2 = add nsw i32 %0, 2
  ret i32 %2
}
define i32 @multiply(i32) #0 {
  %2 = shl nsw i32 %0, 1
  ret i32 %2
}
define i32 @main() local_unnamed_addr #1 {
  %1 = tail call i32 (...) @__VERIFIER_nondet_uint() #3
  %2 = zext i32 %1 to i64
  %3 = getelementptr inbounds [2 x i32 (i32)*], [2 x i32 (i32)*]* @main.fun_ptr_arr, i64 0, i64 %2
  %4 = load i32 (i32)*, i32 (i32)** %3, align 8, !tbaa !2
  %5 = tail call i32 %4(i32 %1) #3
  ret i32 %5
}

declare i32 @__VERIFIER_nondet_uint(...) local_unnamed_addr #2
```

Listing A.5: Function pointers in LLVM, compiled from A.4.

```c
        int __VERIFIER_nondet_int();
        int add2(int a){
        return a+2;
        }
        int mult(int a) {
        return a*2;
        }
        int (*fun_ptr)(int) = add2;
        int (*fun_ptr2)(int) = mult;
        int main() {
        int val = __VERIFIER_nondet_int();
        return fun_ptr(val) + fun_ptr2(val);
        }
```

Listing A.6: C file with function pointers without using arrays.

```
(module
  (type (;0;) (func (result i32)))
  (type (;1;) (func (param i32) (result i32)))
  (type (;2;) (func))
  (import "env" "___VERIFIER_nondet_int" (func $___VERIFIER_nondet_int (type 0)))
  (func $_add2 (type 1) (param i32) (result i32)
    (local i32)
    local.get 0
```

```
    i32.const 2
    i32.add
    local.set 1
    local.get 1)
  (func $_mult (type 1) (param i32) (result i32)
    (local i32)
    local.get 0
    i32.const 1
    i32.shl
    local.set 1
    local.get 1)
  (func $_main (type 0) (result i32)
    (local i32 i32)
    call $___VERIFIER_nondet_int
    local.set 0
    local.get 0
    call $_add2
    local.set 1
    local.get 0
    call $_mult
    local.set 0
    local.get 0
    local.get 1
    i32.add
    local.set 0
    local.get 0)
)
```

Listing A.7: Simple indirect calls in WebAssembly

```
@fp1 = local_unnamed_addr global i32 (i32)* @add2, align 8
@fp2 = local_unnamed_addr global i32 (i32)* @mult, align 8
define i32 @add2(i32) #0 {
  %2 = add nsw i32 %0, 2
  ret i32 %2
}
define i32 @mult(i32) #0 {
  %2 = shl nsw i32 %0, 1
  ret i32 %2
}
define i32 @main() local_unnamed_addr #1 {
main_entry:
  %n = tail call i32 (...) @__VERIFIER_nondet_int() #3
  switch i32 %n, label %error [
    i32 0, label %f1
    i32 1, label %f2
  ]

f1:
  %0 = load i32 (i32)*, i32 (i32)** @fp1, align 8, !tbaa !2
  %1 = tail call i32 %0(i32 0) #3
  br label %main_end

f2:
  %2 = load i32 (i32)*, i32 (i32)** @fp2, align 8, !tbaa !2
  %3 = tail call i32 %2(i32 1) #3
  br label %main_end

error:
  tail call void @__VERIFIER_error() #4
  unreachable

main_end:
  %main_ret = phi i32 [ %1, %f1 ], [ %3, %f2 ]
  ret i32 %main_ret
}
declare i32 @__VERIFIER_nondet_int(...) local_unnamed_addr #2
declare void @__VERIFIER_error() local_unnamed_addr
```

Listing A.8: Indirect calls implementation in LLVM-IR

A.3.1 Type mismatch error

