FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor’s thesis

Title:	WebAssembly Memory Debugger
Student:	Jakub Mareš
Supervisor:	Ing. Jan Matoušek
Study program:	Informatics
Branch / specialization:	Software Engineering 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

The ability to compile programs written in manually managed languages such as C and C++ to WebAssembly has introduced the need for a memory debugger similar to Valgrind's Memcheck. The desired outcome of this thesis is a prototype of a memory debugger for WebAssembly with capabilities similar to Valgrind's Memcheck. Immediate application of this memory debugger is in the Trainer teaching tool used during Programming and Algorithmics 1 and 2 courses at FIT CTU. This tool is used for creating, compiling, and executing students' C and C++ programs in their browser and it currently lacks an alternative to Valgrind's Memcheck — an essential tool for detecting memory management errors in programs used by both students and teachers.

Instructions:

1. Analyze WebAssembly and its memory model
2. Analyze the implementation of Valgrind's Memcheck
3. Analyze Trainer requirements for the debugger
4. Design and implement a functional WebAssembly memory debugger prototype that could be used with Trainer
5. Document the prototype
6. Test the prototype
7. Summarize gained experience and propose possible extensions

---

Electronically approved by Ing. Michal Valenta, Ph.D. on 13 February 2024 in Prague.

Bachelor's thesis

WEBASSEMBLY MEMORY DEBUGGER

Jakub Mareš

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Jan Matoušek
May 16, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Jakub Mareš. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Mareš Jakub. *WebAssembly Memory Debugger*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments vii

Declaration viii

Abstract ix

List of Abbreviations x

Introduction 1

1 Analysis 3

1.1 Trainer . . . 3
1.1.1 Current Problems . . . 3
1.2 Common Memory Bugs . . . 4
1.2.1 Memory Leak . . . 4
1.2.2 Double Free . . . 4
1.2.3 Invalid Free . . . 5
1.2.4 Invalid Read and Write . . . 5
1.2.5 Undefined Value Use . . . 6
1.2.6 Null Dereference . . . 6
1.3 Functional Requirements . . . 7
1.4 Nonfunctional Requirements . . . 8
1.5 Valgrind . . . 8
1.5.1 Dynamic Binary Analysis and Instrumentation . . . 8
1.5.2 Shadow Value Tools and Heavyweight DBA . . . 8
1.5.3 Shadow Value Requirements . . . 9
1.6 Valgrind's Memcheck Tool . . . 9
1.6.1 Operation Overview . . . 9
1.6.2 Observable Behavior . . . 9
1.6.3 Operation Specifics . . . 10
1.6.4 Error Reporting . . . 10
1.6.5 Overhead . . . 10
1.6.6 False Positives . . . 10
1.7 WebAssembly . . . 11
1.7.1 Design Goals . . . 11
1.7.2 WebAssembly Text Format . . . 12
1.7.3 WABT . . . 12
1.7.4 Stack Machine . . . 12
1.7.5 Module . . . 13
1.7.6 Memory Model . . . 13
1.7.7 Undefined Values . . . 13
1.7.8 WASI . . . 14
1.8 Instrumentation . . . 14
1.9 Interpretation . . . 14

1.10 Wasabi . . . 15
1.11 DWARF . . . 15
1.11.1 DWARF for WebAssembly . . . 15
1.12 Possible Analysis Methods . . . 15

2 Design **17**
2.1 Wasm Doctor . . . 17
2.2 Wasm Doctor Library . . . 17
2.2.1 Wasm Doctor . . . 17
2.2.2 Validators . . . 17
2.2.3 Reporter . . . 22
2.2.4 Error Blacklist . . . 23
2.2.5 Wasm State . . . 23

3 Implementation **25**
3.1 Implementation Language . . . 25
3.2 Clang and LLVM . . . 25
3.2.1 Clang and LLVM Implementation Specifics . . . 25
3.2.2 Clang's Linear Stack . . . 25
3.3 Instrumentation and Interpretation Comparison . . . 27
3.4 Interpreter Choice . . . 27
3.4.1 WABT wasm-interp . . . 28
3.4.2 Wasm3 . . . 28
3.4.3 WebAssembly spec interpreter . . . 28
3.4.4 Toywasm . . . 28
3.5 Valgrind IR . . . 28
3.5.1 Wasm Doctor Library Initialization . . . 29
3.5.2 Load and Store . . . 29
3.6 Doug Lea's Memory Allocator (dlmalloc) . . . 30
3.7 Wasm Doctor Library Build System . . . 30

4 Testing **31**
4.1 Wasm Doctor Testing . . . 31
4.2 Wasm Doctor Library Testing . . . 31

5 Discussion **33**
5.1 Possible Improvements . . . 33
5.2 Future Work . . . 33

Conclusion **35**

A Wasm Doctor Analysis Examples **37**
A.1 Example 1 . . . 37
A.2 Example 2 . . . 38
A.3 Example 3 . . . 38
A.4 Example 4 . . . 39
A.5 Example 5 . . . 40
A.6 Example 6 . . . 40
A.7 Example 7 . . . 41
A.8 Example 8 . . . 42
A.9 Example 9 . . . 45

B Manual **47**
B.1 Local 47
B.2 WebAssembly Binary 47
B.3 Compilation to WebAssembly 48

Contents of the Attachment **53**

List of Figures

2.1 Trainer with Wasm Doctor diagram 18

3.1 WebAssembly linear memory when compiled by Clang 26

List of Tables

5.1 Comparison of Memcheck and Wasm Doctor capabilities 33

List of Listings

1.1 A simple example of a memory leak 4
1.2 A more advanced example of a memory leak 4
1.3 An example of double free 5
1.4 An example of an invalid free 5
1.5 An example of invalid read 5
1.6 A more advanced example of invalid write 6
1.7 A simple example of undefined value use 6
1.8 An example of null pointer dereference 7
1.9 A simple example of WAT 12

2.1 An example of code resulting in a false positive detected by Wasm Doctor 20
2.2 An example of a false positive detected by Wasm Doctor 20
2.3 Valgrind Memcheck false positive example 20
2.4 An example of false positive detected by both Memcheck and Wasm Doctor . . . 20
2.5 Example of a code which results in an invalid write error 21
2.6 Example of invalid write error detected by heap use validator 21
2.7 Example of a stack allocated buffer overflow 22

3.1 Simplified Toywasm load implementation 29

4.1 An example of test of double free error detection. 32

I am grateful to my supervisor Ing. Jan Matoušek for his technical guidance, moral support, and for noticing my interests, which eventually lead to the proposal to work on this thesis.
I would also like to thank my parents for always being there for me.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 16, 2024

Abstract

The ability to compile programs written in manually managed languages such as C and C++ to WebAssembly has introduced the need for a WebAssembly memory debugger similar to Valgrind's Memcheck. The result of this thesis is Wasm Doctor, a WebAssembly memory debugger inspired by Valgrind's Memcheck. It focuses on the detection of undefined value bugs and the detection of dynamic memory allocation bugs. The detection of memory bugs works best for C programs compiled by Clang, as the compiler choice affects some structures inside the executable, that Wasm Doctor uses for the detection of bugs. Immediate application of this memory debugger is in the Trainer education tool used during Programming and Algorithmics 1 and 2 courses at FIT CTU. The memory debugger utilizes shadow memory for undefined value bug detection. Instead of instrumentation, as is the case for Valgrind, Wasm Doctor uses an existing interpreter — Toywasm. This thesis also outlines the unique problems associated with the analysis of WebAssembly executables.

Keywords memory debugger, WebAssembly, Valgrind, Memcheck, shadow memory, dynamic binary analysis, FIT CTU Trainer, web development environment, interpretation, Toywasm

Abstrakt

Možnost kompilace programů napsaných v programovacích jazycích s manuální režií paměti do WebAssembly vytvořilo potřebu existence WebAssembly paměťového dubuggeru s vlastnostmi podobnými nástroji Valgrind Memcheck. Výsledkem této práce je Wasm Doctor, WebAssembly paměťový debugger inspirovaný nástrojem Valgrind Memcheck. Důraz je především kladen na detekci chyb způsobených nedefinovanými hodnotami a špatnou prací s dynamickou alokací paměti. Detekce chyb souvisejících s pamětí funguje nejlépe pro programy napsané v jazyce C a kompilované překladačem Clang, protože použitý kompilátor ovlivňuje struktury uvnitř spustitelného souboru, které Wasm Doctor používá pro detekci chyb. Aplikací tohoto paměťového debuggeru je výukový nástroj Trainer, který se používá při cvičeních Programování a algoritmizace 1 a 2 na FIT ČVUT. Paměťový debugger používá shadow memory pro detekci chyb způsobených nedefinovanou pamětí. Namísto instrumentace, kterou používá Valgrind, používá Wasm Doctor existující interpret — Toywasm. Tato práce též popisuje unikátní problémy, které souvisí s analýzou spustitelných souborů ve WebAssembly.

Klíčová slova pameťový debugger, WebAssembly, Valgrind, Memcheck, shadow memory, dynamická binární analýza, FIT ČVUT Trainer, webové vývojové prostředí, interpretace, Toywasm

List of Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
BI-PA1	Programming and algorithmics 1
BI-PA2	Programming and algorithmics 2
DBA	Dynamic Binary Analysis
DBI	Dynamic Binary Instrumentation
DWARF	Debugging With Arbitrary Record Formats
ELF	Executable and Linkable Format
IR	Intermediate Representation
ISA	Instruciton Set Architecture
WASI	WebAssembly System Interface
Wasm	WebAssembly
WAT	WebAssembly Text Format

Introduction

The compilation of programs written in manually managed languages, such as C and C++, to WebAssembly has created a demand for a memory debugger similar to Valgrind's Memcheck. The immediate application of this memory debugger is in the "Trainer" education tool utilized during Programming and Algorithmics 1 and 2 courses at FIT CTU. This tool compiles and executes students' C and C++ programs within their browser, but it currently lacks a substitute for Valgrind's Memcheck, an important tool for detecting memory management errors in programs used by both students and teachers.

This thesis concentrates on Valgrind and Valgrind's Memcheck analysis, WebAssembly analysis, and the unique requirements of a memory debugger tailored for WebAssembly. The focus is on the utilization of shadow memory for the detection of uses of undefined values. The relative simplicity of WebAssembly, as compared to alternatives such as x86, enables implementation without the use of an intermediate representation. Furthermore, the performance of the resulting memory debugger is not a concern, which allows the use of alternatives to instrumentation.

This thesis shows that there is a need for a memory debugger for WebAssembly programs, even when its code is validated and executed in a memory-safe environment. It presents the differences between WebAssembly and other ISAs, especially x86, from the point of view of shadow memory DBA tools.

The goal of this thesis is to build a prototype of a memory debugger for WebAssembly with functionality inspired by Valgrind's Memcheck. Since a memory debugger with such functionality capable of being compiled to WebAssembly and validating a program compiled to WebAssembly does not exist, this thesis aims to analyze Valgrind and its Memcheck tool to inspire a memory debugger suitable for WebAssembly. It has to be possible to integrate the memory debugger with Trainer, as the memory debugger has an immediate application in it. To ensure maximum portability and standalone functionality, the memory debugger prototype depends on as little environmental information as possible, such as the compiler used to compile the analyzed program to WebAssembly.

