Emulating Memory Corruption Attacks

Bachelor Thesis

Malte Laukötter

September 2019

Supervisor:

Prof. Dr. Dieter Gollmann

M.Sc. A.-C. Kycler

Hamburg University of Technology

Security in Distributed Applications

https://www.tuhh.de/sva

Am Schwarzenberg-Campus 3

21073 Hamburg

Germany

Declaration

I, Malte Laukötter, solemnly declare that I have written this bachelor thesis independently, and that I have not made use of any aid other than those acknowledged in this bachelor thesis. Neither this bachelor thesis, nor any other similar work, has been previously submitted to any examination board.

Hamburg, September 19, 2019

Malte Laukötter

Abstract

The paper "Weird machines, exploitability, and provable unexploitability" by Thomas Dullien [1] proposes a way to formally prove that a program is not exploitable by introducing the notion of an intended finite-state machine.
This thesis describes the creation of an emulator of the machine model described in this paper. Furthermore, it shows some mistakes in the implementations of the example program provided in the paper, explains how these were fixed and then shows that the described attack for one of the implementations can be executed in the emulator using the fixed implementation of the program.

Contents

Abstract iii

1. Introduction 1

1.1. Weird Machines and Memory Corruption Attacks 1
1.2. Motivation 2
1.3. Overview of this Thesis 2
1.4. Related Work 2

2. The Machine Model 3

2.1. Operators 3

3. The Programming Language 5

3.1. Basic Structure 5
3.2. Mathematical Expressions 6
3.3. Independence of Lines 6

4. Design Decisions for the Implementation of the Emulator 7

4.1. Requirements on the Emulator 7
4.2. CLI 8
4.3. Programming Language 8
4.4. Development Tools 8
4.5. Lexing and Parsing of the Source Code 9
4.6. Implementation of the Machine 10
4.6.1. State Object 11
4.6.2. Memory 11
4.6.3. Operators 12
4.7. Configuring the Emulator 14
4.8. Storing and Loading Executions 14

5. The Example Programs 17

5.1. Fixing *variant2A.s* 17
5.1.1. Wrong Usage of the `J` Operator 17
5.1.2. Typo in the Last Line 18
5.1.3. Initialization of Linked Lists Not Working 18

5.1.4. Check for Matching Stored Keys Not Working . . . 19
5.1.5. Unlink of Read Entries Not Working . . . 19
5.2. Implementing *variant2A.s* as Described in the Paper . . . 21
5.3. Attack on *variant2A.s* . . . 21
5.4. A Simpler Attack on *variant2A.s* . . . 22
5.5. Attack on *variant2B.s* . . . 22

6. Evaluation **23**
6.1. Additional Requirements on the Emulator . . . 23
6.2. Design Decisions . . . 24

7. Conclusion **27**

A. Action Files **29**

B. Configuration Files **33**

C. Implementations of the Example Program **35**
C.1. variant1A.s . . . 35
C.2. variant2A.s . . . 36
C.3. variant2A_fixed.s . . . 37
C.4. variant2B.s . . . 39

List of Abbreviations **41**

Bibliography **43**

1. Introduction

The concept of exploitability has been formalized by Thomas Dullien in his paper "Weird machines, exploitability, and provable unexploitability" [1].

1.1. Weird Machines and Memory Corruption Attacks

For this, he introduces the notion of an Intended Finite-State Machine (IFSM). This IFSM is a model for the software that will run on a general-purpose machine. Any difference in the working of the software on the general-purpose machine and the IFSM can then be defined as a bug. As not every bug in software leads to security problems he further introduces security properties for the IFSM that then need to hold in the emulation of the IFSM on the general-purpose machine. As an IFSM is a theoretical construct and needs an implementation that emulates it to run on a general-purpose machine, he furthermore introduces a computing environment.

For a more formal definition of a bug, he further introduces weird states. For this, he defines all states of the implementation of the software on a real-world general-purpose machine to be in one of three categories. Every state is either a state of the IFSM, a state in a transition between two states of the IFSM or a weird state. Once one of these weird states is reached during emulation of the IFSM on the real-world general-purpose machine a new machine emerges that transforms between different weird states using the transactions that were meant for the IFSM. This leads first of all to the property that weird machines have unknown state spaces. Due to this, it becomes difficult to determine if the security properties still hold. Furthermore, the computational power of a weird machine is.

Following this definition of a weird machine Dullien then defines an exploit as a program for the weird machine that violates a security property of the IFSM and exploitation as choosing a state that is not weird, entering a weird state and then programming the weird machine with an exploit.

To further his discussion and to answer if multiple implementations of the same IFSM can fulfill different security properties he then introduces an attacker model and two different implementations of a program with a secret-value store. The model is called "Arbitrary program-point, chosen-bitflip, registers" and allows the attacker to stop the execution of the program at any time and then chose to flip an arbitrary bit that is not stored in a register.

This attacker model models an attacker that tries to do a memory corruption attack. Memory corruption attacks are one of the oldest problems in computer science and no general solution to protect against these is currently known [2]. These attacks use programming bugs or other means to modify pointers in the program to reach and possibly modify parts of the memory that should not be accessible to the attack and thus violating memory safety. Such a modification can also happen without any mistake by the programmer, for example, due to cosmic rays [3]. The attacker model used in Dullien's

paper allows modifying pointers that are stored outside of the registers without such a programming error and thus assumes that memory corruption is possible regardless of the implementation. Therefore an implementation that is secure against this attacker is secure against all memory corruption attacks that do not target registers.

1.2. Motivation

For further evaluating the proposal by Thomas Dullien it seemed helpful to have a way of testing his theoretical ideas in a more practical matter. For this, it is necessary to have a way of emulating the machine model he proposes in the paper. Therefore this thesis will explain the creation of an emulator for this model. Furthermore, to check that the emulator and his implementations are working, the two implementations of the example program he used will be run in the emulator. The memory corruption attack he proposes for one of the implementations will also be shown to work.

1.3. Overview of this Thesis

For this, the machine model is first described in Chapter 2 and then the programming language used to program for this machine model is explained in Chapter 3. Both of these chapters also explain the necessary assumptions that have been made to be able to implement these. In Chapter 4 the further requirements and the resulting design decisions made while implementing the emulator are explained. Finally, the two programs written by Thomas Dullien are explained and made to work in Chapter 5. Furthermore, this chapter shows how the attack explained in the weird machines paper can be emulated in the emulator.