```
(module
  (table 1 10 anyfunc)
  (type (func (result i32)))
  (type (func (param i32) (result i32)))
  (func $_main (type 0) (result i32)
    i32.const 0
    call_indirect (type 0)
  )
  (func $_add2 (type 1) (param i32) (result i32)
    (local i32)
    local.get 0
    i32.const 2
    i32.add
  )
  (elem (i32.const 0) $_add2)
  (export "_main" (func 0))
)
```

Listing A.9: Type mismatch

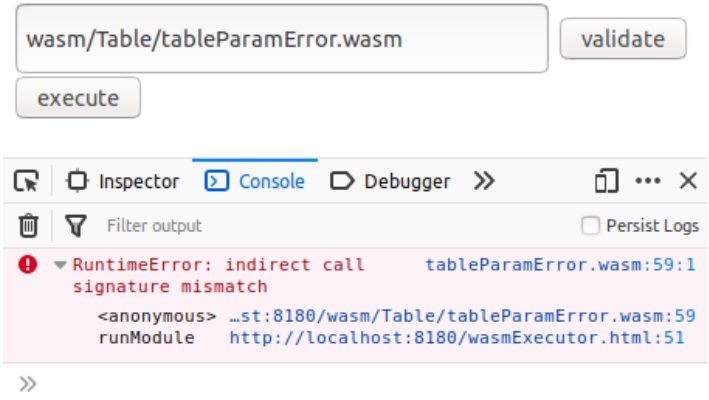

FIGURE A.1: Mismatch runtime error

A.4 Illegal memory access

```
@mem = global [128 x i8] zeroinitializer, align 1
@__verif_memory_size_bytes = global i32 128

define i32 @main() {
main_entry:
  br label %box_0

box_0:
  %s_0 = add i32 0, 500
```

```
  %s_1 = add i32 0, 10
  %temp_2 = add i32 50, %s_0
  %temp_0 = getelementptr inbounds [128 x i8], [128 x i8]* @mem, i32 0, i32 %temp_2
  %temp_1 = bitcast i8* %temp_0 to i32*
  %memsize.i = load i32, i32* @__verif_memory_size_bytes
  %if.i = icmp sgt i32 554, %memsize.i
  br i1 %if.i, label %box_error.i, label %__VERIFIER_memaccess.exit

box_error.i:
  call void @__VERIFIER_error()
  unreachable

__VERIFIER_memaccess.exit:
  store i32 %s_1, i32* %temp_1
  %s_2 = add i32 0, 0
  br label %main_exit

main_exit:
  ret i32 %s_2
}
declare void @__VERIFIER_error()
```

Listing A.10: Memory trap example

```
(module
  (type (;0;) (func (result i32)))
  (import "env" "memory" (memory (;0;) 1 1))
  (func $_main (type 0)(result i32)
    i32.const 500          (; addr ;)
    i32.const 10           (; value to store ;)
    i32.store8 offset=50   (; stores 4 bytes ;)
    i32.const 0
  )
)
```

Listing A.11: Truncated memory trap example

```
@mem = global [128 x i8] zeroinitializer, align 1
@__verif_memory_size_bytes = global i32 128

define i32 @main() {
main_entry:
  br label %box_0

box_0:                                            ; preds = %main_entry
  %s_0 = add i32 0, 500, !Stack !0
  %s_1 = add i32 0, 10, !Stack !1
  %temp_0 = trunc i32 %s_1 to i8
  %temp_3 = add i32 50, %s_0, !Stack !2
  %temp_1 = getelementptr inbounds [128 x i8], [128 x i8]* @mem, i32 0, i32 %temp_3, !Stack !2
  %temp_2 = bitcast i8* %temp_1 to i8*, !Stack !2
  %memsize.i = load i32, i32* @__verif_memory_size_bytes
  %if.i = icmp sgt i32 551, %memsize.i
  br i1 %if.i, label %box_error.i, label %__VERIFIER_memaccess.exit

box_error.i:                                      ; preds = %box_0
  call void @__VERIFIER_error()
  unreachable

__VERIFIER_memaccess.exit:                        ; preds = %box_0
  store i8 %temp_0, i8* %temp_2, !Stack !2
  %s_2 = add i32 0, 0, !Stack !3
  br label %main_exit

```

```
main_exit:                                        ; preds = ↩
    %__VERIFIER_memaccess.exit
  ret i32 %s_2
}

declare void @__VERIFIER_error()
```

Listing A.12: Truncated memory trap example translation

A.5 Illegal operations