Chapter 1
Analysis

In this chapter, the requirements for a WebAssembly Memory Debugger are outlined, as specified by the Trainer education tool. An analysis of Valgrind and its Memcheck tool is provided along with the analysis of WebAssembly memory model. Along with Memcheck, it analyzes the concept behind its bit-precise definedness checking — shadow memory. It also outlines common errors in student programs and presents possible analysis methods.

1.1 Trainer

Trainer is a unique education tool used during the BI-PA1 and BI-PA2 laboratories, which provides an environment in which students can write and run their C and C++ written programs.

It utilizes WebAssembly to compile and run the students' code in their own browser. This means that the compiler which compiles the students' programs has been itself compiled into WebAssembly. This makes it easier especially for beginners to focus on the assignments and not on solving unrelated compiler and similar issues.

1.1.1 Current Problems

Several unique problems stem from the way Trainer utilizes WebAssembly.

Because 0 is a valid address in WebAssembly, the students' programs do not end in a segmentation fault when they dereference null pointers. This causes a discrepancy between Trainer, local development, and Progtest, a system for assignment of homework and the automatic evaluation of the students' solutions, used at FIT CTU. More importantly, it thus lacks the immediate feedback when attempting to do something that is, at least on other architectures, wrong.

Currently, there is no detection of undefined value uses.

The detection of double free and memory leaks is currently provided by the use of customized `malloc()` (specifically `dlmalloc()`) and `free()` functions, which count the number of `malloc()` and `free()` calls and this way make sure that at least the number corresponds. Valgrind's Memcheck also utilizes customized `malloc()` and `free()` calls due to historical reasons, but in a more sophisticated way [1].

Trainer compiles the students' programs into WebAssembly to be able to run them in their browser, because of that it can not use the usual tools used during C and C++ development like GDB or Valgrind.

1.2 Common Memory Bugs

In this section common memory bugs are presented. It focuses on memory bugs in C. Memory bugs can also occur in other languages with manual memory management. Most of the examples apply for C++ too, but there are some differences in syntax, for example `malloc()` would be `new`, and `free()` would be `delete`.

1.2.1 Memory Leak

Memory leak is a type of memory bug that occurs when an allocated block of memory is not freed after its use. A simple example is 1.1. In case the user program runs on a modern OS like Linux, all memory used by the program during execution will be returned. But that does not mean that memory leaks are not problematic. Programs that do not properly deallocate memory throughout execution may eventually run out of memory, resulting in a crash. Memory leaks are especially dangerous for kernel processes, where they can lead to serious system stability issues. [2]

```
int *numbers = (int*)malloc(10 * sizeof(*numbers));

// code without a free(numbers) call
```

Listing 1.1 A simple example of a memory leak

Memory leaks can be more difficult to notice, especially for begginers. In 1.2 is an example of an allocation of an array of arrays. There is a memory leak present because, although all memory allocated for each individual name is freed. The array of names is not freed. In the examples, the code that allocates and frees memory is located in a single scope. This is usually not the case for programs, and it makes these errors less noticeable.

```
int number_of_names = 4;

char **names = (char**)malloc(number_of_names * sizeof(*names));

names[0] = (char*)malloc(5 * sizeof(**names));
names[1] = (char*)malloc(8 * sizeof(**names));
names[2] = (char*)malloc(7 * sizeof(**names));
names[3] = (char*)malloc(5 * sizeof(**names));

for (int i = 0; i < number_of_names; ++i) {
  free(names[i]);
}
```

Listing 1.2 A more advanced example of a memory leak

1.2.2 Double Free

Double free is a type of memory bug that occurs when free is called more than once on a singular allocated block of memory.

Although it seems improbable for double free to occur, there are circumstances in which it is not as trivial to spot it, as in example 1.3. The two `free()` calls can be separated by hundreds of lines of code or possibly be located in different files. Error conditions and exceptional circumstances along with it being not obvious which part of the program should be responsible for freeing the memory. [3]

```
int *numbers = (int*)malloc(10 * sizeof(*numbers));

free(numbers);
free(numbers);
```

Listing 1.3 An example of double free

1.2.3 Invalid Free

Invalid free occurs when `free()` is called with an address that does not correspond to any address of an allocated block. This may be caused by, as is the case in example 1.4, off by one error.

```
int number_of_names = 4;

char **names = (char**)malloc(number_of_names * sizeof(*names));

names[0] = (char*)malloc(5 * sizeof(**names));
names[1] = (char*)malloc(8 * sizeof(**names));
names[2] = (char*)malloc(7 * sizeof(**names));
names[3] = (char*)malloc(5 * sizeof(**names));

for (int i = 0; i <= number_of_names; ++i) { // i is incremented upto 4
  free(names[i]);
}
```

Listing 1.4 An example of an invalid free

1.2.4 Invalid Read and Write

Invalid read or write is detected when a region of memory is accessed outside of allocated memory blocks, global data sections, and the stack.

```
int number_of_names = 2;

char **names = (char**)malloc(number_of_names * sizeof(*names));
names[0] = (char*)malloc(5 * sizeof(**names));
names[1] = (char*)malloc(5 * sizeof(**names));

// The string "John" is copied into memory that was not allocated
strcpy(names[2], "John");
```

Listing 1.5 An example of invalid read

In 1.6 invalid write happens in `strcpy()`, because there are 4 bytes allocated for the name string, but because C strings are null-terminated (`"John\0"`) there are 5 bytes needed for it to fit.

```
char *name = (char*)malloc(4 * sizeof(*name));

strcpy(name, "John");
```

Listing 1.6 A more advanced example of invalid write

1.2.5 Undefined Value Use

Undefined value use is a type of memory bug. Example 1.7 shows how an undefined value can affect the execution of a program. The variable `x` is declared but not initialized. This means that `x` will contain whatever value is in its place in memory, and thus the conditional jump in the resulting assembly will depend on an undefined (uninitialised) value. This results in undefined behavior of the program.

```
int x;

if (x) {
  // do something
}
```

Listing 1.7 A simple example of undefined value use

1.2.6 Null Dereference

A null pointer dereference occurs when a pointer is used in a way that expects it to contain a valid address in memory [4]. An example of nontrivial null pointer dereference is in 1.8.

```c
#include <stdlib.h>

struct node {
  int x;
  struct node *next;
};

int main() {
  struct node node_b = {.x = 2, .next = NULL};
  struct node node_a = {.x = 1, .next = &node_b};

  int x = 0;
  int accumulator = 0;
  struct node *current_node = &node_a;
  while (x <= 2) {
    // null pointer dereference
    accumulator += current_node->x;
    ++x;
    // second null pointer dereference
    current_node = current_node->next;
  }

  return 0;
}
```

Listing 1.8 An example of null pointer dereference

1.3 Functional Requirements

From the discussion with Ing. Jan Matoušek, the following functional requirements were established.

F1 Undefined Memory Access Detect and report the access of undefined values from WebAssembly linear memory. This requirement has a high priority.

F2 Undefined Local Variable Access Detect and report the loading of undefined WebAssembly local variables. This requirement has a medium priority.

F3 Use After Free Detect and report the loading from already freed heap blocks and storing to them. This requirement has a high priority.

F4 Memory Leaks Detect and report memory leaks. This requirement has a medium priority, because basic detection is already implemented in Trainer.

F5 Double Free Detect and report free of an already freed heap block. This requirement has a medium priority, because basic detection is already implemented in Trainer.

F6 Invalid Free Detect and report free called on address that does not contain a heap block. This requirement has a high priority.

F7 Invalid Reads and Writes Detect and report reads from and writes to WebAssembly linear memory outside of allocated heap blocks, global data segments and Clang's linear stack. This requirement has a high priority.

F8 Null Pointer Report the access of the address 0, as it indicates access using null pointer. This requirement has a medium priority.

1.4 Nonfunctional Requirements

NF1 Executability in WebAssembly Wasm Doctor is able to run in a WebAssembly environment. This is important for the use in Trainer. This requirement has a high priority.

NF2 Portability Wasm Doctor depends on the smallest possible number of environmental information. Where it is beneficial, especially for Trainer, implement the specific features as opt-in to maintain at least basic level of analysis for all WebAssembly programs. This requirement has a lower priority.

1.5 Valgrind

Valgrind is an instrumentation framework used for the creation of heavyweight DBA (Dynamic Binary Analysis) tools. It uses dynamic binary instrumentation (DBI). It provides a number of tools, one of which is Memcheck. [5]

1.5.1 Dynamic Binary Analysis and Instrumentation

Program analysis tools, such as profilers and error checkers, are a popular way to improve the quality of software. One type of tools that analyze the program at run-time at the level of machine code are called dynamic binary analysis tools. [5]

DBA tools are frequently implemented using dynamic binary instrumentation (DBI), a method that incorporates analysis code into the original code of the user's program during run-time. This is advantageous since no preliminary steps, like recompiling or relinking, are necessary. Moreover, it ensures complete coverage of user-mode code without the need for source code. As it is dynamic analysis it can only check the parts of the code that are actually executed. DBI frameworks enable DBA tools to work as plugins which extend their core functionality, instrumentation, and execution of code, with their own functionality. [5, 1]

1.5.2 Shadow Value Tools and Heavyweight DBA

Shadow value tools are a subset of DBA tools which are important to this thesis. Such tools shadow every value, whether it is stored in memory or in a register or an equivalent construct, for example, a local variable in the case of WebAssembly. There are a number of shadow value tools that use shadow values in different ways. [5] Some examples of uses include:

Memcheck for definedness checking of values for detection of dangerous uses of undefined ones. In 2007 it was already used by thousands of C and C++ programmers and was probably the most widely-used DBA tool in existence. [5] Memcheck is in this thesis' author's experience sometimes inaccurately referred to as Valgrind.