1.4. Related Work

This thesis is mostly based on the paper "Weird machines, exploitability, and provable unexploitability" by Thomas Dullien. The patent [4] proposes an emulation of the hardware of a computer to find bugs, including memory leaks. But this does not include an actual implementation of an emulator and also does not allow for the software to be attacked. "Weird Machines as Insecure Compilation" [5] also looks at Dullien's paper and provides a generalized definition of weird machines by seeing them as an insecure compilation of a program.

2. The Machine Model

This chapter will describe the machine model used by Thomas Dullien so it can later be implemented in the emulator. For this the description of the model from the paper is summarized. Afterward, the definition of the operator `J` is explained as it is used but not described in [1].

The machine model is described to be a Cook and Reckhow random access memory machine [6]. Furthermore, it assumes, at least for the example programs that a Harvard architecture is used and therefore the memory used for storing the machine instructions and for storing the values used by these are separated. The memory for the data is limited to 2^{16} cells and every memory cell can store a 32-bit number. Also, the first seven memory cells are labeled as registers.

2.1. Operators

The operators of the machine are described in the paper as shown in Table 2.1.

LOAD(C, r_d)	$: r_d \leftarrow C$	Load a constant
ADD(r_{s_1}, r_{s_2}, r_d)	$: r_d \leftarrow r_{s_1} + r_{s_2}$	Add two registers or a register and constant
SUB(r_{s_1}, r_{s_2}, r_d)	$: r_d \leftarrow r_{s_1} - r_{s_2}$	Subtract two registers or a register and constant
ICOPY(r_p, r_d)	$: r_d \leftarrow r_{r_p}$	Indirect memory read
DCOPY(r_d, r_s)	$: r_{r_d} \leftarrow r_s$	Indirect memory write
JNZ/JZ(r, I_z)		Transfer control to I_z if r is nonzero, zero
READ(r_d)	$: r_d \leftarrow input$	Read a value from input
PRINT(r_s)	$: r_d \rightarrow output$	Write a value to output

Table 2.1.: Operators of the machine described in [1]

This list is not describing all operators used in the two code samples as an operator called `J` is used. This operator takes a label as an argument, as seen in lines 28, 33, 36 and 43 of *variant1A.s* (Listing 12). The operator can also be found with two parameters, a register, and a label, in line 19 of *variant2A.s* (Listing 13), but this is probably not intended as will be explained in Section 5.1.1. To figure out what this operator is supposed to do the operators `JNZ` and `JZ` are helpful as these are the only other operators working with labels. Both of these operators are conditional jump operators. `JNZ` tests for a nonzero value in a register and `JZ` for zero value, therefore it is to be assumed that `JNZ` is an abbreviation for “Jump NonZero” and `JZ` for “Jump Zero”. It is, therefore, to be expected that `J` should stand for “Jump” and is thereby an unconditional jump operator. To test if this definition of the operator is correct it makes sense to check if the program is working as explained in the paper under the assumption that `J` is an unconditional jump operator. It was possible to show that this is the case

for *variant1A.s* after implementing the emulator. Therefore, validating the decision to use `J` as defined in Table 2.2.

J(I_z)	Transfer control to I_z

Table 2.2.: Definition of the missing operator

3. The Programming Language

This chapter will explain the programming language used for the example programs in [1]. As this language is not defined or explained in the paper it was necessary to make certain assumptions as to how certain statements should be interpreted. These assumptions and the reasoning for these are also explained in this chapter. Furthermore, a property of this language is shown to simplify the implementation of the emulator.

3.1. Basic Structure

The programming language used in *variant1A.s* and *variant2A.s* looks similar to assembly code. Under this assumption, it can be assumed that lines like “BasicStateA:” that are not indented, are labels. And that the indented lines refer to different operators. Lines, starting with “.const” or “const”, are assumed to declare constants as “const” is a common keyword for constants [7] and the names behind the “const” keyword are later used again in places that need constants according to the operator definition. Therefore, each line in a program file is either a label, a definition of a constant, an operator with some parameters or a blank line. These different statements will now be looked at more closely.

In the examples, all labels start with an upper case letter and all constants with a lower case letter. As these have no other simple way to be separated, without looking at the places these are defined, it was decided to say that a label is a word that starts with an upper case letter and a line defining a label contains the label followed by a colon. Whenever labels are referenced only the name of the label is used.

Constants are defined in a line starting with either “.const” or “const” followed by the name of the constant and then some arithmetic definition of the value of the constant. This can be either just an integer or a mathematical expression evaluating to an integer.

All remaining lines in the example files contain some identifier for one of the operators, as defined in the next section. Furthermore, each line contains an operator and one to three references like “r0”, “PWasFound”, “2” or “firstIndex”. These references are separated by a comma and space. It will now be reasoned what these references should mean and that this is consistent with the definitions of the operators in Section 2.1.

It was already explained that references starting with an upper case letter define labels and those starting with a lower case letter refer to constants. This definition would also label references starting with an “r” followed by a number as constants, but these are not defined in either file. Therefore, one can assume that these have a different meaning. As the definition of the operator `READ` says that it needs a register as a parameter and it is used in line 4 of *variant1A.s* with the reference “r0”, it is assumed that registers are referenced by these references.

This leaves just a hand full of not yet defined references of which most are just numbers, and are therefore probably just constants. Thereby only "used_head+1" and "5000*3+7" remain, were `used_head` is earlier defined as a constant. Both could, therefore, be computed at the beginning of the emulation and this would result in a constant number. By looking at the definitions of the operators and the parameters these take and comparing this with the two code samples and the above definitions it is possible to check if the definitions above fit these. This is the case.

Some lines have some commentary sentences at the end of the line behind a hash. Therefore, it is assumed that comments start with a hash. This also matches a common usage of this symbol as it used in the same way in, for example, the UNIX command shell [8, p.70]. Therefore, everything following a hash can be safely ignored.

For the simplicity of writing additional code in this language and to simplify the parsing, it turned out that it is helpful to additionally allow the use of additional whitespace at the beginning and end of a line and everywhere else where whitespace is allowed.

3.2. Mathematical Expressions

The mathematical expressions that can be used in constants or constant expressions use the operators "+" and "*" and some parentheses. To later save the work of implementing all of these by hand and to even allow more mathematical operations it seemed reasonable to allow all JavaScript expressions that evaluate to a number in the definitions of constants and constant expressions. These expressions then allow using `eval()` to directly evaluate these and also allow for a lot more flexibility.

3.3. Independence of Lines

In some programming languages, context of other lines is needed to parse a line. JavaScript, for example, contains comments that can span multiple lines. Therefore the information that the current line is a comment might follow from information from earlier lines. This section will now show that such context is not needed for parsing the language used here and that every line of the program can be parsed independently without the knowledge of any other line.