```
define i32 @main() {
main_entry:
  br label %box_0

box_0:
  %s_0 = add i32 0, 10
  %s_1 = add i32 0, 0
  br label %box_2

box_2:
  %temp_0 = icmp eq i32 %s_1, 0
  br i1 %temp_0, label %box_2_box_error, label %box_2_op

box_2_box_error:
  tail call void @__VERIFIER_error()
  unreachable

box_2_op:
  %s_2 = udiv i32 %s_0, %s_1
  br label %main_exit

main_exit:
  ret i32 %s_2
}

declare void @__VERIFIER_error()
```

Listing A.13: Generated code for the illegal operation example

A.6 Stack overflows

```
(module
  (type (;0;) (func (param i32 i32)))
  (type (;1;) (func (param i32)))
  (type (;2;) (func (result i32)))
  (type (;3;) (func (param i32) (result i32)))
  (type (;4;) (func))
  (import "env" "abortStackOverflow" (func $abortStackOverflow (type 1)))
  (import "env" "___VERIFIER_nondet_int" (func $___VERIFIER_nondet_int (type 2)↩
      ))
  (import "env" "__memory_base" (global (;0;) i32))
  (import "env" "__table_base" (global (;1;) i32))
  (import "env" "tempDoublePtr" (global (;2;) i32))
  (import "env" "DYNAMICTOP_PTR" (global (;3;) i32))
```

```
  (import "global" "NaN" (global (;4;) f64))
  (import "global" "Infinity" (global (;5;) f64))
  (import "env" "memory" (memory (;0;) 1))
  (import "env" "table" (table (;0;) 0 funcref))
  (func $_factorial (type 3) (param i32) (result i32)
    (local i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32 i32)
    global.get 8
    local.set 12
    global.get 8
    i32.const 16
    i32.add
    global.set 8
    global.get 8
    global.get 9
    i32.ge_s
    if  ;; label = @1
      i32.const 16
      call $abortStackOverflow
    end
    local.get 0
    local.set 3
    local.get 3
    local.set 4
    local.get 4
    i32.const 0
    i32.eq
    local.set 5
    local.get 5
    if  ;; label = @1
      i32.const 1
      local.set 1
      local.get 1
      local.set 2
      local.get 12
      global.set 8
      local.get 2
      return
    else
      local.get 3
      local.set 6
      local.get 6
      i32.const 1
      i32.sub
      local.set 7
      local.get 7
      call $_factorial
      local.set 8
      local.get 3
      local.set 9
      local.get 8
      local.get 9
      i32.mul
      local.set 10
      local.get 10
      local.set 1
      local.get 1
      local.set 2
      local.get 12
      global.set 8
      local.get 2
      return
    end
    unreachable)
  (func $_main (type 2) (result i32)
    (local i32 i32 i32 i32 i32)
    global.get 8
    local.set 4
    global.get 8
```

```
    i32.const 16
    i32.add
    global.set 8
    global.get 8
    global.get 9
    i32.ge_s
    if  ;; label = @1
      i32.const 16
      call $abortStackOverflow
    end
    i32.const 0
    local.set 0
    call $___VERIFIER_nondet_int
    local.set 1
    local.get 1
    call $_factorial
    local.set 2
    local.get 4
    global.set 8
    local.get 2
    return)
  (func $__post_instantiate (type 4)
    global.get 0
    i32.const 0
    i32.add
    global.set 8
    global.get 8
    i32.const 1024
    i32.add
    global.set 9)
  (global (;6;) (mut i32) (global.get 2))
  (global (;7;) (mut i32) (global.get 3))
  (global (;8;) (mut i32) (i32.const 0))
  (global (;9;) (mut i32) (i32.const 0))
  (global (;10;) (mut i32) (i32.const 0))
  (global (;11;) (mut i32) (i32.const 0))
  (global (;12;) (mut i32) (i32.const 0))
  (global (;13;) (mut i32) (i32.const 0))
  (global (;14;) (mut i32) (i32.const 0))
  (global (;15;) (mut i32) (i32.const 0))
  (global (;16;) (mut i32) (i32.const 0))
  (global (;17;) (mut f64) (f64.const 0x0p+0 (;=0;)))
  (global (;18;) (mut f64) (global.get 4))
  (global (;19;) (mut f64) (global.get 5))
  (global (;20;) (mut f32) (f32.const 0x0p+0 (;=0;)))
  (global (;21;) (mut f32) (f32.const 0x0p+0 (;=0;)))
  (export "__post_instantiate" (func $__post_instantiate))
  (export "_main" (func $_main)))
```

Listing A.14: Factorial in WebAssembly compiled with Emscripten.