TaintCheck tracks the use of tainted values that originate from untrusted sources. [5]

Annelid for tracking array bounds errors. [5]

"Shadow value tools are a perfect example of "heavyweight" DBA tools. They involve large amounts of analysis data that is accessed and updated in irregular patterns. They instrument many operations (instructions and system calls) in a variety of ways — for example, loads, adds, shifts, integer and FP operations, and allocations and deallocations are all handled differently. For heavyweight tools, the structure and maintenance of the tool's analysis data is comparably complex to that of the client program's original data. In other words, a heavyweight tool's execution is as complex as the client program's. In comparison, more lightweight tools such as trace

collectors and profilers add a lot of highly uniform analysis code that updates analysis data in much simpler ways (e.g. appending events to a trace, or incrementing counters).” [5]

“Shadow value tools are powerful, but difficult to implement. Most existing ones have slow-down factors of 10x–100x or even more, which is high but bearable if they are sufficiently useful.” [5]

1.5.3 Shadow Value Requirements

In the following text, the requirements of shadow value tools are presented. It shows that the requirements of shadow value tools are universal and not Valgrind specific. [5]

“There are three characteristics of program execution that are relevant to shadow value tools: (a) programs maintain state, S, a finite set of locations that can hold values (e.g. registers and the user-mode address space), (b) programs execute operations that read and write S, and (c) programs execute operations (allocations and deallocations) that make memory locations active or inactive.” [5]

1.6 Valgrind's Memcheck Tool

Memcheck is the most used of the Valgrind's available tools. It is used for detection of undefined memory and register uses, detection of heap use bugs such as double free or memory leaks, addresability tracking of every byte of memory, and detection of overlap in memory blocks supplied to functions like `strcpy()` and `memcpy()`. Detection of undefined memory and register uses (definedness checking) is the most sophisticated of them. Memcheck's differentiating factor from alternatives is mainly its bit-precision detection of undefined value uses, which is possible by the use of shadow memory. Shadow memory contains a definedness (validity) bit for each bit in memory. [1, 5]

1.6.1 Operation Overview

Memcheck's definedness checking is built upon three ideas. [1]

Firstly, every bit of data, both in registers and in memory, has a shadow bit associated with it which contains information about its definedness. [1]

Secondly, every operation that creates a value has a shadow operation associated with it, which computes the definedness information of the output from the definedness information of the input. [1]

Thirdly, every operation that uses a value in a way which could affect observable behavior is checked. If the inputs of such operation are undefined, Memcheck reports an error. If the operation can not affect the observable behavior, even when the inputs are undefined, the output definedness is computed based on these (partially) undefined inputs, but no errors are reported as it is possible that the operation does not have an effect on any observable behavior. [1]

1.6.2 Observable Behavior

Observable behavior can be affected by several actions. Memory exception caused by the use of undefined address in a load or store, conditional jump based on undefined values, passing undefined values to a system call, and loading uninitialised values from memory into a SIMD or FP register. [1]

1.6.3 Operation Specifics

1.6.3.1 Lazy and Eager Approximation Schemes

There are two ways of approaching unknown operations.

Firstly, in lazy approximation scheme the bits of all inputs are pessimistically united into a single bit describing the worst-case definedness of the result. This way there are no errors reported when undefined bits enter the inputs of such operations, rather the undefinedness “flows” through the unknown operations. [1]

Secondly, in eager approximation scheme the bits of all inputs are also pessimistically united into a single bit describing the worst-case definedness of the resutl, but if the result is undefined, error is reported right away. [1]

Memcheck uses eager approximation scheme for floating point and SIMD operations, and lazy approximation for all other operations. [1]

1.6.4 Error Reporting

There are two possible ways to handle the issuing of error messages. When an undefined value is consumed by an operation, Memcheck has to decide whether to report the error right away and mark the result as defined or propagate the undefinedness into the result. [1]

The first option is to report the error right away. Uninitialised memory is the main origin for undefined value errors. Disadvantage of this strategy is that these errors can propagate for a fairly long time before they hit a check point and the error report can thus lose locality information accuracy. [1]

The second option is to delay the error checking and reporting which has two advantages. Improved performance, as the error checks are computationally expensive, and more importantly reduction of the number of false positives detected. This stems from the fact, that the undefined values may be used in a safe manner and then not used in a way that would affect the observable behavior of the program. This is in contrast with the eager strategy which would report a pointless error. [1]

Memcheck mostly uses the second option. [1]

1.6.5 Overhead

The overhead of Memcheck is defined by two main factors. Memory use of programs being checked by Memcheck effectively doubles, as for each bit of data there has to be a definedness bit. The time overhead is caused by the need to compute output definedness of most instructions (operations) as most of them compute new values. This means that one or more instructions need to be added for the definedness computation itself. [1]

1.6.6 False Positives

The amount of false positives detected by Valgrind is very low. There is a small amount of hand-coded and even smaller amount of compiler-generated assembly sequences that are known to cause false positives. [1]

One example is `xor %reg,%reg`, the value in `%reg` is defined even when the input of the operation is undefined. This is solved by Valgrind’s use of intermediate representation, because when this specific case is encoutered during translation from x86 to UCode (Valgrind’s IR) it is translated as if `mov $0,%reg` was encoutered before the `xor %reg,%reg`. [1]

Other interesting source of false positives is GNU libc which contains highly-optimized, hand-written assembly for common string functions. These functions, particularly `strlen()`, traverse the string one word at a time and rely on detailed properites of carry-chain propagation for correct

behavior. Memcheck's way of handling such propagation with its `Left` operator is not precise enough and causes false positives. The `Left(v)` unary operation simulates the worst possible outcome of carry-chain propagation of undefined bits. This simulation works by setting all bits left from the rightmost undefined bit as undefined. For example, `Left(00010100) = 11111100`. For optimization reasons the undefined bit is, somewhat counterintuitively, represented by a 1 and defined bit is represented by a 0. [1]

Memcheck implements a workaround for this false positive and in [1] presents a better solution. The workaround consists of two parts. It replaces the GNU libc versions of such functions with its own, less optimized version that do not contain the optimizations that cause the false positives. A problem of this solution is that GCC sometimes inlines calls to these functions, and thus the replacement may not work in all cases. This is solved in a way that is described as a nasty hack. It counts on the fact that the code of these optimized functions contains the addition or subtraction of carefully chosen constants, such as 0x80808080. If the code contains additions or subtractions of these constants, Memcheck omits the portion of undefined value checks in the relevant basic block. This omission of undefined value checks may result in false negatives. [1]

A better solution for the inlined string functions from GNU libc is to use the presence of the mentioned constants as a signal for a more sophisticated and expensive instrumentation strategy, which focuses particularly on proper carry propagation. [1]

1.7 WebAssembly

In 2019 Solomon Hykes, the founder of Docker, posted: *"If WASM+WASI existed in 2008, we wouldn't have needed to created Docker. That's how important it is. WebAssembly on the server is the future of computing."* [6]

"WebAssembly (Wasm) is a safe, portable, low-level code format designed for efficient execution and compact representation. At its core, it is a virtual ISA (instruction set architecture). Its main goal is to enable high performance applications on the Web, but it does not make any Web-specific assumptions or provide Web-specific features, so it can be employed in other environments as well." [7] It enables the use of many existing programming languages (e.g. C, C++, Rust, Go, Lua [8]) on the Web and thus enables execution of existing programs, for example in case of Trainer the C and C++ compiler, in the browser.

1.7.1 Design Goals

WebAssembly's design goals focus on creating fast, secure, and easily transferrable software. It targets fast execution times by leveraging common capabilities across modern hardware, leading to close-to-native performance levels. This results in lower latency and improved user experience. [7]

Security is essential in WebAssembly. The code is validated and executed in a memory-safe, sandboxed environment to prevent data corruption or security breaches. Although WebAssembly's linear memory model is safe, which means that WebAssembly's memory model can not be broken, the program itself can still corrupt its own memory inside the linear memory. [7]

Additionally, WebAssembly provides clear rules for defining valid programs and their expected behaviors, making them easier to analyze both informally and formally. These rules apply consistently regardless of the target platform. Furthermore, it does not favor any specific programming language, model, or object structure — promoting inclusivity and flexibility for different tools and approaches. [7]

WebAssembly is hardware independent and can be compiled on all modern architectures. That means architectures used in desktops, mobile devices, and embedded devices. It is also platform independent in the sense that it can run in a browser or in a separate virtual machine. WebAssembly specifies the way in which programs can access and interact with their

environment. [7]

The encoding technique used in WebAssembly emphasizes simplicity and efficiency. It comes in a compact binary format designed for fast transmissions, saving bandwidth compared to conventional text or native code formats. Modularity plays a key role, as programs are broken down into smaller units that can load independently, cache efficiently, and consume fewer resources. [7]

Streamability allows fast decoding, validation, and compilation, since it can be done even before all the data have been seen. Being parallelizable allows for splitting of decoding, validation, and compilation into parallel tasks. It makes no architectural assumptions not broadly supported across modern hardware, which would limit its use on it. [7]

1.7.2 WebAssembly Text Format

WebAssembly Text Format (`.wat`) is a textual representation of a WebAssembly module abstract syntax. Similarly to Lisp it uses S-expressions. The instructions can either be written in order or, for improved readability, folded in S-expression form. [9, 10]

A simple example of what WAT looks like can be seen in 1.9.

```
(module

  (memory $memory 1)
  (export "memory" (memory $memory))

  (func (export "store_in_mem") (param $num i32)
    i32.const 0
    local.get $num

    ;; store $num at position 0
    i32.store
  )

)
```

Listing 1.9 A simple example of WAT [11]

1.7.3 WABT

WABT (WebAssembly Binary Toolkit) is a suite of tools for WebAssembly. [12]

An important tool for the analysis of WebAssembly files done during this thesis is wasm2wat. It is used for translating from binary to text format [12]. WABT also contains one of the interpreters considered for Wasm Doctor — wasm-interp [12].

1.7.4 Stack Machine

"WebAssembly is a binary instruction format for a stack-based virtual machine." [13] It is somewhat similar to Java bytecode, but it is programming language and paradigm agnostic [7], whereas Java bytecode is specific to Java and its object oriented nature. WebAssembly instructions are executed in order and manipulate the values on the operand stack. It also utilizes local variables, which are similar to registers [14] and can be written (`local.set`) to and read

(`local.get`) from. The instructions do not manipulate these values directly. To write a value to a local variable, it has to be first put on the operand stack and then the `local.set` *identifier* has to be called. Similarly, a value is read by calling `local.get` *identifier*. This puts the value on top of the operand stack.

This is a distinction from x86, where local variables and other information (e.g. function call data) are stored on the stack and thus located in memory. The WebAssembly stack machine does have an operand stack, but it is separate and is not part of its linear memory, as WebAssembly is a Harvard architecture [14]. It does not have a stack or a stack pointer. This presents an untrivial challenge for the WebAssembly Memory Debugger, as Valgrind's validation of memory addresses depends on the use of a stack pointer. The concepts of stack and stack pointer are added by, for example, LLVM, to the linear memory [14].

1.7.5 Module

WebAssembly module is a unit of deployment, loading, and compilation that contains the definitions of types, functions, tables, memories, and globals. A module can define which functions are imported or exported, it can define a start function, and define the data segments [15]. Memories and data segments are important to this thesis. From the point of view of this thesis, a module is similar to ELF (Executable and Linkable Format) used, for example, for Linux executables. For example the `.data` and `.rodata` can have their equivalent in WebAssembly module's `(global $.data ...)` and `(global $.rodata ...)`. The use of globals in this way is not related to WebAssembly itself, but rather to the way Clang compiles the program into WebAssembly [14].

1.7.6 Memory Model

WebAssembly uses linear memory, where memory means a vector of raw uninterpreted bytes. The WebAssembly specification references memories, but it currently supports at most one memory, and all constructs implicitly reference this memory 0. Although this restriction may be lifted in future versions, the rest of this thesis will take this into account and only refer to it as memory. [16]

Linear memory is addressed from 0, meaning that 0 is a valid address. [15] This, along with the fact that all values in memory are initially zeroed [17], has implications for programs, written in C or C++, where the address 0 is reserved for `NULL` or `nullptr`.