Therefore, it is necessary to show this for the three different kinds of lines that exist. For constants and labels, this is intuitively clear as these only assign a fixed value to a variable or a line number to the name of a label. For statements, this is a bit harder to conclude as the parameters might refer to labels, constants, constant expressions or registers. But this is no problem either as the values of these things are not important for the parsing and it is still possible to differentiate them based on the naming of labels and constants described earlier. Thereby it is shown that the content of other lines is not needed for parsing the source code.

4. Design Decisions for the Implementation of the Emulator

The requirements for the emulator that was written for this thesis, arising from the machine model and programming language used in [1], will be explained in this chapter. Furthermore, some of the design decisions surrounding the implementation of the emulator are explained. Of these, first, the decision to create a command-line interface (CLI) is explained. Then the decisions surrounding the programming language and tooling are explained as well. Next, decisions surrounding the parsing of the code and for the implementation of the machine are reasoned. In the end, the decisions surrounding the configuration and the storage of executions will be explained.

4.1. Requirements on the Emulator

For evaluating the concept of weird machines as proposed by Dullien the machine emulated by the emulator must be the same as the one he proposes in the paper. This machine model is further explained in Chapter 2. For the evaluation, it is furthermore helpful to be able to execute the two implementations of the program he is providing. For this, the emulator should support programs written in the language he uses for these implementations. This language has been defined in Chapter 3 of this thesis.

Due to the usage of weird machines to formalize exploits, and more specifically memory corruption exploits the emulator needs to support an attacker with the capabilities of the “Arbitrary program-point, chosen-bitflip, registers” attacker model. This means more specifically that the emulation should be able to stop at any point and that bitflips can be executed on memory cells when the program is stopped.

For further evaluation of the proposal and to make sure it is not only working on this specific machine certain properties of the machine should be configurable. These are the total size of the memory and amount of registers. To increase the flexibility of the emulator, even more, it should be possible to define new operators and to overwrite or completely remove the existing operators.

A reliable way to repeat the same emulation is also necessary to repeat a possibly working attack. For this, it is necessary to have the emulation be run automatically from an input file that provides several instructions for to the emulator, like “execute the next 20 steps of the emulation” or “flip the 23rd bit of memory cell 42”. These instructions will be defined more clearly later on.

To test an emulation by hand it is also necessary that the emulator has some way to be controlled manually. This should allow the user to execute a fixed amount of emulation steps, provide input to the INPUT operator, view the values of arbitrary memory cells and run the attack. It would also be good to be able to save such an execution so it can later be run again in the automatic execution.

4.2. CLI

To be able to control the emulator without investing much time into developing a GUI and to also allow the execution of the program on multiple different operating systems the decision was made to develop the emulator as a CLI.

4.3. Programming Language

The programming language used for writing the emulator is TypeScript, a superset of EcmaScript that adds a static type system and can be compiled to JavaScript [9].

This language was chosen as static types are helpful for a project of this scale, as these help catch many bugs [10] and allow for better tooling and IDE support.

4.4. Development Tools

To make the setup and running of the emulator as simple as possible it is better to use as few tools as necessary while at the same time making the development as simple as possible. For the development in TypeScript, it is necessary to use Node.js and obviously, the TypeScript compiler. Furthermore, some other tools are used to simplify the development process, namely a linter and a testing framework.

To ensure a consistent code style across all files and prevent even more common bugs linting is used. For a coding style, the decision was made for the JavaScript Standard Style. These code style rules are used by multiple big open source projects, like express and electron [11]. To make these rules work with TypeScript it was necessary to install eslint and then apply the JavaScript Standard Style config and some additional plugins to the eslint config. It was also necessary to install the TypeScript eslint plugin and parser. As the linter then still threw some wrong errors when exporting type definition the "import/export" rule was disabled. As there were some more problems with type definitions being identified as unused variables even when these were used, the "no-unused-vars" rule was replaced by "@typescript-eslint/no-unused-vars". With these changes the linting does work and many simple errors can even be fixed automatically by the linter.

For writing tests, a testing framework was needed. The decision here was made for Jest as it is, according to the State of JavaScript 2018 survey, the testing framework with the highest developer satisfaction and also has good documentation [12]. This additionally meant that Jest needed to be installed together with TypeScript type definitions for it.

The testing is used to ensure that some key parts of the emulator are working correctly. They are used during the implementation of the parsing to have proper examples of what the parsed statements should look like without needing to manually look at the abstract syntax tree (AST) of the program for every step of the implementation. This also makes sure that further development does not break these key parts again.

All tools are installed via npm and are defined in the package.json file, so they can all be installed at once. To not need to remember the specific commands of the different tools[1], general commands, for

[1] Like "eslint" for linting, which also requires some additional parameters.

running the emulator, building it, formatting the code and testing it, were defined in this file[2]. The test command not only runs the tests but also checks that the linting was successful. To not commit code that is failing the tests or has linting errors a pre-commit git hook was also added that runs the tests and lints the code before committing.

4.5. Lexing and Parsing of the Source Code

The compilation of a programming language is typically done in multiple phases. At first, the lexer splits the code into tokens. Then a parser converts these tokens into a syntax tree. And this syntax tree is then type-checked and converted to a list of instructions by the type checker and code generator [13, p. 18 - 20]. As the scope of this thesis is to create an emulator it is only necessary to interpret the code and therefore no code generation at the end is necessary. Thus, the last phase of the creation of a compiler can be skipped [13, p. 20].

The way of creating a lexer and a parser described in [13] is to write a language definition in the Backus Naur Form (BNF) and then use BNF Converter (BNFC) to generate a lexer and a parser. The problem with this approach for this project is that BNFC can produce a lexer and a parser implemented in Haskell, C, C++, C, Java, and OCaml but not in JavaScript or TypeScript [14, p. 1] [15]. The programming language used in the paper is quite similar to assembler and therefore a low-level language. Low-level languages are simpler to compile than higher-level languages [13, p. 15]. Therefore, it seemed to be too much work to use another programming language to be able to use BNFC and it was instead decided to implement a lexer and a parser by hand.

The main part of the lexer is to split the source code at every line break as each line in our language contains exactly one task. Thereby the code is already split into all the different statements. To then convert each line further into tokens all whitespace is replaced by single spaces and whitespace at the beginning and end of each line is removed. Comments are not relevant for the execution of the program. They can also be simply removed in this step by removing everything behind a #. This also simplifies the further parsing as all the remaining code contains some information needed to parse the code. To also remove whitespace directly in front of comments the comment removal is happening before unnecessary whitespace is removed. The lexer is, therefore, doing the steps as shown in Table 4.1 and produces a consistent version of the source code that can then be used to create a syntax tree.