```
; ModuleID = 'Factorial-O0-wasm2llvm.ll'
source_filename = "Factorial-O0-wasm2llvm.ll"
target datalayout = "e-p:32:32-m:e-i64:64-f80:128-n8:16:32:64-S128"

@Global_0 = local_unnamed_addr global i32 0
@Global_1 = local_unnamed_addr global i32 0
@Global_2 = local_unnamed_addr global i32 0
@Global_3 = local_unnamed_addr global i32 0
@Global_4 = local_unnamed_addr global double 0x7FF8000000000000
@Global_5 = local_unnamed_addr global double 0x7FF0000000000000
@Global_6 = local_unnamed_addr global i32 0
@Global_7 = local_unnamed_addr global i32 0
```

```
@Global_8 = local_unnamed_addr global i32 0
@Global_9 = local_unnamed_addr global i32 0
@Global_10 = local_unnamed_addr global i32 0
@Global_11 = local_unnamed_addr global i32 0
@Global_12 = local_unnamed_addr global i32 0
@Global_13 = local_unnamed_addr global i32 0
@Global_14 = local_unnamed_addr global i32 0
@Global_15 = local_unnamed_addr global i32 0
@Global_16 = local_unnamed_addr global i32 0
@Global_17 = local_unnamed_addr global double 0.000000e+00
@Global_18 = local_unnamed_addr global double 0x7FF8000000000000
@Global_19 = local_unnamed_addr global double 0x7FF0000000000000
@Global_20 = local_unnamed_addr global float 0.000000e+00
@Global_21 = local_unnamed_addr global float 0.000000e+00
@mem = local_unnamed_addr global [128 x i8] zeroinitializer, align 1
@__verif_memory_size = local_unnamed_addr global i32 1
@__verif_memory_size_bytes = local_unnamed_addr global i32 128

define i32 @_factorial(i32 %param_0) local_unnamed_addr {
_factorial_entry:
  %s_0 = load i32, i32* @Global_8, align 4, !Stack !0
  %s_3 = add i32 %s_0, 16, !Stack !1
  store i32 %s_3, i32* @Global_8, align 4, !Stack !2
  %s_5 = load i32, i32* @Global_9, align 4, !Stack !3
  %temp_1 = icmp slt i32 %s_3, %s_5, !Stack !4
  br i1 %temp_1, label %box_12, label %box_10

box_10:                                           ; preds = %_factorial_entry
  tail call void @__VERIFIER_error()
  br label %box_12

box_12:                                           ; preds = %_factorial_entry, ↩
    %box_10
  %temp_3 = icmp eq i32 %param_0, 0, !Stack !5
  br i1 %temp_3, label %box_29, label %box_30

box_29:                                           ; preds = %box_12, %box_30
  %merge = phi i32 [ %s_26, %box_30 ], [ 1, %box_12 ]
  store i32 %s_0, i32* @Global_8, align 4
  ret i32 %merge

box_30:                                           ; preds = %box_12
  %s_21 = add i32 %param_0, -1
  %temp_6 = tail call i32 @_factorial(i32 %s_21)
  %s_26 = mul i32 %temp_6, %param_0, !Stack !6
  br label %box_29
}