1.7.7 Undefined Values

One important difference of WebAssembly from other instruction set architectures, such as x86, is that memory is initially zeroed. [17]

This has important effects on the use of, from the programming language's (e.g. C, C++) point of view, undefined values. Because of that, when a value is read in WebAssembly from previously undefined variable, the value is 0. [18]

This means that the use of undefined variables does not result in undeterministic behavior. One could argue that even in C and C++, languages that do not set variables after declaration to any specific value, but rather take the value that is left at its place in memory, it may be the programmers intention to use this determinism to their advantage. Unfortunately, this may create a type of bug that is difficult to detect. If the code was firstly written with WebAssembly as the primary target, it could go its whole development without any issues with the use of undefined values, because they would be defined and be 0. But if for some reason there was a need to port this code to a more traditional target, like x86, it would pose a great challenge

because the once deterministic decisions and calculations based on undefined values would become undeterministic.

The use of undefined values is a common problem in the code of beginner students. As Trainer is an education tool, it is necessary to make the use of undefined values apparent and discourage it. Currently, a simple strategy is implemented. Linear memory is filled with predefined garbage data. This partially solves the problem of students relying on undefined variables being zeroed. Even still, as was confirmed by Ing. Jan Matoušek, the students still sometimes rely on the garbage data being always exactly the same. This problem could be solved by randomization of the garbage data, but still the students' program might function properly a handful of times due to chance. The resulting memory debugger, Wasm Doctor, is capable of reliably detecting the use of undefined values and reporting it to the student. The fact that the error can be reported to the student, rather than relying on the student's program not working as expected, is an important advantage.

1.7.8 WASI

WASI (WebAssembly System Interface) is a set of APIs for WebAssembly. It is in active development, and the LLVM compilers currently use the version called Preview 1 with the import module name `wasi_snapshot_preview1`. [19, 20]

During the writing of this thesis a Preview 2 has been published, but for the purposes of this thesis it is too new. [21]

The purpose of WASI is to enable WebAssembly code to interact with the outside world through WebAssembly-native APIs while maintaining the essential sandboxed nature of WebAssembly. The API design has capability-based security, which means that all access to external resources is provided by capabilities. [19]

Build on top of WASI system calls is a libc implementation for WebAssembly called wasi-libc. It provides the support for standard IO, file IO, filesystem manipulation, memory management, time, string, environment variables, program startup. [22]

WASI and wasi-libc are crucial for Trainer. As explained in 1.6.6 and 2.2.2.1 an unoptimized version of wasi-libc is necessary for Wasm Doctor to work not report a number of false positives inside the functions provided by it.

1.8 Instrumentation

Code instrumentation is a technique used to insert new code into an existing program. It is usually used in a way that does not alter the behavior of the existing program but rather makes the retrieval of information about the running program possible. [23]

Instrumentation refers to the capability of code tracing, a technique used during development to gain insight into the inner workings of the program on a lower level than is offered by logging, debugging used for the detection of errors, performance counters (profilers), used for the monitoring of programs performance, and event logging, which provides high level information used for diagnostics and auditing by system administrators. [23]

1.9 Interpretation

Interpretation is a way of executing a program. The interpreter reads and directly executes the source code as it is read. It is different from compilation, which takes the source code and translates it into assembly or machine code which is later executed. [24]

1.10 Wasabi

Wasabi is a general-purpose DBA framework for WebAssembly binaries. The programming language used for the analysis is Javascript. The framework works by inserting WebAssembly code that calls analysis functions into the WebAssembly program's original instructions. This can be unnecessarily computationally expensive, so it is possible to only instrument the instructions that are relevant for a particular analysis. This process is called selective instrumentation. [25]

It comes with a number of tools built on top of it, but they are rather simple and are used to demonstrate the ease of use of the framework. None of the tools offer capabilities similar to Valgrind's Memcheck. [25]

1.11 DWARF

DWARF (Debugging With Arbitrary Record Formats) is a debugging information file format. It enables debuggers to use information about the original source code. [26]

DWARF was developed alongside the ELF object file format. This is why it is most commonly associated with it, but it is independent of the object file format. [27]

1.11.1 DWARF for WebAssembly

DWARF debug information can be embedded inside a WebAssembly file or it can be located inside a separate file. In case of embedding the DWARF debug information inside a WebAssembly binary, each DWARF section has a corresponding WebAssembly section. The section names in WebAssembly match the section names defined in the DWARF standard. [28]

1.12 Possible Analysis Methods

There are multiple ways to handle undefined memory use error detection. One of them is, similarly to Memcheck, through the use of shadow memory and bit-precise mirroring of operations. Other tools do not use shadow memory, and their detection is less precise (on the level of bytes), but their implementation is simpler, and their overhead is lower. Wasm Doctor utilizes a hybrid approach, which is constructed in a way to allow for more precision through the same mirroring of operations as Memcheck. This hybrid approach is possible because of WebAssembly's relative simplicity when compared to x86 and the development time saved by the lack of need for abstraction through an intermediate representation.

Chapter 2
Design

This chapter describes the design of Wasm Doctor, Wasm Doctor library, and Toywasm.

2.1 Wasm Doctor

Wasm Doctor is a memory debugger that targets WebAssembly compiled C code. It currently uses a fork of an existing interpreter, Toywasm, as a way to execute and analyze the WebAssembly code. The main functionality of the analysis is implemented as a standalone library, and thus it is fairly independent of the interpreter chosen. The analysis functions from the library should even potentially be able to be called from Wasabi hooks.

The diagram 2.1 shows how a C program is written, compiled, and run inside Trainer. It also shows how Wasm Doctor is expected to be integrated with Trainer. Notice that Wasm Doctor itself is compiled into WebAssembly and executes the WebAssembly binary provided by the compilation step. This means that the browser executes the Wasm Doctor, which executes the student's program.

The way Wasm Doctor is used for the analysis of WebAssembly programs is shown in appendix B and the analysis results are presented in appendix A.

2.2 Wasm Doctor Library

Wasm Doctor library is implemented as a suite of C functions that are meant to be called during WebAssembly execution. It contains the code that performs the memory error detection. It contains the implementation of shadow memory, Clang/LLVM Webassembly C ABI specific linear stack validation, etc.

2.2.1 Wasm Doctor

All the possible analysis functions are exposed through functions provided by `wasm_doctor.h`. In this way, the state of the execution is properly maintained and the user of this library does not have to interact with the validators themselves.

2.2.2 Validators

The detection of errors is performed inside a set of validators. Each validator is responsible for the detection of a specific error type.

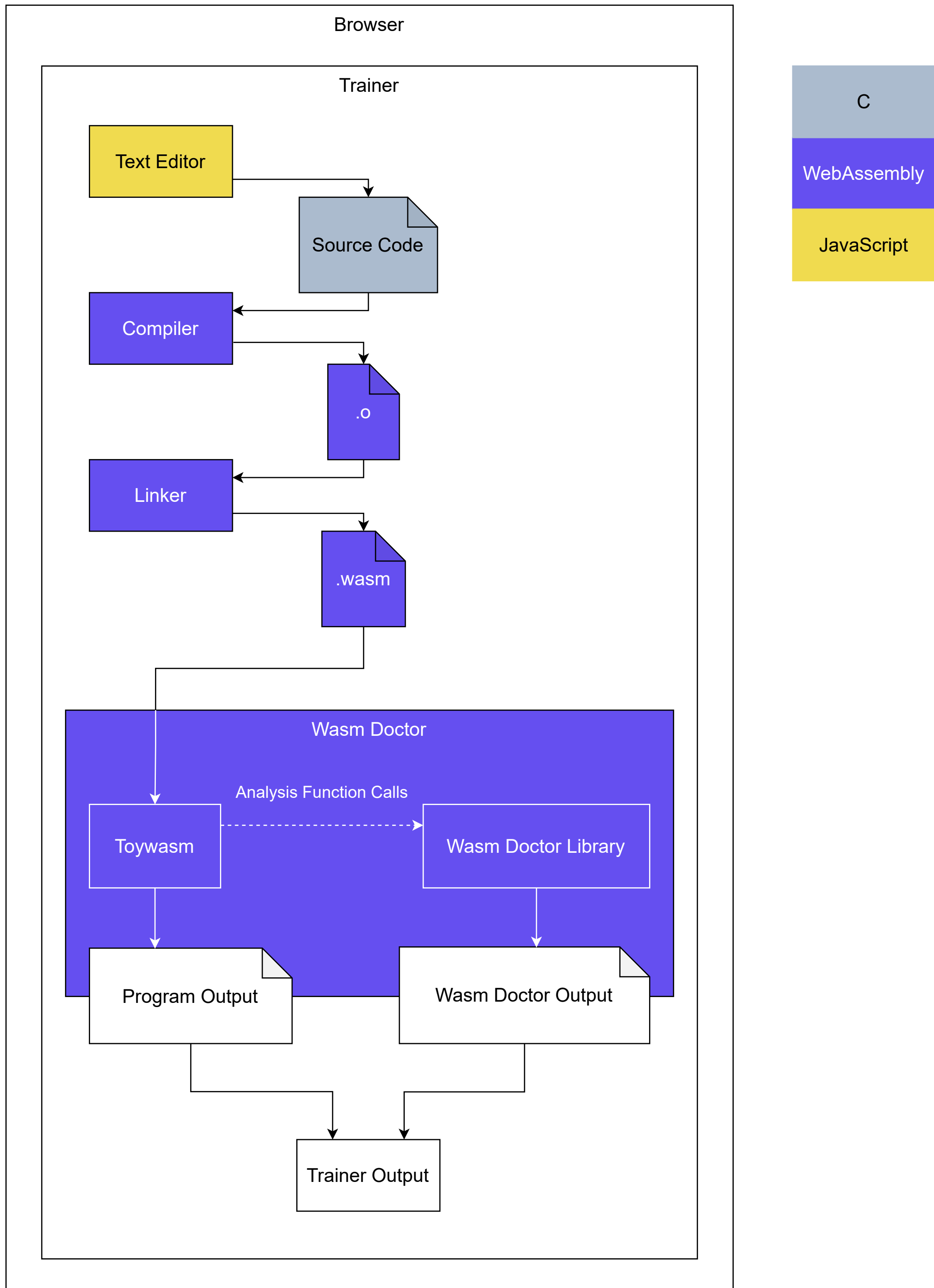


Figure 2.1 Trainer with Wasm Doctor diagram

2.2.2.1 Shadow Memory Validator

Shadow memory validator is responsible for the detection of undefined memory uses as described in the functional requirement 1.3. For this purpose, it uses shadow memory. Thus, similarly to Valgrind's Memcheck [1], it can report undefined memory use with bit precision.

Shadow memory in shadow memory validator is implemented as a bit array. Validation and invalidation are performed through a set of functions in it.

Wasm Doctor's way of using shadow memory for undefined value error checking is different from Memcheck's in two ways. Firstly, Wasm Doctor does not mark the region of memory that causes an undefined value error as defined after detection. During testing, repeated errors provided more insight into the origin of the error and thus were not suppressed, but it does not pose a significant difficulty to change the behavior may the need arise. Secondly, Wasm Doctor does not wait with its error reporting until an operation that can affect observable behavior is executed, as is described in 1.6.1 and 1.6.4. Although this approach may result in an increased number of false positives detected, none that were not possible to solve by the use of unoptimized libc functions were caused by this simlified approach were detected during the testing of example common student errors. The way Wasm Doctor simplifies its definedness checking should not produce more false negatives as its main problem is its high sensitivity.