The next step is to convert these tokens into an abstract syntax tree, for this a parser is needed.

As shown in Section 3.3 each line can be parsed independently of the other lines, therefore, the parser is implemented in this way. First, it is therefore needed to decide if the current line contains a label, a constant or a statement. This is done by first checking if the line starts with ".const " or "const " and therefore is a constant, as this is how a constant is defined. The next step is identifying labels by looking if it ends with a colon which is the end character of a label definition and can not be used as the last character of a statement as this is not an allowed character for the name of a constant or a label and no JavaScript expression can end with a colon [3]. The two remaining possibilities are blank

[2] These general commands then e.g. allow running the linting with "npm run lint". It is therefore no longer needed to know which linting tool was used and which parameters it needs.

[3] The colon is not allowed to be used in identifiers and the usage in the ternary operator and the assignment of object

<table>
<tr><th>Step</th><th colspan="2">Output</th></tr>
<tr><td>Input</td><td colspan="2"><code>ADD r3, 1, r2</code>
<code>ICOPY r2, r1 # Load the stored secret.</code></td></tr>
<tr><td rowspan="2">Split lines</td><td colspan="2"><code>ADD r3, 1, r2</code></td></tr>
<tr><td colspan="2"><code>ICOPY r2, r1 # Load the stored secret.</code></td></tr>
<tr><td rowspan="2">Remove Comments</td><td colspan="2"><code>ADD r3, 1, r2</code></td></tr>
<tr><td colspan="2"><code>ICOPY r2, r1</code></td></tr>
<tr><td rowspan="2">Remove unnecessary whitespace</td><td colspan="2"><code>ADD r3, 1, r2</code></td></tr>
<tr><td colspan="2"><code>ICOPY r2, r1</code></td></tr>
</table>

Table 4.1.: The different steps of the lexer and the output after each step

lines and statements. Therefore, a simple check, for blank lines, was added and the remaining lines are treated as statements.

To now parse a label line the only need is to remove the last character off the line as this is a colon. To parse constants the first space is used to split off the name and the remaining text is then evaluated to the value of the constant with the `eval()`-function to get a number as a value.

Parsing of statements is a bit more complicated as the parameters of the statements also need to be parsed. To get the name of the operator it was again possible to just take the first word. To parse the parameters the remainder of the line is split at each occurrence of ", ", the decision to split at ", " and not at every comma was made to allow constant parameters to still contain commas and thereby support more JavaScript, like functions with more than one parameter. This is for example needed for the calculation of maximums and minimums with the help of `Math.max(a,b)` or `Math.min(a,b)`. It thereby seemed like a good tradeoff to enforce the space after the comma in between parameters while not allowing it in the JavaScript parts. This also does not limit the capabilities of the JavaScript expression as a space after a comma is optional in JavaScript [7, sec. 11.2]. Checking for labels and constants gets simpler if the registers are already filtered out as there no longer is a need for a special rule for the "r" character. Therefore, the parser starts by testing if the parameter is referencing a register by checking if it is the letter r followed only by numbers with the help of the following regular expression: `/^r\d+\$/`. Now the next two types of parameters to detect are labels and constants as these both have a simple known structure and these are words that either start with a lower or an uppercase letter. So the parser can use the regular expressions `/^A-Z\w*\$/` and `/^a-z\w*\$/` to check for these respectively. The only remaining parameters are now constant expressions. These expressions are not evaluated further at this stage as these can contain constants and the parser has no access to these.

4.6. Implementation of the Machine

This section describes the decisions for the implementation of the machine. More precisely the way the current state of the machine is stored, how the memory is represented and how the operators are implemented.

properties both need another operand afterward. All other places where a colon can appear (like strings and regular expressions) are enclosed by some other characters [7, cha. 12].

4.6.1. State Object

The current state of the emulator is saved in one object that is passed around between the different parts of the emulator. The idea behind this is that the emulator can be stopped at any point and restarted with the same state at a later time without changing anything of the execution and also allowing the same execution to be repeated multiple times by passing the same state. Furthermore, this allows the different parts of the emulator to be tested directly and independently without needing to setup anything but the state. It allows the tests to check if the state that is returned by the part of the emulator is the same as the expected state.

One alternative for this would have been to use a more object-oriented approach and to store the information in the properties of some class. On the one hand this would allow for more direct access to the different information but on the other hand it would encourage functions that have side effects and thereby increase the complexity of what happens when some function is called and also the testability is made more complicated as it is not enough to check the return value of the method with some precomputed value.

A third option that was considered was not passing the complete state to every function but to instead pass only the information the function needs. This would lead to even fewer things one needs to consider when calling the function as it is already clear from the beginning which values the function might read. The big problem with this approach has been that a function can normally only return a single value but a lot of the functions used for the emulator would need to change more than one property of the state. This could have been solved by returning the modified parts of the state object and then merging these but this would probably also lead to a lot more issues as merging of objects can only be simply done for a single layer of properties[4]. A further problem with this approach would be that some functions would need a lot of parameters and that this would lead to less readable code[16].

Therefore, the first approach was chosen for storing the state of the emulator.

4.6.2. Memory

The memory is modeled by the class described in Figure 4.1. This class internally uses a simple array of numbers (*_memory*) to save the values of the different registers and memory cells as this is a simple method of storing this kind of information and there is no need for a more complex data structure. The array is wrapped in the Memory class to provide a way to check the restrictions of the machine model. These checks are done by creating methods of the class that check these restrictions before reading or writing the memory. Furthermore, this allows storing the register and memory values in the same data structure, which is the same way it is described in the machine model while also being able to check if a read or write is made to a register or memory cell. For this purpose, the read and write methods exist in three different kinds: One pair that allows access to both registers and memory cells (*get*, *set*), one that only allows accessing registers (*getRegister*, *setRegister*) and the last one that only allows access

[4] A JavaScript object can have another object as a property. But when merging such an object with another object with the same nested property by using object spread or `Object.assign` the properties of the nested object would not be merged but overwritten. So when merging `{ a: { b: 1 }, c: 2 }` and `{ a: { c: 3 }, d: 4 }` the result would be `{ a: { c: 3 }, c: 2, d: 4 }` and the value of `b` would have been lost. Similar problems also occur when trying to merge objects containing lists.