define i32 @main() local_unnamed_addr {
main_entry:
  %s_43.i = load i32, i32* @Global_0, align 4, !Stack !7
  %s_48.i = add i32 %s_43.i, 1024, !Stack !8
  store i32 %s_48.i, i32* @Global_9, align 4, !Stack !2
  %s_34 = add i32 %s_43.i, 16, !Stack !9
  store i32 %s_34, i32* @Global_8, align 4, !Stack !2
  %temp_9 = icmp slt i32 %s_34, %s_48.i, !Stack !10
  br i1 %temp_9, label %box_66, label %box_64

box_64:                                           ; preds = %main_entry
  tail call void @__VERIFIER_error()
  br label %box_66

box_66:                                           ; preds = %main_entry, ↩
    %box_64
  %temp_11 = tail call i32 @__VERIFIER_nondet_int()
  %temp_12 = tail call i32 @_factorial(i32 %temp_11)
  store i32 %s_43.i, i32* @Global_8, align 4, !Stack !2
  ret i32 %temp_12
```

```
}

; Function Attrs: norecurse nounwind
define void @__post_instantiate() local_unnamed_addr #0 {
__post_instantiate_entry:
  %s_43 = load i32, i32* @Global_0, align 4, !Stack !7
  store i32 %s_43, i32* @Global_8, align 4, !Stack !2
  %s_48 = add i32 %s_43, 1024, !Stack !8
  store i32 %s_48, i32* @Global_9, align 4, !Stack !2
  ret void
}

declare void @__VERIFIER_error() local_unnamed_addr

declare i32 @__VERIFIER_nondet_int() local_unnamed_addr

attributes #0 = { norecurse nounwind }

!0 = !{!"Stack((s_0, W32Int()))"}
!1 = !{!"Stack((s_3, W32Int()))"}
!2 = !{!"Stack()"}
!3 = !{!"Stack((s_5, W32Int()), (s_4, W32Int()))"}
!4 = !{!"Stack((s_6, W32Int()))"}
!5 = !{!"Stack((s_12, W32Int()))"}
!6 = !{!"Stack((s_26, W32Int()))"}
!7 = !{!"Stack((s_43, W32Int()))"}
!8 = !{!"Stack((s_48, W32Int()))"}
!9 = !{!"Stack((s_34, W32Int()))"}
!10 = !{!"Stack((s_37, W32Int()))"}
```

Listing A.15: Transtaltion into LLVM-IR of A.14

```
(module
  (type (;0;) (func (param i32) (result i32)))
  (type (;1;) (func))
  (type (;2;) (func (result i32)))
  (import "env" "__memory_base" (global (;0;) i32))
  (import "env" "___VERIFIER_error" (func (;0;) (type 1)))
  (import "env" "___VERIFIER_nondet_int" (func (;1;) (type 2)))
  (func $_factorial (type 0) (param i32) (result i32)
    (local i32 i32)
    (;-- call stack usage control --;)
    get_global 1
    set_local 2       (;store current state of used stack;)
    get_global 1
    i32.const 16
    i32.add
    set_global 1
    get_global 1
    get_global 2
    i32.ge_s
    if  ;; label = @1
      call $___VERIFIER_error
    end
    (;---------------------- -;)
    get_local 0
    if (result i32)  ;; label = @1
      get_local 0
      i32.const -1
      i32.add
      call $_factorial
      get_local 0
      i32.mul
      return
    else
      i32.const 1
    end
    get_local 2
```

```
    set_global 1 (;restore the used stack size;)
  )
  (func $_main (type 2) (result i32)
    (local i32)
    call 1
    call 2
  )
  (global (;1 -Used stack space;) (mut i32) (i32.const 0))
  (global (;2 -Max stack space ;) (mut i32) (i32.const 1024))
  (export "_main"       (func 3))
)
```

Listing A.16: Factorial in WebAssembly manually crafted.