Similarly to Memcheck (1.6.6), where `strlen()` and similar functions have to be substituted with their unoptimized counterparts, shadow memory validator reports false positives when checking their optimized counterparts. This is solved by supplying the Trainer compiler with unoptimized library functions like `strlen()`.

A proposed way of improved definedness checking is to maintain a shadow operand stack, which would maintain a mirrored state of the actual WebAssembly operand stack and would allow Wasm Doctor to determine whether an instruction with observable behavior depends on an undefined value or not. This approach was not developed mainly due to time constraints. The development time was focused on other memory bug detection methods, as it was determined to be more beneficial for the prototype, and the current approach to definedness checking was deemed good enough.

In the example 2.1 the code analysis ends in a false positive by Wasm Doctor (2.2). Wasm Doctor reports a false positive because `undefined_coords` is copied when it is taken as a parameter of the function `zero_coordinates()`. Wasm Doctor interprets this copying as an undefined value use. As the programs only observable behavior, for simplicity, is the statement `return 0` and does not depend on any undefined values, Valgrind does not report any errors.

An example that the author of this thesis thought would also showcase Memcheck's better accuracy is in the source code 2.3. There seems to be a use of an undefined value, but although the value from the variable x is undefined, it is used in a manner that makes its value defined, specifically all its original bits are shifted left, until only 0 remains. This means that although an undefined value is loaded, its undefinedness does not affect the observable behavior. Surprisingly, Valgrind's Memcheck reports it as: *Conditional jump or move depends on uninitialised value(s).* Wasm Doctor also reports an error, as can be seen in 2.4.

2.2.2.2 Local Validator

Local validator detects the use of WebAssembly local variables that have not been assigned any value as described in functional requirement F2 (1.3). The detection of such uses suffers from a high number of false positives, because, as described in subsection 1.7.6, all local variables are initially zeroed, and compilers take advantage of that. The detection of these previously unassigned local variables proved to be not very useful. The main reasons are the high number of zeroed uses and thus error detections, and the analysis of unoptimized code where all the values seemed to be loaded from linear memory anyway. Higher levels of optimization during compilation were tried, but did not improve the local validator's performance.

```
struct coordinates {
  uint8_t x;
  uint8_t y;
};

struct coordinates zero_coordinates(struct coordinates c) {
  c.x = 0;
  c.y = 0;

  return c;
}

int main() {
  struct coordinates undefined_coords;

  // struct coordinates contains undefined values at this point

  /* struct containing undefined values is copied
     but the undefined values are not in an execution altering way */
  struct coordinates zeroed_coordinates = zero_coordinates(undefined_coords);

  // struct coordinates contains defined values at this point

  return 0;
}
```

Listing 2.1 An example of code resulting in a false positive detected by Wasm Doctor

```
==Wasm Doctor== Undefined value of size 2 bytes read from address 1049752.
==Wasm Doctor== validity: 0000000000000000
==Wasm Doctor== __original_main <- _start
```

Listing 2.2 An example of a false positive detected by Wasm Doctor

```
uint32_t x;

if (x << 32) {
  do_something();
}
```

Listing 2.3 Valgrind Memcheck false positive example

```
==Wasm Doctor== Undefined value of size 4 bytes read from address 1049752.
==Wasm Doctor== validity: 00000000000000000000000000000000
==Wasm Doctor== __original_main <- _start
```

Listing 2.4 An example of false positive detected by both Memcheck and Wasm Doctor

```c
#include <stdlib.h>
#include <string.h>

int
main(void)
{
  char orange[] = "orange";
  // allocated memory is too small by 1 byte
  char *orange_copy = (char *)malloc(strlen(orange));
  // strcpy writes outside of the allocated block
  strcpy(orange_copy, orange);

  free(orange_copy);
  return 0;
}
```

Listing 2.5 Example of a code which results in an invalid write error

```
==Wasm Doctor== Invalid write of size 1 bytes detected at address 1050278.
==Wasm Doctor== __stpcpy <- strcpy <- __original_main <- _start
```

Listing 2.6 Example of invalid write error detected by heap use validator

2.2.2.3 Heap Use Validator

Heap use validator focuses on the detection of memory leaks (functional requirement F4 1.3), double free (functional requirement F5 1.3), invalid free (functional requirement F6 1.3), use after free (functional requirement 3 1.3), and invalid read and write (functional requirement F7 1.3). The detection of invalid reads and writes is the most interesting of them, and it uses information about location of the linear stack, global data, and currently allocated heap blocks to determine whether the read or write is valid or not. If the address is inside one of the mentioned regions of memory, it is determined valid, otherwise it is reported as invalid. The information about allocated heap blocks is kept even after freeing them, and if there is a read or write inside them and there was no new heap block allocated, Wasm Doctor reports a use after free error.

Heap use validator provides a set of functions for the validation of the load and store instructions. The heap use validator store instruction validation function `check_write_validity()` is called from `doctor_store()` and takes the address of the store and the size of the stored value in bytes. When the `check_write_validity()` function is called, the corresponding region of the shadow memory is marked as valid. Then, when the `check_read_validity()` function is called, the validity of the corresponding shadow memory region is checked. If there is at least one invalid bit, the whole load is reported as invalid. The position and number of valid and invalid bits are later reported.

The way Wasm Doctor reports an invalid write error detected by the heap use validator in 2.5 can be seen in 2.6.

Wasm Doctor is capable of detecting and reporting `scanf()` buffer overflow in stack allocated buffers in certain cases. The detection depends on two main things. Firstly, it depends on the position of the buffer on the stack. Secondly, it is affected by the number of bytes written outside of the stack. This is something Valgrind is not able to detect when tested on the same code. The native program ends in a segmentation fault in case the amount of bytes written is too large.

```c
char name[10] = {};

scanf("Write your name:\n");

if(scanf("%s", name) != 1 || name[9]) {
  printf("Wrong input.\n");
  return;
}

printf("Your name is %s.\n", name);
```

Listing 2.7 Example of a stack allocated buffer overflow

An example of code that is susceptible to this overflow is 2.7. The overflow occurs when a string longer than 9 characters is given as input.

2.2.2.4 Linear Stack Validator

Linear stack validator is responsible for updating the information about the position of the base of the linear stack and the current linear stack pointer position. The information about linear stack base and linear stack pointer positions is used for invalid read and write detection. The linear stack validator is also responsible for the invalidation of a region of memory that is no longer a part of the linear stack when the stack size is decreased, in the case of Clang/LLVM C ABI the linear stack pointer address is increased, because the stack grows downwards [14].

An important caveat is that when a function is a leaf function, meaning that no other function will be called from it, the C ABI does not require the linear stack pointer to be moved and rather the function can use the top 128 bytes above the current stack pointer as if they were part of the stack. This region of linear memory is called the red zone. [14]

The omission of the red zone during definedness checking resulted in difficult-to-understand false positive during development.

2.2.2.5 Zero Address Access Validator

The zero address access validator is responsible for the detection of the address 0 accesses.

2.2.3 Reporter

The reporter provides a set of functions that make it easy to report an error and to pretty print the errors to standard output. It is used heavily during testing to determine what memory errors were detected. Also defined in error reporter are all of the errors detected by Wasm Doctor along with their specific attributes like address, size, location, etc. This can potentially be used by Trainer to display the errors outside of the standard output. Wasm Doctor currently reports the location in the form of a stack trace. The use of DWARF should be possible, but was determined outside of the scope of this thesis. Through the use of DWARF it could be possible to report the line number of the error and possibly show the error location interactively in the Trainer text editor.

2.2.4 Error Blacklist

Several library and WebAssembly functions contain code that Wasm Doctor interprets as having memory errors. These errors may be caused by imperfect implementation of Wasm Doctor or are detected correctly, but some (especially) library functions may do some things that would be otherwise considered errors. These errors are ignored in the output of Wasm Doctor as they may confuse the user. This blacklisting is done at the expense of possible false negatives.

A perfect blacklist was not a priority. It may be possible to further improve it by scanning the source code of wasi-libc for patterns that have already caused false positives.

2.2.4.1 Memory Allocators

A good example of such a function is malloc. It naturally reads from and writes to memory that has not been allocated nor is located in the region of the (linear) stack, or global data.

2.2.4.2 File System API

Functions from the file system API result in false positives. For example, error reports from functions `writev()` and `__fwritex()` are ignored. More file system API functions are expected to cause false positives, and their discovery is a matter of further experimentation and testing.

2.2.4.3 Memory Used for IO

Other type of functions that contain code that is improperly detected by Wasm Doctor as containing errors are functions utilizing memory for input and output.

This works by designating certain memory addresses for the input and output of data. This means that data read from certain address is expected to be defined by some outside process and writing data to other address is interpreted as the program output. [29]

2.2.5 Wasm State

The state of execution is stored in a `wasm_state` struct. It holds information like the current stack trace and the size in bytes of the currently analyzed load or store instruction. In this way, it is possible to not have to load this information about the state of execution and then immediately pass it into a reporting function.

Chapter 3
Implementation

In this chapter, the details of Wasm Doctor implementation and the implementation details of other technologies that affect it are described.

3.1 Implementation Language

The chosen language for implementation is C. There are multiple reasons for this. Firstly, it can be compiled to WebAssembly using a number of compilers and toolchains, for example, Clang, Emscripten, or the Zig compiler. Secondly, as the languages of choice for many WebAssembly interpreters are mainly C, C++, and Rust, C was chosen for its ability to be easily integrated into them.

3.2 Clang and LLVM

Wasm Doctor implementation uses Clang/LLVM C ABI for WebAssembly for its analysis. It is not the only possible WebAssembly C ABI and thus the analysis is Clang/LLVM specific. [14]

3.2.1 Clang and LLVM Implementation Specifics

Clang, specifically version 15.0.7, is the compiler used by Trainer to compile student programs. The programs compiled by Clang are generated with additional systems that modify the usual way in which WebAssembly programs are executed.

3.2.2 Clang's Linear Stack

The `$__stack_pointer` WebAssembly global variable is used in programs compiled by Clang. It holds an address of the top of the linear stack. This stack is not an actual call stack, but rather a representation of the call stack available in linear memory. Clang imitates the use of stack frames and stores values of local variables (i.e. local variables in a C program, not WebAssembly local variables) in linear memory. This way of storing local variables is not necessary and is the way it is defined by the WebAssembly C ABI. The compiler could use the WebAssembly local variables for this purpose as they are similar to registers in other architectures, but unlike those, they are not limited in quantity. [14]

The way linear memory of C programs compiled by Clang to WebAssembly looks is shown in figure 3.1.