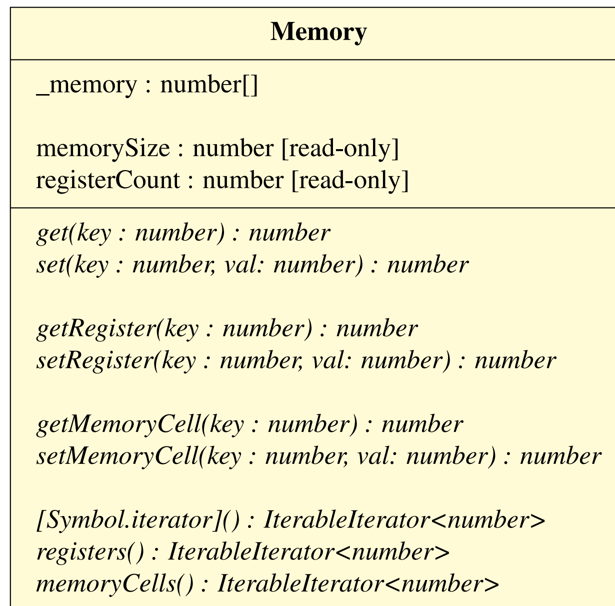

Figure 4.1.: Partial class diagram of the memory

to memory cells (*getMemoryCell*, *setMemoryCell*). There also exist three iterators that support to read the complete memory, only the registers and only the memory cells without iterating over the possible *key* values. These are mainly used for informing the user of the current state of the memory.

4.6.3. Operators

Operators need to be able to modify the state of the emulator and can get a known fixed number of parameters of certain types (Register, Constant or Label). To allow for great flexibility of what the operators can do the best method seemed to be to give them access to the complete state of the machine and to return a new state. To reduce the number of mistakes while programming a program for the emulator it was decided to include some type checking for the parameters of the operators. Therefore it is also necessary to define what each parameter of the operator is allowed to be. Several operators allow for some parameters to not only be of one type but multiple different. The `ADD` operator, for example, can take the second parameter to be a reference of a constant, a constant expression or a register. For this reason, the decision was made to store these types in a list of lists, so the first list lists all the types the first parameter can have, and the second list the different types for the second parameter and so on. For the `ADD` operator, this would lead to the definition in Listing 1.

```
[
    ['REGISTER'],
    ['CONSTANT', 'CONST_EXPRESSION', 'REGISTER'],
    ['REGISTER']
]
```

Listing 1: Parameter type definition of the `ADD` operator

Therefore, every operator is represented by an object containing the parameter types and a function

that will be called by the current state and the parameters and then needs to execute the operator. To allow for better type checking with TypeScript a helper function `createOperator` was created that infers the types of the parameters automatically based on the parameter type names provided in the list. With this, it is possible to define the operators in a relatively simple way as can be seen in Listing 2 for the `LOAD` operator.

```
17    LOAD: createOperator([
18      ['CONSTANT', 'CONST_EXPRESSION'],
19      ['REGISTER']
20    ], (state, val, target) => {
21      state.memory.set(target.value, loadValue(state, val))
22
23      return state
24    }),
```

Listing 2: Implementation of `LOAD` operator

As some operators, like `ADD` and `SUB`, are using the same signature and quite similar implementations further helpers for simple arithmetic and logical jump operators were defined that define the parameter types and also handle the memory accesses or setting of the program pointer. With these it is possible to define the `ADD` and `SUB` operators each in only one line as can be seen in Listing 3.

```
5    ADD: createArithmeticOperator((a, b) => a + b),
6    SUB: createArithmeticOperator((a, b) => a - b),
```

Listing 3: Implementation of `ADD` and `SUB` operator

READ Operator

For the implementation of the `READ` operator, it is necessary to be able to allow the user to input information via the CLI but it is also necessary that this input can be provided automatically for the automatic execution. Furthermore, future inputs might not be available at the beginning of the execution as it would be inconvenient for the user to need to decide about all future inputs before beginning the emulation of the program. Therefore, it is not possible to store the input in a simple immutable list that could then be read by the `READ` operator. Therefore, it is necessary to add elements to this list while the emulation is running, more precisely at exactly the moment, the `READ` operator is executed. Multiple implementations of this operator, for both the manual and automatic emulation, seemed to produce duplicated code. As a solution asynchronous iterators, provided by the manual or automatic emulation that will provide the inputs, are used instead. This allows to prompt the user to input the next input value at exactly the moment the operator tries to read it while also allowing the automatic emulation to provide a list of inputs at the beginning of the execution while using the same implementation of the operator for both emulation variants.

4.7. Configuring the Emulator

As the emulator should be configurable the question arises how to provide the emulator with the configuration values. Typical ways for passing such information to a program are by using command-line arguments or environment variables. Additionally, some programs also allow the use of a configuration file. As it should be allowed to configure operators, which need the full support of JavaScript to work, the thought of using only environment variables or command-line arguments was quickly scrapped as transferring multiple lines of source code in this way is quite cumbersome.

The configuration options of the machine were decided based on what that made sense to change and is simple enough to change for the scope of this thesis. These are first of all the complete size of the memory (`memorySize`) and the amount of these memory cells that are registers (`registerCount`). Also, additional operators can be defined in the `operators` object. The default operators might not always be wanted so these can also be removed with the help of the `useDefaultOperators` option.

The remaining configuration options were added to simplify the execution of programs with certain options and to also not need multiple command line parameters. They are `sourceFile` for the source code of the program that should be emulated and `actionFile` for a file with a stored emulation. Furthermore, the option `continueAfterAutomaticExecution` was added to be able to shut down the emulator after the execution of an action file has ended.

If the emulator is executed in a TypeScript environment[5] the emulator expects a TypeScript file as a configuration file. If a JavaScript environment is used[6] a JavaScript file is expected. This way of using different configuration files for different execution environments is needed as the compiled JavaScript code requires the default export of a module after "`module.exports =`" while TypeScript requires this, with the chosen TypeScript configuration, behind "`export default`". Providing the configuration file in a general format, for example as a JSON file, was not possible as the full power of the programming language is needed for implementing additional operators.

The configuration for the emulator that is explained in [1], is provided as a default and can also be seen in Listing 10 and 11.

4.8. Storing and Loading Executions

One of the requirements is to be able to store the execution of an emulation to a file. For this, a file format, for storing these executions that also allows reapplying these again was needed. It is also necessary that the same inputs to the emulator will result in the same state of the machine. This last part is already given through the functional style of running the emulator. The file format needs to allow defining that a certain number of emulation steps should happen, what input should be provided when and also the bitflip attack should be definable within it. For this, it seemed reasonable to use a simple format that has one input to the emulator in every line and that have their different parameters split by one space each. This allows for the parsing of the different actions to be simple[7] as it is only

[5] For example by using *ts-node*.

[6] By first building the emulator using *npm run build* and then executing it with for example *npm start*.