```
@Global_0 = local_unnamed_addr global i32 0
@Global_1 = local_unnamed_addr global i32 0
@Global_2 = local_unnamed_addr global i32 100

define i32 @main() local_unnamed_addr {
_main_entry:
  br label %tailrecurse.i

tailrecurse.i:
  %accumulator.tr.i = phi i32 [ 1, %_main_entry ], [ %s_14.i, %box_11.i ]
  %param_0.tr.i = phi i32 [ 2, %_main_entry ], [ %s_10.i, %box_11.i ]
  %s_0.i = load i32, i32* @Global_1, align 4, !Stack !0
  %s_2.i = add i32 %s_0.i, 16, !Stack !1
  store i32 %s_2.i, i32* @Global_1, align 4, !Stack !2
  %s_4.i = load i32, i32* @Global_2, align 4, !Stack !3
  %s_5.i = icmp slt i32 %s_2.i, %s_4.i, !Stack !2
  br i1 %s_5.i, label %box_9.i, label %box_8.i

box_8.i:
  tail call void @__VERIFIER_error()
  br label %box_9.i

box_9.i:
  %temp_1.i = icmp eq i32 %param_0.tr.i, 0, !Stack !2
  br i1 %temp_1.i, label %_factorial.exit, label %box_11.i

box_11.i:
  %s_10.i = add i32 %param_0.tr.i, -1, !Stack !4
  %s_14.i = mul i32 %param_0.tr.i, %accumulator.tr.i, !Stack !5
  br label %tailrecurse.i

_factorial.exit:
  ret i32 %accumulator.tr.i
}
declare void @__VERIFIER_error() local_unnamed_addr
```

Listing A.17: Transtaltion into LLVM-IR of A.16.

```
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   ...
   <graph edgedefault="directed">
      <data key="witness-type">violation_witness</data>
      <data key="sourcecodelang">C</data>
      <data key="producer">skink</data>
      <data key="specification">CHECK( init(main()), LTL(G ! call(__VERIFIER_error())) )</data>
```

```
8       <data key="programfile">/home/diego/uni/repos/wasm2llvm-testfiles/wasm/↩
            Factorial/CallStackOverflow/SimpleFactorial/FactorialAssert-wasm2llvm↩
            -opt2.ll</data>
9       <data key="programhash">6a20b5759b1986a89fd2081796e9860bcf3a7dc9</data>
10      <data key="memorymodel">simple</data>
11      <data key="architecture">32bit</data>
12      <node id="N0">
13         <data key="entry">true</data>
14      </node>
15      <edge id="E0" source="N0" target="N1">
16         <data key="assumption">\result == 7;</data>
17         <data key="assumption.note">hex: 7</data>
18         <data key="assumption.scope">main</data>
19         <data key="assumption.resultfunction">__VERIFIER_nondet_int</data>
20      </edge>
21      <node id="N1" />
22      <edge id="E1" source="N1" target="N2" />
23      <node id="N2">
24         <data key="violation">true</data>
25      </node>
26   </graph>
27 </graphml>
```

Listing A.18: Manual factorial failure witness.

```
1       function abortStackOverflow(allocSize) {
2       abort('Stack overflow! Attempted to allocate ...');
3       }
```

Listing A.19: JavaScript function that captures stack overflows in WebAssembly.

```
1  extern int __VERIFIER_nondet_int();
2
3  int factorial(int n) {
4    if (n == 0){
5      return 1;
6    }else{
7      return factorial(n-1) * n;
8    }
9  }
10
11 int main(){
12   return factorial(__VERIFIER_nondet_int());
13 }
```

Listing A.20: Factorial in C

B | Skink logs and other output

B.1 Unreachable *unreachable*

Skink log from the execution that verified WebAssembly code present in listing 6.1.

```
/skink$ skink-frontend wasm unreachable.wat
skink.verifier.TraceRefinement - main has no failure traces
skink.verifier.Verifier - verify: CORRECT
skink.verifier.Verifier - verify: main is correct
```

Listing B.1: Running Skink over unreachable.wat with the WASM frontend.

B.2 Reachable *unreachable*

Skink log from the execution that verified WebAssembly code present in listing 6.3.

```
/skink$ skink-frontend wasm reachableUnreachable-switch.wat
skink.verifier.Verifier - verify: INCORRECT
skink.verifier.Verifier - verify: main is incorrect
```

Listing B.2: Running Skink on the switch with the WASM frontend.