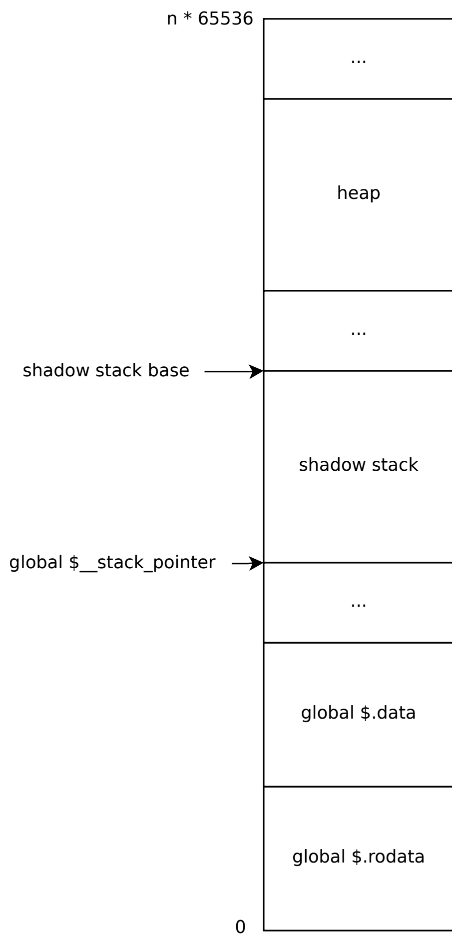


Figure 3.1 WebAssembly linear memory when compiled by Clang (65536 is the page size [30])

The WebAssembly memory debugger must take into account these changes in WebAssembly program execution. The changes mainly affect the invalidation of parts of linear memory dedicated to the linear stack. The Wasm Doctor library can handle both programs that use Clang's linear stack and ones that do not. The decision to utilize the Clang specific functions provided by the Wasm Doctor library has to be made in the implementation of the program that uses this library. In case of this thesis, in the interpreter fork.

3.3 Instrumentation and Interpretation Comparison

Implementation of both instrumenter and interpreter was outside of the scope of this thesis. Implementation of an instrumenter was also not considered because there exists a WebAssembly instrumentation framework Wasabi.

One of Valgrind's strengths is its relatively small slow-down compared to alternatives. Some memory debugging tools are faster, but do not offer the same level of memory bug detection. [5]

Speed is not a priority for the resulting WebAssembly memory debugger. This allows the resulting memory debugger to be implemented using an extension of an existing WebAssembly interpreter. The option of using interpretation is highly advantageous from the point of view of implementation. It not only dramatically reduces the amount and complexity of the implementation, which would be high even for a prototype, but it also allows the access to runtime and WebAssembly module information through the interpreter structures. It also allows this thesis to focus more on the use of shadow memory and other methods used for memory error detection by Valgrind and Memcheck and less on the implementation details of Valgrind as a framework for building heavyweight DBA tools.

Traditionally, instrumentation in a classical debugger can utilize functions provided by the OS. This may be possible through WASI, but when tried on a basic example of a debugger-like code there were some functions not implemented in it. This topic was kept in mind during the analysis phase because Trainer would benefit from a classical debugger. As memory debuggers are not very similar to classical debuggers, further research is needed on the possibilities of a classical debugger for WebAssembly.

3.4 Interpreter Choice

The choice of an ideal interpreter was based on two criteria. Firstly, the ease with which it can be extended with the Wasm Doctor library. Secondly, the ability to compile it to WebAssembly.

Another option for the implementation of the WebAssembly memory debugger prototype is Wasabi, a dynamic analysis framework for WebAssembly programs.

Toywasm is the interpreter chosen for the implementation of Wasm Doctor. Its self-hosting capability has several advantages. Firstly, it makes it possible to be compiled to WebAssembly and used by Trainer for execution of student code. This is something Wasabi, although through different mechanisms is able to provide. Wasabi is an instrumentation framework, and it makes it possible to use the browser's WebAssembly runtime. This would result in a very small amount of overhead during execution, especially when compared to running an interpreter inside the browser, as then the interpretation is effectively running twice (WebAssembly being interpreted by WebAssembly). Wasabi's reliance on JavaScript might make it difficult to use as a standalone tool. Interpreters already provide the ability to use them as standalone tools; moreover, some of them offer the ability to be compiled to WebAssembly, which also makes them very portable.

The use of an interpreter as a way of running the analysis code also has the advantage that the analysis software can be created as a library. This makes it easy to switch to another, for example, more performant interpreter, with minimal development effort. The fact that the analysis part of the resulting memory debugger is located inside Wasm Doctor library means that it would be possible to use call the library functions from Wasabi hooks.

3.4.1 WABT wasm-interp

WABT contains its own WebAssembly interpreter called wasm-interp. It is written in C++. [12]

This interpreter was considered, but its architecture is not as clear as the architecture of Toywasm.

3.4.2 Wasm3

Wasm3 is a fast WebAssembly interpreter. It is self-hosting and written in C. The interpreter strategy used is called M3. It is in minimal maintenance phase. [31]

Due to its more complex nature and the fact that it is in minimal maintenance phase it was not chosen as the interpreter used by Wasm Doctor.

3.4.3 WebAssembly spec interpreter

The WebAssembly spec interpreter is written in OCaml. It follows the specification closely and is declarative. [32]

The fact that it follows the specification so closely is an advantage, but as the author of this thesis does not have experience with OCaml it was not selected.

3.4.4 Toywasm

The interpreter of choice for Wasm Doctor is Toywasm. Toywasm is a WebAssembly interpreter written in C. One of the goals of Toywasm's author is to learn the specification by implementing it [33]. The author of this thesis found it to be the easiest to understand of all the interpreters considered. Easily understandable code is important because the interpreter works as the runner of analysis code. As it is relatively new, it already supports many proposals.

One important differentiating factor from other interpreters is that it is self-hosting, i.e. the ability to run itself, and it makes it immediately apparent as one of its build targets is WebAssembly. This is important because Wasm Doctor is supposed to run in the students' browsers and thus has to be able to run in a WebAssembly runtime.

3.5 Valgrind IR

Valgrind works as a framework for the creation of heavyweight DBA tools. To make the creation and maintenance of such tools easier, it utilizes an IR (Intermediate Representation). The need for an IR is facilitated by the desire to separate the implementation of DBA tools from the target-specific implementation. [5]

Wasm Doctor does not use an IR because it is not necessary. As Wasm Doctor is supposed to be WebAssembly specific, there is no need to separate the implementation of the DBA specific code and the target specific code. WebAssembly is also a simple virtual ISA, compared to the ISAs targeted by Valgrind. It is already somewhat similar to the Valgrind's IR.

The porting of Valgrind to a new architecture was deemed outside the scope of this thesis during the analysis. For example, it would require writing new code for the JIT compiler. [5] The advantage of this approach is that it would enable the use of not only Memcheck, but all the tools that Valgrind offers. However, it is not certain that such porting is even possible because of the differences in the ISAs.

```
  const struct module *m = MODULE;
  struct memarg memarg;
  int ret;
  LOAD_PC;
  READ_MEMARG(&memarg);
  uint32_t offset = memarg.offset;
  POP_VAL(TYPE_i32, i);
  struct val val_c; // A

  void *datap;
  ret = memory_getptr(ECTX, memarg.memidx, val_i.u.i32,
                      memarg.offset, MEM / 8, &datap);
  if (ret != 0) {
    goto fail;
  }
  val_c.u.i##STACK = CAST le##MEM##_decode(datap); // B

  doctor_load(val_i.u.i32 + offset, MEM / 8); // D

  PUSH_VAL(TYPE_##I_OR_F##STACK, c); // C
  SAVE_PC;
  INSN_SUCCESS;
fail:
  INSN_FAIL;
```

Listing 3.1 Simplified Toywasm load implementation

3.5.1 Wasm Doctor Library Initialization

Wasm Doctor Library has to receive some information before the start of execution of a WebAssembly program. Linear stack base address, offsets (starting addresses), sizes of global data regions, and size of memory all need to be known.

3.5.2 Load and Store

The store and load instructions are implemented as a C macro (3.1). To understand this macro, it is important to understand the way the macros `POP_VAL` and `PUSH_VAL` work. They take the variable name as the second argument. For example, when `POP_VAL(TYPE_i32, i)`, a new variable is declared and defined. Its name is based on the input of `POP_VAL`, but it is prefixed with `val_` so the value popped from the operand stack is put in `struct val val_i`. The `PUSH_VAL` macro works in an opposite direction, and the `struct val val_c` on line marked A is set on line marked B and then is pushed onto the operand stack on line marked C. The integer value of `struct val` is accessed through a union of integer and floating point types. In this case, a 32-bit integer is accessed.

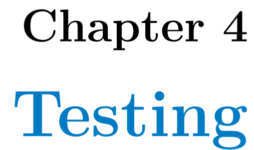
Wasm Doctor registers the load on line marked D. The variable `val_i` contains the address that is loaded from. WebAssembly load instructions can be extended with the value `offset` that is added to the address. `MEM` contains the size in bits of the value that is loaded. Store instruction is implemented and registered similarly.

3.6 Doug Lea's Memory Allocator (dlmalloc)

Wasm Doctor can track malloc, specifically `dlmalloc()`, and `free()` calls to reason about the use of heap-allocated memory and possibly report heap use bugs. The tracking of `dlmalloc()` makes it possible to easily track all variants of malloc such as `malloc()`, `realloc()`, `calloc()`, etc.

3.7 Wasm Doctor Library Build System

Wasm Doctor library uses make as its build system. It is distributed as libwasmdoctor.a, a statically linked library.



Chapter 4 Testing

In this chapter, the way Wasm Doctor and Wasm Doctor library are tested is presented.

4.1 Wasm Doctor Testing

Wasm Doctor, which means the fork of Toywasm, is manually tested on examples of C programs compiled to WebAssembly and available in the `examples` directory inside the Wasm Doctor library. There is a mixture of simple programs that focus on reproducing errors in a simple way and programs that were created based on the errors commonly done by students as reported by Ing. Jan Matoušek.

4.2 Wasm Doctor Library Testing

Wasm Doctor library is tested using automated tests in the `test` directory. They utilize the `error_reporter` structure that accumulates all detected errors along with useful information about their location and parameters. In 4.1 is an example of a test of double free detection.

```c
void
test_double_free(void)
{
  struct wasm_doctor doctor;
  doctor_init(&doctor, 2, false);

  doctor_frame_enter(10, "test_function");

  assert(doctor.reporter.double_free_errors_size == 0);

  size_t address = 42;

  doctor_register_malloc(address, 100);
  doctor_register_free(address);

  assert(doctor.reporter.double_free_errors_size == 0);

  doctor_register_free(address);

  assert(doctor.reporter.double_free_errors_size == 1);
  assert(doctor.reporter
    .double_free_errors[doctor.reporter.double_free_errors_size - 1]
    .address == address);

  doctor_frame_exit();

  doctor_exit(false);
  doctor_reporter_exit();

  printf("[OK] double free test\n");
}
```

Listing 4.1 An example of test of double free error detection.

Chapter 5
Discussion

A comparison of Memcheck and Wasm Doctor is in table 5.1. It is important to point out, that Wasm Doctor has a higher number of false positives detected.