[7] The parser just splits first at linebreaks and then splits every line at every space and looks at the first element to figure out the action and uses the remaining elements of each line as parameters for the action.

necessary to split at new lines and spaces and then look at the first element of each line to determine the action to be executed. The different actions that can be used are defined in Table 4.2.

Action	Definition
EXECUTE n	Execute the next n steps of the emulation.
INPUT n	Provide the input n to the emulator the next time an input is needed.
CHOOSEN_BITFLIP n m	Flip the mth bit of the memory cell n.
PRINT_MEMORY n m	When n and m are provided the values of the memory cells n to m are printed. If only n exists the binary value of the memory cell n is printed. If neither exists the values of the first 20 memory cells are printed.
MODIFY_MEMORY n m	Changes the value of memory cell n to m.

Table 4.2.: The different actions that can be used in the actions file format.

Furthermore, the ability to add comments to the file by ignoring all lines that start with "# "[8] was added, so it is possible to explain what will be done directly in the file.

[8]The space behind the hash is required so that the parser for the actions can be used for detecting comments. Internally a comment, therefore, is just an action that does nothing and allows for an arbitrary amount of parameters.

5. The Example Programs

After describing the implementation of the emulator this thesis will now continue to look at the two example implementations of a program used in [1]. These implementations were used by him to show that different implementations of the same program can fulfill different security properties. The program is a key - secret store that should allow to safely store a secret that can only be obtained by knowing the corresponding key. The first implementation of the program, known as *variant1A.s*, stores these keys and secrets in a simple flat array structure where all odd fields contain a key and the following fields contain the corresponding secrets. The other implementation, called *variant2A.s*, instead uses a linked list to store these values.

variant1A.s could be executed in the emulator without any problems. But the execution of *variant2A.s* turned out to be more complicated due to some mistakes in the program code of this implementation. These mistakes will now be explained further and it will also be shown how they were resolved. Then a second implementation of *variant2A.s* is explained that is implemented as described in the paper. Furthermore, the execution of the described attack of *variant2A.s* in the emulator will be explained and a simpler attack will be shown. Also, an attack for the second implementation will be shown.

5.1. Fixing *variant2A.s*

The execution of *variant2A.s* did, other than *variant1A.s*, not work directly. This was due to multiple errors. First of all the `J` Operator was used with too many parameters and there was also a typo in the last line. The bigger problem then occurred after fixing these two problems, as the program was then not doing what was expected from the description of the program in the paper. The first thing that did not work was the initialization of the linked list to store the key-value pairs. Afterward, it also became clear that the check for finding a stored key did not work as intended and that the removal of a key-value pair after it had been read did not work as well. In the following sections, each of these problems will be described and it will also be shown how they were fixed.

5.1.1. Wrong Usage of the `J` Operator

In line 19 of *variant2A.s* (Listing 4) the `J` operator is used with an additional parameter before the label, namely the third register. This is not compatible with our definition of the `J` operator but matches the way both `JZ` and `JNZ` are used. The first thing to assume therefore was that this operator should be either `JZ` or `JNZ` and then analyze if the code still makes sense.

To start with `JNZ` if `r3` is not zero in line 19 the jump will happen and the execution continues in line 13, where another conditional jump is possible, but this time if `r3` is zero. Therefore, this jump will

```
12 CheckForPresenceOfP:
13     JZ      r3, EndOfUsedListFound
14     ICOPY   r3, r4         # Load 'p' of the entry.
15     SUB     r4, r0, r4     # Compare against the password
16     JZ      r4, PWasFound  # Element was found.
17     ADD     r3, 2, r3      # Advance to 'next' within [p, s, nxt]
18     ICOPY   r3, r3         # Load the 'next' pointer.
19     J       r3, CheckForPresenceOfP
20 EndOfUsedListFound:
```

Listing 4: Code with the wrong usage of J

happen. This then takes us to the label `EndOfUsedListFound` and thereby to line 20. The program would also have reached line 20 directly if `r3` had been zero as this is the line directly after the wrong usage of `J`. In conclusion, it makes no sense for the operator to be `JNZ` as it could just have been left out.

Similarly, it makes no sense for the Operator to be `JZ` as this would just run the same test in both line 19 and 13. This program would be the same as if there is a simple unconditional jump in line 19.

This concludes that the parameter `r3` was just placed in the line by mistake and the correct version is supposed to be "`J CheckForPresenceOfP`".

5.1.2. Typo in the Last Line

The last line of the second implementation contains the code "`J BasiceStateA`". The label `BasiceStateA` used here is not defined anywhere in the program but another label with a similar name exists in line 4 (`BasicStateA`). This label is only different from the used label by a single character[1] and furthermore, a word called **Basice** does not exist in English while **Basic** does. Therefore, it can be concluded that this was just a typo and the correct line would instead be "`J BasicStateA`".

5.1.3. Initialization of Linked Lists Not Working

Even after fixing these two syntax errors the execution of the program still failed during the initialization of the linked list. The execution runs into an out of memory error. By looking at the execution more closely it became clear that the calculation in line 52 that checks whether enough of the memory has been initialized, can not become zero and the initialization thereby never ends. This can be shown by first showing that the value in register zero is always of the form $5+3*x$. This is the case as the initial value of the register is the value of `free_head` (5) and it then gets increased every iteration of `LoopToInitialize` by three[2]. Therefore, the calculation in line 52 can not become zero as $10000*3+7 = 30007$, $30007-5 = 30002$ and $30002 \textit{ mod } 3 = 2$. To fix this the code of the initialization of the free head was moved in front of the initialization of the values of the free list.

[1] The **e** at the sixth position does not exist

[2] This does not happen by simply adding three to `r0` but instead, the value of `r0+3` is written into `r1` and this value is then copied to `r0` before the calculation.

```
45 InitializeFreeList:
46     LOAD    free_head, r0
47 LoopToInitialize:
48     ADD     r0, 3, r1        # Advance to the next element.
49     ADD     r0, 2, r0        # Advance to the next pointer inside.
50     DCOPY   r0, r1           # Write the next pointer.
51     ADD     r1, 0, r0        # Set current elt = next element.
52     SUB     r0, 5000*3+7, r2  # Have we initialized enough?
53     JNZ     r2, LoopToInitialize
54 TerminateFreeList:
55     SUB     r0, 1, r0
56     DCOPY   r0, r2 # Set the last next-pointer 0 to terminate
57                    # the free list.
58 WriteInitialFreeHead:
59     LOAD    used_head+1, r0
60     LOAD    free_head, r1
61     DCOPY   r1, r0     # Set the free_head to point to the first triple.
62     J       BasicStateA
```