```
unreachable.wat:6:5: error: type mismatch in implicit return, expected [i32] but got []
end
^^^
```

Listing B.3: WebAssembly validator: Invalid function

B.3 Skink on detecting illegal memory access

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   ...key declarations...
   <graph edgedefault="directed">
      <data key="witness-type">violation_witness</data>
      <data key="sourcecodelang">C</data>
      <data key="producer">skink</data>
      <data key="specification">CHECK( init(main()), LTL(G ! call(__VERIFIER_error())) )</data>
      <data key="programfile">.../MemoryTrap.wat</data>
      <data key="programhash">9268a5b2fb41224e430495e453633ff1f3b3922e</data>
      <data key="memorymodel">simple</data>
      <data key="architecture">32bit</data>
      <node id="N0">
         <data key="entry">true</data>
      </node>
      <edge id="E0" source="N0" target="N1" />
```

```
17        <node id="N1">
18           <data key="violation">true</data>
19        </node>
20     </graph>
21 </graphml>
```

Listing B.4: Memory access verification witness.

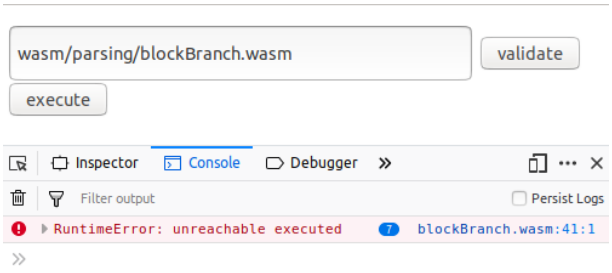

FIGURE B.1: Reaching an *unreachable* instruction using the WebAssembly API [1].