Memory Error Detection Support	Memcheck	Wasm Doctor
Undefined Memory Use	yes	yes[1]
Undefined Local Variable Use	not applicable[2]	yes[3]
Use After Free	yes	yes
Memory Leak	yes	yes
Double Free	yes	yes
Invalid Free	yes	yes
Invalid Read and Write	yes	yes
Null Pointer	yes	yes

Table 5.1 Comparison of Memcheck and Wasm Doctor capabilities

5.1 Possible Improvements

The presented memory debugger is fairly feature-complete. Its definedness checking could be improved as described in 2.2.2.1. This should result in the reduction of false positives. Further finetuning of the blacklist may also be necessary for production use in Trainer. Another future improvement could be optimization of the analysis methods. Switch to a faster interpreter or instrumentation framework could also be beneficial and should be possible thanks to the Wasm Doctor library.

5.2 Future Work

Wasm Doctor is built in a way to make it possible to use it in Trainer. The proposed method of integration is to use the Wasm Doctor WebAssembly binary inside Trainer for the interpretation of WebAssembly compiled student's source codes. The way to present the detected errors on the Trainer side also needs to be decided.

[1] Wasm Doctor's definedness checking is limited in comparison with Memcheck's.
[2] As Memcheck (Valgrind) does not target WebAssembly it does not check undefined local variable use.
[3] In the current prototype reporting of this error is disabled.

Conclusion

The goal of creating a prototype of a WebAssembly debugger with similar functionality to Valgrind's Memcheck was successful. Valgrind, Valgrind's Memcheck, memory errors, and WebAssembly were also successfully analyzed. The prototype creation required a combination of knowledge from seemingly different fields. As it turned out WebAssembly is not, despite its name, very web-specific and can be viewed rather as a new target ISA for a memory debugger. The fact that WebAssembly is a Harvard architecture and a stack machine along with the fact that the Clang-compiled code has a unique WebAssembly C ABI made the initial analysis difficult and consumed a substantial portion of the time dedicated for this thesis.

Wasm Doctor can successfully detect many common memory errors without false positives in less complicated source code. The main problems in error detection are currently false positives. They primarily come from library functions, which are usually highly optimized or work in an unusual manner.

Wasm Doctor should be usable in Trainer with some small improvements. The exact improvements necessary depend on further research. Possible improvement paths that could be researched are improvement of the error blacklist and the switch of the undefined value use detection strategy to the one used in Memcheck. Another possible improvement is the ability to report the line numbers of detected errors. This will require further research on DWARF and DWARF for WebAssembly.

Appendix A

Wasm Doctor Analysis Examples

The current Wasm Doctor's capabilities are presented in the following examples.

A.1 Example 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char orange[] = "orange";
  // allocated memory is too small by 1 byte
  char *orange_copy = (char *)malloc(strlen(orange));
  // the content of the string is copied without the terminating 0
  for (int i = 0; i < strlen(orange); ++i) {
    orange_copy[i] = orange[i];
  }
  // string that is not null terminated is printed
  printf("%s\n", orange_copy);

  free(orange_copy);
  return 0;
}
```

```
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1053862.
==Wasm Doctor== memchr <- strnlen <- printf_core <- vfprintf <- printf
<- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Undefined value of size 1 bytes read from address 1053862.
==Wasm Doctor== validity: 00000000
==Wasm Doctor== memchr <- strnlen <- printf_core <- vfprintf <- printf
<- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1053862.
==Wasm Doctor== printf_core <- vfprintf <- printf <- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Undefined value of size 1 bytes read from address 1053862.
==Wasm Doctor== validity: 00000000
==Wasm Doctor== printf_core <- vfprintf <- printf <- __original_main <- _start
==Wasm Doctor==
orange
```

A.2 Example 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char orange[] = "orange";
  // allocated memory has the correct size
  char *orange_copy = (char *)malloc(strlen(orange) + 1);
  // the content of the string is copied with the terminating 0
  for (int i = 0; i < strlen(orange) + 1; ++i) {
    orange_copy[i] = orange[i];
  }
  // string that is null terminated is printed
  printf("%s\n", orange_copy);

  free(orange_copy);
  return 0;
}
```

```
orange
```

A.3 Example 3

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
  char orange[] = "orange";
  // allocated memory has the correct size
  char *orange_copy = (char *)malloc(strlen(orange) + 1);
  // the content of the string is copied without the terminating 0
  for (int i = 0; i < strlen(orange); ++i) {
    orange_copy[i] = orange[i];
  }
  // string that is null terminated is printed
  printf("%s\n", orange_copy);

  free(orange_copy);
  return 0;
}
```

```
==Wasm Doctor== Undefined value of size 1 bytes read from address 1053862.
==Wasm Doctor== validity: 00000000
==Wasm Doctor== memchr <- strnlen <- printf_core <- vfprintf <- printf
<- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Undefined value of size 1 bytes read from address 1053862.
==Wasm Doctor== validity: 00000000
==Wasm Doctor== printf_core <- vfprintf <- printf <- __original_main <- _start
==Wasm Doctor==
orange
```

A.4 Example 4

```c
#include <stdlib.h>

int main() {
  int number_of_names = 4;

  char **names = (char **)malloc(number_of_names * sizeof(*names));

  names[0] = (char *)malloc(5 * sizeof(**names));
  names[1] = (char *)malloc(8 * sizeof(**names));
  names[2] = (char *)malloc(7 * sizeof(**names));
  names[3] = (char *)malloc(5 * sizeof(**names));

  for (int i = 0; i < number_of_names; ++i) {
    free(names[i]);
  }

  return 0;
}
```

```
==Wasm Doctor== Memory leak of size 16 bytes detected at address 8402176.
```

A.5 Example 5

```c
#include <stdlib.h>
#include <string.h>

int main() {
  char *name = (char *)malloc(4 * sizeof(*name));

  strcpy(name, "John");

  free(name);

  return 0;
}
```

```
==Wasm Doctor== Invalid write of size 1 bytes detected at address 1050276.
==Wasm Doctor== __stpcpy <- strcpy <- __original_main <- _start
==Wasm Doctor==
```

A.6 Example 6

```c
#include <stdlib.h>

int main() {
  int *numbers = (int *)malloc(10 * sizeof(*numbers));

  free(numbers);
  free(numbers);

  return 0;
}
```

```
==Wasm Doctor== Double free detected at address 8402176.
==Wasm Doctor== free <- __original_main <- _start
==Wasm Doctor==
Error: [trap] out of bounds memory access (3):
invalid memory access at 0000 55455328 + 00000024, size 4, meminst size 17
frame[  3] funcpc 0020d5 (<unknown>:dlfree) callerpc 0020cf
  param [0] = 0000f930
  local [1] = 55455328
  local [2] = 00100698
  local [3] = 0000f931
  local [4] = 0010ffc8
  local [5] = 55455328
  local [6] = 00000000
  local [7] = 00000000
  local [8] = 00000000
  local [9] = 00000000
frame[  2] funcpc 0020c7 (<unknown>:free) callerpc 000201
  param [0] = 001006a0
frame[  1] funcpc 0001a8 (<unknown>:__original_main) callerpc 000189
  local [0] = 00100690
  local [1] = 00000010
  local [2] = 00100680
  local [3] = 00000000
  local [4] = 00000028
  local [5] = 001006a0
  local [6] = 001006a0
  local [7] = 001006a0
  local [8] = 00000000
  local [9] = 00000000
  local [10] = 00000000
frame[  0] funcpc 000155 (<unknown>:_start)
  local [0] = 00000000
```

A.7 Example 7

```c
#include <stdlib.h>

int main() {
  int number_of_names = 4;

  char **names = (char **)malloc(number_of_names * sizeof(*names));

  names[0] = (char *)malloc(5 * sizeof(**names));
  names[1] = (char *)malloc(8 * sizeof(**names));
  names[2] = (char *)malloc(7 * sizeof(**names));
  names[3] = (char *)malloc(5 * sizeof(**names));

  // i is incremented upto 4
  for (int i = 0; i <= number_of_names; ++i) {
    free(names[i]);
  }

  return 0;
}
```

```
 ==Wasm Doctor== Invalid read of size 4 bytes detected at address 1050288.
==Wasm Doctor== __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Undefined value of size 4 bytes read from address 1050288.
==Wasm Doctor== validity: 00000000000000000000000000000000
==Wasm Doctor== __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid free detected at address 0.
==Wasm Doctor== free <- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Memory leak of size 16 bytes detected at address 8402176.
```

A.8 Example 8

The following examples were provided by Ing. Jan Matoušek. They are supposed to sort words, given to them as input. They usually contain multiple errors.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct word_t {
  char data[10];
} word_t;

int cmp(const void * a, const void * b) {
  return strcmp((const char *)a, (const char *)b);
}

int main() {
  size_t capacity = 10;
  size_t count = 0;
  word_t * words = (word_t *)malloc(sizeof(word_t) * capacity);
  while(scanf("%s", words[count++].data) != EOF) {
    if(count == capacity) {
      capacity += capacity + 10;
      words = (word_t *)realloc(words, sizeof(word_t) * capacity);
    }
  }
  count--;
  qsort(words, count, sizeof(word_t), cmp);
  for(size_t i = 0; i < count; i++) {
    printf("%s\n", words[i].data);
  }
  free(words);
  return 0;
}
```

Wasm Doctor correctly does not detect any errors for less than ten words. If the number of words submitted is greater than ten, `realloc()` is called and a number of (probably) false positives is detected. Memcheck does not report any errors for both inputs. The exact cause of the false positives was not yet determined.

```
hatch
bread
second
bow
ambitious
ambitious
bow
bread
hatch
second
```

```
hatch
bread
second
bow
ambitious
material
fan
drink
prepare
ignore
summon
==Wasm Doctor== Invalid write of size 1 bytes detected at address 1055604.
==Wasm Doctor== vfscanf <- vscanf <- scanf <- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid write of size 1 bytes detected at address 1055605.
==Wasm Doctor== vfscanf <- vscanf <- scanf <- __original_main <- _start
...
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055605.
==Wasm Doctor== strcmp <- cmp(void const*, void const*) <- wrapper_cmp
<- trinkle <- __qsort_r <- qsort <- __original_main <- _start
==Wasm Doctor==
ambitious
bow
bread
drink
fan
hatch
ignore
material
prepare
second
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055604.
==Wasm Doctor== memchr <- strnlen <- printf_core <- vfprintf <- printf
<- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055605.
==Wasm Doctor== memchr <- strnlen <- printf_core <- vfprintf <- printf
<- __original_main <- _start
==Wasm Doctor==
...
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055604.
==Wasm Doctor== __fwritex <- printf_core <- vfprintf <- printf
<- __original_main <- _start
==Wasm Doctor==
...
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055609.
==Wasm Doctor== memcpy <- __fwritex <- printf_core <- vfprintf
<- printf <- __original_main <- _start
==Wasm Doctor==
summon
```

A.9 Example 9

In the following example Wasm Doctor correctly detects errors caused by wrong memory allocation.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct word_t {
  char data[10];
} word_t;

int cmp(const void * a, const void * b) {
  return strcmp((const char *)a, (const char *)b);
}

int main() {
  size_t capacity = 32;
  size_t count = 0;
  word_t * words = (word_t *)malloc(capacity);
  while(scanf("%s", words[count++].data) != EOF) {
    if(count == capacity) {
      capacity += capacity + 10;
      words = (word_t *)realloc(words, capacity);
    }
  }
  count--;
  qsort(words, count, sizeof(word_t), cmp);
  for(size_t i = 0; i < count; i++) {
    printf("%s\n", words[i].data);
  }
  free(words);
  return 0;
}
```