Listing 5: Initialization of the free list in *variant2A.s*

5.1.4. Check for Matching Stored Keys Not Working

While looking for further logical mistakes in the program, as it was still not working as expected, the check for the presence of a key has attracted attention as this check was not working as it never found an existing key and instead always adds a new entry to the list. While looking at the code it can be seen that in line 11 the value of the `used_head` constant is loaded into the third register and that this value then is used in line 14 to load the key of the next entry. But this loads the value of the memory cell six, which stores the pointer to the head of the used list and not the key of the first entry of the used list (see Table 5.1). Therefore, the check in line 16 is not checking if the key is matching a stored secret but instead checking if the key is the number of the memory cell, storing the head of the used list. This also does not match the comment in line 14 that claims to load the key of the triple. To instead load the key in line 14 it is possible to load the value of the memory cell six into register 3 and therefore having a pointer to the next memory cell with a stored key in register 3. Therefore, the statement `ICOPY r3, r3` was added into the code between line 11 and 12 and get the code shown in Listing 6. With this change, the code now correctly compares the provided key with the stored keys and also the iteration through the list entries is working correctly.

5.1.5. Unlink of Read Entries Not Working

With the changes above the program now works for storing secrets and it is also possible to read one secret from the storage. But after one read the program is no longer able to add new secrets to the linked

description	cell	value
key input	0	33
secret input	1	44
unused	2	0
value of used_head	3	6
unused	4	9
pointer to free list head	5	10
pointer to used list head	6	7
stored key 1	7	11
stored secret 1	8	22
pointer to next used entry	9	0
empty	10	0
empty	11	0
pointer to next free entry	12	13

Table 5.1.: Memory cells with descriptions when reaching line 12 of *variant2A.s* after the inputs 11, 22, 33 and 44

```
8  CheckForNullSecret:
9      JZ     r1, OutputErrorMessage # Zero secret not allowed.
10     JZ     r0, OutputErrorMessage # Zero password not allowed.
11     LOAD   used_head, r3 # The list consists of [p, s, nxt] tuples.
12     ICOPY  r3, r3
13 CheckForPresenceOfP:
14     JZ     r3, EndOfUsedListFound
15     ICOPY  r3, r4         # Load 'p' of the entry.
```

Listing 6: Fixed version of the code used to load the stored keys. The original version is missing line 12.

list and ends up in a key out-of-bounds error when adding another entry or trying to read another entry of the linked list that was added earlier. After investigating this issue further it became clear that the unlinking of the read elements of the linked list is not working. By running the code in the emulator it can be seen that the memory appears to be correct after inserting three elements (see Figure 5.1) but after reading the second element of the used list the memory is no longer appears to be correct as there are multiple parts of the linked lists that are no longer connected and the removed element is not even pointing to an element of the list (see Figure 5.2). The out-of-bounds error, therefore, is happening as the program tries to jump to the memory position of the value two memory cells after cell 5 and this is the secret of the first entered entry (44444) and this is no longer part of the memory. To fix this it was necessary to make sure that the pointers of the linked list point to the correct positions.

For this, the first change was to change the link for the read element from memory cell 5 to instead point to the value of memory cell 5. With this change, the free list is once again working as intended. The more complicated part is now to link up the used list correctly. For this, it is needed to find the element that is pointing to the current element. This was implemented by extending the `PWasFound`

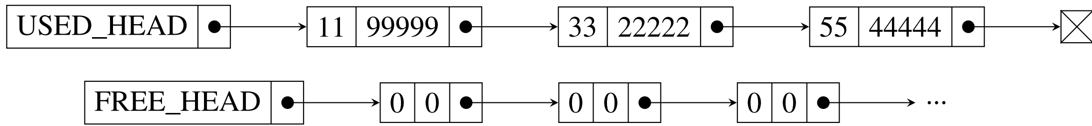

Figure 5.1.: Memory Layout after inserting three elements.

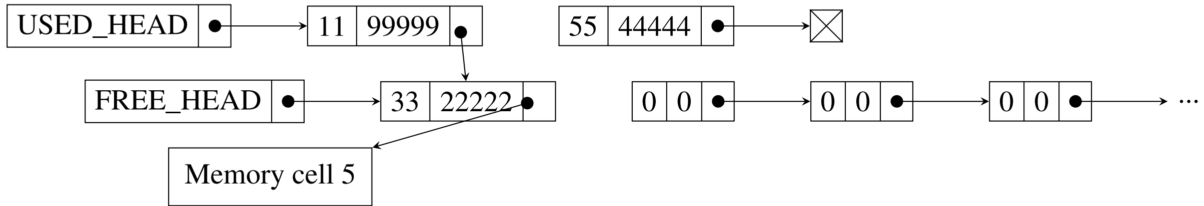

Figure 5.2.: Memory layout after reading the second element.

part to find the previous element and to then point the pointer of this element to the target of our previous element. The modified code can be seen in Listing 14 and produces the expected memory structure shown in Figure 5.3.

With these changes, the program does now work, from the users perspective, the same as *variant1A.s*.

5.2. Implementing *variant2A.s* as Described in the Paper

While using the now working implementation of *variant2A.s* it became clear that this implementation is not working as described in the paper. The implementation links the list with the used elements up in such a way that the head of the list points to the newest element in the list while the paper describes the head to point to the oldest element. To be able to test if this has any implications on the ability to attack the implementation a second implementation (*variant2B.s*) was created that implements the linked list in such a way that the head points to the oldest element. The code of this implementation can be seen in Listing 15.

5.3. Attack on *variant2A.s*

With these three changes also *variant2A.s* is working as expected. So it is now possible to look at the attack explained in [1] and try to run it using the emulator. As the attack is described for the implementation of the linked list as it is implemented in *variant2B.s* it is necessary to modify the attack a bit to work. To keep the structure of the memory simple to understand throughout the attack the values 10X are used as keys and 20X as secrets, were X is a number increment with every write, for the attacker's input. And for the attacked users input 99 is used as key and 999 as secret.