C | Diagrams

C.1 CFG diagram for the *switch* instruction in section 6.3.1

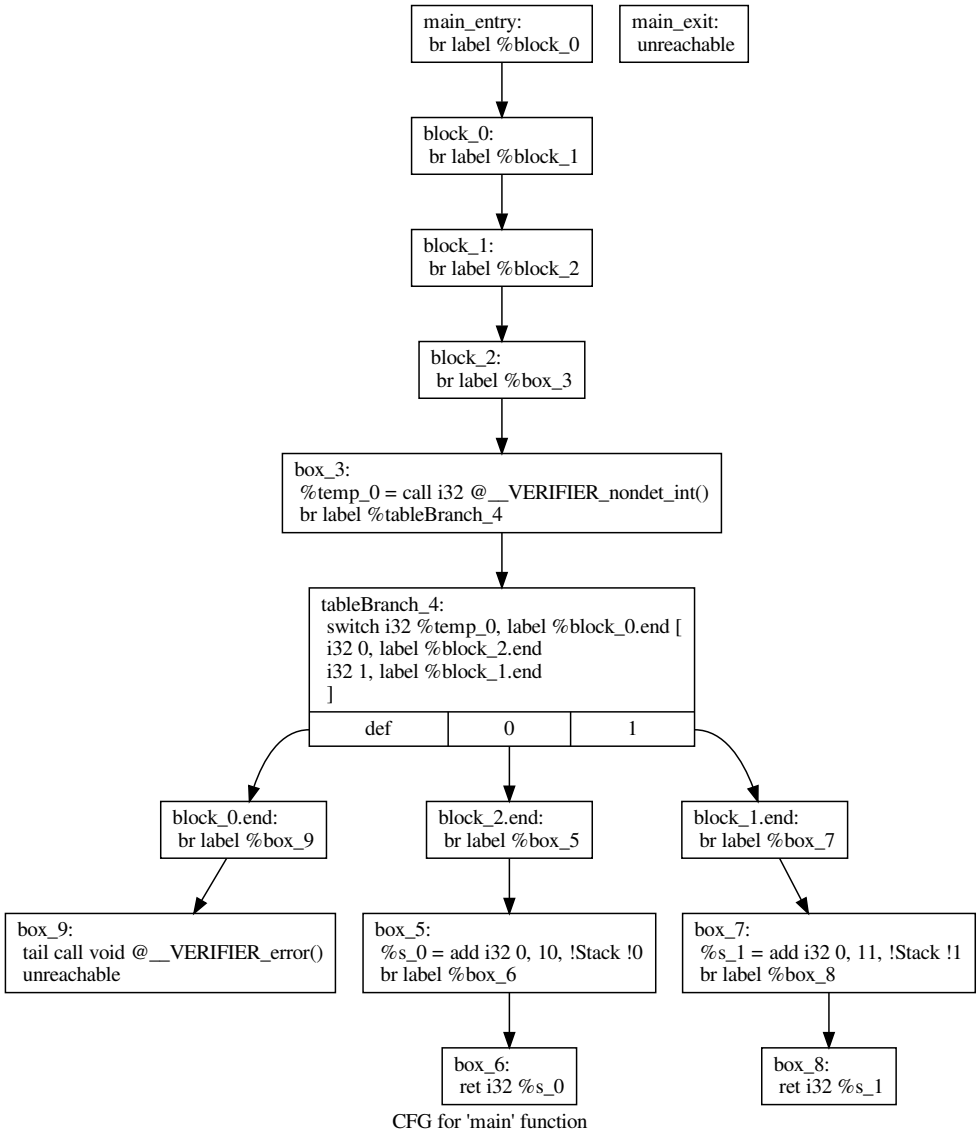

FIGURE C.1: Tools and environment

C.2 CFG diagram for illegal memory access verification

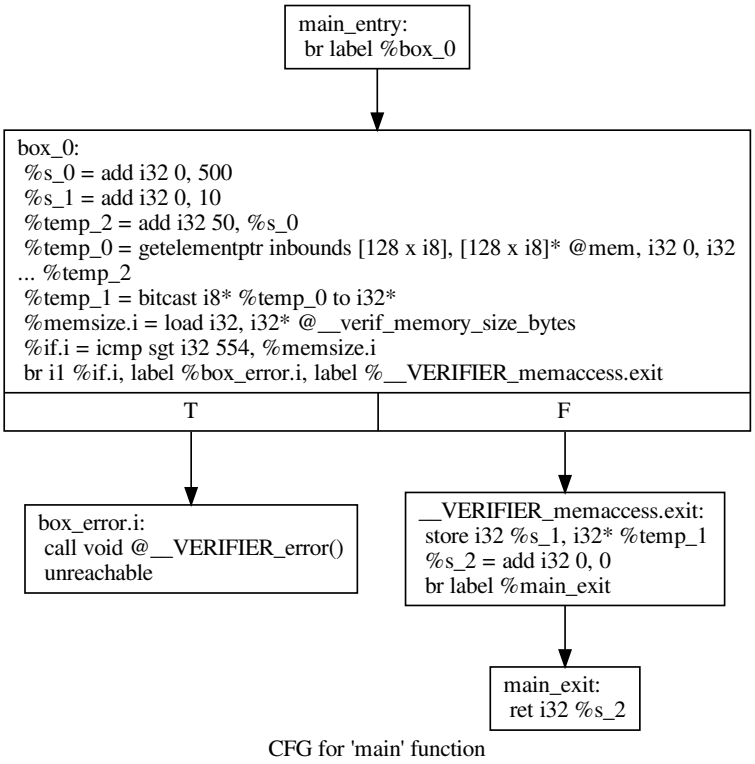

FIGURE C.2: CFG for memory access verification

D | Tables

D.1 Environment configuration

Tool	Version
Scala	2.12.6
JDK	OpenJDK version "1.8.0_191"
Sbt	1.2.1
sbt-rats	2.6.0
Skink	3.0-SNAPSHOT
Emscripten	1.38.27
Clang	6.0.1
WABT[1]	1.0.10
O.S.	Ubuntu 18.04.2 LTS
Memory	15GiB
CPU	Intel® Core™ i7-4500U CPU @ 1.80GHz
	64 bits
	Haswell type

TABLE D.1: Host configuration and used tools.

[1]WebAssembly provides a binary toolkit (WABT) that includes a validator, that validates modules in binary format and is also executed when a module in text format is being translated to binary format. WABT is available at https://webassembly.github.io/wabt