```
one
two
three
four
==Wasm Doctor== Invalid write of size 1 bytes detected at address 1055536.
==Wasm Doctor== vfscanf <- vscanf <- scanf <- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid write of size 1 bytes detected at address 1055537.
==Wasm Doctor== vfscanf <- vscanf <- scanf <- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid write of size 1 bytes detected at address 1055538.
==Wasm Doctor== vfscanf <- vscanf <- scanf <- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid read of size 8 bytes detected at address 1055536.
==Wasm Doctor== memcpy <- trinkle <- __qsort_r <- qsort
<- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid write of size 8 bytes detected at address 1055536.
==Wasm Doctor== memcpy <- trinkle <- __qsort_r <- qsort
<- __original_main <- _start
==Wasm Doctor==
four
one
three
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055536.
==Wasm Doctor== memchr <- strnlen <- printf_core <- vfprintf
<- printf <- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055537.
==Wasm Doctor== memchr <- strnlen <- printf_core <- vfprintf
<- printf <- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055537.
==Wasm Doctor== printf_core <- vfprintf <- printf <- __original_main
<- _start
==Wasm Doctor==
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055536.
==Wasm Doctor== __fwritex <- printf_core <- vfprintf <- printf
<- __original_main <- _start
==Wasm Doctor==
==Wasm Doctor== Invalid read of size 1 bytes detected at address 1055536.
==Wasm Doctor== memcpy <- __fwritex <- printf_core <- vfprintf
<- printf <- __original_main <- _start
==Wasm Doctor==
two
```

Appendix B
Manual

B.1 Local

To build Wasm Doctor (Toywasm fork), it is necessary to build the Wasm Doctor library first. The steps to build Wasm Doctor are as follows:

1. Run `make` inside the `wasm_doctor_library` directory
2. Run `mkdir build` inside the `wasm_doctor` directory
3. Run `cmake -DTOYWASM_ENABLE_WASI=ON ..` inside the `wasm_doctor/build` directory
4. Run `cmake --build .` inside the `wasm_doctor/build` directory

To run analysis of a WebAssembly binary run
`wasm_doctor/build/toywasm --wasi wasm_doctor_library/examples/invalid_free.wasm`

B.2 WebAssembly Binary

Wasm Doctor has the ability to run inside a WebAssembly runtime. To use the Wasm Doctor WebAssembly binary run:

```
toywasm/build/toywasm --wasi --wasi-dir . -- \
wasm_doctor.wasm --wasi wasm_doctor_library/examples/invalid_free.wasm
```

In this case the `wasm_doctor.wasm` WebAssembly binary needs to be itself run by some WebAssembly runtime with WASI support, for example the Toywasm interpreter. It is necessary to build the Toywasm interpreter with WASI support:

1. Run `mkdir build` inside the `toywasm` directory
2. Run `cmake -DTOYWASM_ENABLE_WASI=ON ..` inside the `toywasm/build` directory
3. Run `cmake --build .` inside the `toywasm/build` directory

B.3 Compilation to WebAssembly

During development of Wasm Doctor "trainer_compiler" (wasm-experiment) was primarily used for the compilation of example source codes due to ease of use. To use it for compilation:

1. Run `npm install` inside the `trainer_compiler` directory
2. Run `npm run serve` inside the `trainer_compiler` directory
3. Trainer compiler should be available at `localhost` (usually at port `8000`)

Compilation of the source codes to WebAssembly is also possible with `wasi-sdk`. Information about its use is available at `https://github.com/WebAssembly/wasi-sdk`. Instead of using `libc` provided by `wasi-sdk` it is necessary to provide `libc` from `https://gitlab.fit.cvut.cz/trainer-fit/wasm-artifacts`.

To compile an example C source code, the following command can be used:

```
/opt/wasi-sdk/bin/clang -fno-builtin \
-isysroot=/some-path/wasm-artifacts/wasi-libc/sysroot example.c
```

Bibliography

1. SEWARD, Julian; NETHERCOTE, Nicholas. *Using Valgrind to detect undefined value errors with bit-precision* [online]. 2005. [visited on 2024-04-20]. Available from: `https://valgrind.org/docs/memcheck2005.pdf`.
2. OWASP FOUNDATION. *Memory leak* [online]. 2024. [visited on 2024-05-02]. Available from: `https://owasp.org/www-community/vulnerabilities/Memory_leak`.
3. OWASP FOUNDATION. *Doubly freeing memory* [online]. 2024. [visited on 2024-05-02]. Available from: `https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory`.
4. OWASP FOUNDATION. *Memory leak* [online]. 2024. [visited on 2024-05-02]. Available from: `https://owasp.org/www-community/vulnerabilities/Null_Dereference`.
5. NETHERCOTE, Nicholas; SEWARD, Julian. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation* [online]. 2007. [visited on 2024-04-20]. Available from: `https://valgrind.org/docs/valgrind2007.pdf`.
6. HYKES, Solomon [online]. 2019. [visited on 2024-04-20]. Available from: `https://twitter.com/solomonstre/status/1111004913222324225`.
7. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly 2.0 (Draft 2024-04-12)* [online]. 2024. [visited on 2024-04-20]. Available from: `https://webassembly.github.io/spec/core/intro/introduction.html`.
8. AKINYEMI, Stephen. *Awesome WebAssembly Languages* [online]. 2024. [visited on 2024-04-20]. Available from: `https://github.com/appcypher/awesome-wasm-langs`.
9. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly 2.0 (Draft 2024-04-12)* [online]. 2024. [visited on 2024-04-20]. Available from: `https://webassembly.github.io/spec/core/text/instructions.html`.
10. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly 2.0 (Draft 2024-04-12)* [online]. 2024. [visited on 2024-04-20]. Available from: `https://webassembly.github.io/spec/core/text/conventions.html`.
11. MOZILLA. *store: Wasm text instruction* [online]. 2024. [visited on 2024-05-11]. Available from: `https://developer.mozilla.org/en-US/docs/WebAssembly/Reference/Memory/Store`.
12. WEBASSEMBLY COMMUNITY GROUP. *WABT: The WebAssembly Binary Toolkit* [online]. 2023. [visited on 2024-05-12]. Available from: `https://github.com/WebAssembly/wabt`.
13. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly* [online]. 2024. [visited on 2024-04-20]. Available from: `https://webassembly.org`.

14. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly Tool Conventions* [online]. 2024. [visited on 2024-05-02]. Available from: https://github.com/WebAssembly/tool-conventions/blob/main/BasicCABI.md.
15. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly 2.0 (Draft 2024-04-12)* [online]. 2024. [visited on 2024-04-20]. Available from: https://webassembly.github.io/spec/core/syntax/modules.html.
16. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly 2.0 (Draft 2024-04-12)* [online]. 2024. [visited on 2024-04-20]. Available from: https://webassembly.github.io/spec/core/syntax/instructions.html#syntax-memarg.
17. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly 2.0 (Draft 2024-04-12)* [online]. 2024. [visited on 2024-04-20]. Available from: https://webassembly.github.io/spec/core/exec/modules.html.
18. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly 2.0 (Draft 2024-04-12)* [online]. 2024. [visited on 2024-04-20]. Available from: https://webassembly.github.io/spec/core/exec/modules.html#memories.
19. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly System Interface* [online]. 2023. [visited on 2024-05-12]. Available from: https://github.com/WebAssembly/WASI.
20. WEBASSEMBLY COMMUNITY GROUP. *Legacy WASI docs* [online]. 2023. [visited on 2024-05-12]. Available from: https://github.com/WebAssembly/WASI/blob/main/legacy/README.md.
21. WEBASSEMBLY COMMUNITY GROUP. *WASI Preview 2* [online]. 2024. [visited on 2024-05-16]. Available from: https://github.com/WebAssembly/WASI/blob/main/preview2/README.md.
22. WEBASSEMBLY COMMUNITY GROUP. *wasi-libc* [online]. 2023. [visited on 2024-05-12]. Available from: https://github.com/WebAssembly/wasi-libc.
23. ILAS, Filaret. *Using Code Instrumentation for Debugging and Constraint Checking* [online]. 2009. [visited on 2024-05-16]. Available from: https://rex.libraries.wsu.edu/esploro/outputs/99900525162201842#file-0.
24. AHO, Alfred V.; LAM, Monica S.; SETHI, Ravi; ULLMAN, Jeffrey D. *Compilers: Principles, techniques, and tools*. Pearson Higher Ed, 2013.
25. LEHMANN, Daniel; PRADEL, Michael. *Wasabi: A Framework for Dynamically Analyzing WebAssembly* [online]. 2019. [visited on 2024-05-02]. Available from: https://software-lab.org/publications/asplos2019_Wasabi.pdf.
26. DWARF DEBUGGING INFORMATION FORMAT COMMITTEE. *DWARF Debugging Information Format* [online]. 2023. [visited on 2024-05-12]. Available from: https://dwarfstd.org/.
27. EAGER, Michael J. *Introduction to the DWARF Debugging Format* [online]. 2012. [visited on 2024-05-12]. Available from: https://dwarfstd.org/doc/Debugging%5C%20using%5C%20DWARF-2012.pdf.
28. DELENDIK, Yury. *DWARF for WebAssembly* [online]. 2020. [visited on 2024-05-12]. Available from: https://yurydelendik.github.io/webassembly-dwarf/.
29. MALMGREN, Peter. *Getting data in and out of WASI modules* [online]. 2022. [visited on 2024-05-16]. Available from: https://petermalmgren.com/serverside-wasm-data/.
30. WEBASSEMBLY COMMUNITY GROUP. *WebAssembly 2.0 (Draft 2024-04-28)* [online]. 2024. [visited on 2024-05-16]. Available from: https://webassembly.github.io/spec/core/exec/runtime.html#page-size.

31. SHYMANSKYY, Volodymyr. *Wasm3* [online]. 2023. [visited on 2024-05-12]. Available from: `https://github.com/wasm3/wasm3`.
32. WEBASSEMBLY COMMUNITY GROUP. *spec* [online]. 2023. [visited on 2024-05-12]. Available from: `https://github.com/WebAssembly/spec/`.
33. YAMAMOTO, Takashi. *yamt/toywasm: A WebAssembly interpreter written in C* [online]. 2024. [visited on 2024-05-12]. Available from: `https://github.com/yamt/toywasm`.

Contents of the Attachment

```
src
├── implementation ........ implementation part of the thesis
│   ├── toywasm ........ WebAssembly interpreter
│   ├── trainer_compiler ........ compiler used for compilation of examples
│   ├── wasm_doctor ........ WebAssembly memory debugger (Toywasm fork)
│   ├── wasm_doctor.wasm ........ Wasm Doctor WebAssembly binary
│   └── wasm_doctor_library ........ Wasm Doctor library used in Wasm Doctor
└── thesis ........ source code of the thesis in LaTeX
text
└── thesis.pdf ........ text of the thesis in PDF
```