The basic setup of the attack stays the same as in [1] (Step 1 and 2). The now following third step

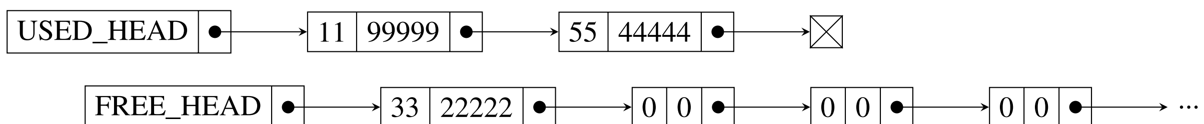

Figure 5.3.: Memory Layout after reading the second element in the fixed version.

is used to get the pointer to point to the attacked user's entry before it points to the entries that will be modified. This step can be skipped as this is already accomplished by the inverse direction of the linked list. Therefore, the memory corruption, of course, needs to be a bit different as the last two bits of the memory cell 18 need to be flipped to get the pointer of the entry before (in terms of the memory, not the linked list) the attacked entry to point to the secret instead of the key. With this done it is now possible to input the secret of that entry and thereby freeing the two last cells of that entry and the key cell of the attacked entry. This cell is then also overwritten by the next free pointer. So with entering the value of this pointer (19), the program will output the secret 999. The concrete actions of the attack can be seen in Listing 7.

5.4. A Simpler Attack on *variant2A.s*

The attack on *variant2A.s* seemed rather complicated as multiple entries are added to the linked list that are not used. It turns out that it is possible to attack the program by only adding the first input and then waiting for the user's input and increasing the pointer from this input by one. For this increase by one four bitflips are needed[3]. If another entry is added before the user is allowed to add their entry only a single bitflip is necessary for the modification of the pointer[4]. After this, the secret can be once again used to overwrite the user's key by the pointer of the free list and then this pointer can be used to read the secret from the memory. The action file for this attack is provided in Listing 8.

5.5. Attack on *variant2B.s*

For attacking *variant2B.s* the attack could be executed as described by Thomas Dullien. The only necessary change was the adjustment of the value of the pointer to the free list, as this value in the paper assumes that memory cell 0 is the first memory cell that stores a value of the linked lists and ignores the existence of the 5 registers and the memory cells for the heads of the two lists in front of this. Therefore, the value 19 was used for this instead of 12. An action file that shows this can be seen in Listing 9.

[3]The pointer previously points to memory cell seven and now needs to point to memory cell eight.

[4]In this case, the pointer points to memory cell ten and only one bit needs to be flipped to change this to memory cell eleven.

6. Evaluation

This thesis aimed to create an emulator for the machine model described in Chapter 2 to be able to evaluate the proposal for a definition of exploitability by Dullien. Now it will be evaluated how well the emulator can fulfill this task.

The emulator can execute the example programs *variant1A.s* and *variant2A.s* and run the memory corruption attack on *variant2A.s*. It was also possible to run further implementations on the emulator (*variant2B.s*). Other smaller programs, for example, a program to add two numbers, were also tested and could be executed without problems. It was furthermore also possible to attack *variant2B.s* successfully. In conclusion, it can, therefore, be said that the thesis is fulfilling its main task of creating an emulator to test the proposal by Dullien.

Furthermore, there are a few other things to evaluate, namely if all the additional requirements for the emulator have been met and how suitable the design decisions about implementing the emulator were.

6.1. Additional Requirements on the Emulator

The first additional requirement was that certain properties of the machine can be modified, this is the case for both the memory size and the number of registers. This can be seen in the example configurations files (Listing 10 and 11) and was also tested successfully.

The addition of more operators to the language supported by the emulator is also possible with the configuration files and was tested by adding an operator `MUL` that multiplies two numbers and using it in a small program. The removal of the default parameters is also possible and therefore the user can define a completely new set of operators that then work on the same machine. Therefore all the requirements for the configuration of the emulator are fulfilled.

Furthermore, the emulator is supposed to have a way to store executions to be able to reliably reproduce results. This is implemented using the action files that have also been used in this matter for the different attacks of *variant2A.s* and *variant2B.s*. Therefore this requirement is also fulfilled.

The requirements for the manual execution, namely executing arbitrary many instructions of the machine, inspecting the memory, and executing the bit-flip attack are also all implemented in the emulator. These three things can all be seen in action in Figure 6.1. There also exists a way to store the manual execution to an action file, with the help of the *.actions* command.

Thereby all the requirements listed in Section 4.1 are implemented in the emulator.

```
    3:     READ    r0             # Read p
>   4:     READ    r1             # Read s
    5: CheckForNullSecret:
| Cell  |  0 |  1 | 2 |  3 |     4 | 5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| Value | 10 | 21 | 0 | 15 | 10006 | 0 | 23 | 55 | 32 | 11 | 89 | 98 | 44 | 12 | 12 | 21 |  0 |  0 |  0 |  0 |
#
Input: 30

    4:     READ    r1             # Read s
>   5: CheckForNullSecret:
    6:     JZ      r1, OutputErrorMessage
| Cell  |  0 |  1 | 2 |  3 |     4 | 5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| Value | 10 | 30 | 0 | 15 | 10006 | 0 | 23 | 55 | 32 | 11 | 89 | 98 | 44 | 12 | 12 | 21 |  0 |  0 |  0 |  0 |
# .memory 15
00000000000000000000000000010101
# .bitflip 15 31
# .memory 15
00000000000000000000000000010100
# 10

   14:     JZ      r5, PWasFound
>  15:     ADD     r3, 2, r3      # Advance the index into the tuple array.
   16:     SUB     r3, r4, r5     # Have we checked all elements of the array?
| Cell  |  0 |  1 | 2 | 3 |     4 |  5 |  6 |  7 |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
| Value | 10 | 30 | 0 | 6 | 10006 | 13 | 23 | 55 | 32 | 11 | 89 | 98 | 44 | 12 | 12 | 20 |  0 |  0 |  0 |  0 |
# |
```

Figure 6.1.: Screenshot of the emulator during the manual execution of *variant2A.s*. The screenshot first shows the input mechanism, then the reading of a memory cell in binary, the execution of a bit-flip and then the execution of the next 10 steps in the emulation.

6.2. Design Decisions

The design decisions allowed for a simple implementation of the emulator. Some problems occurred with the state object, as this is by now polluted by data that exists multiple times, for example, the AST is stored there but also the extracted constants and labels are stored as additional properties. Therefore, it might have been better to not use the same state object throughout the whole execution of the program but only pass the properties to the different functions that are needed. This would reduce the amount of data one needs to think about at the same time. Furthermore, it would have been better to think of the manual and automatic execution modes beforehand as in the first versions of these had a lot of duplicated code that was only removed later on and these would have probably been better implemented as different implementations of the same interface/function type. For the lexer and parser, it might have been worth the time to look at ways to implement them as final state machines as this would lead to a cleaner implementation that does not depend that much on regular expression test and multiple other different string methods. The last thing of the implementation not satisfying is that the constant statement parameters are evaluated at multiple places, once in the parser for all that do not use constant expressions and for the others while loading values in the evaluation. Here especially the fact that these require the use of helper functions that might not be used by someone implementing additional operators. This could, therefore, lead to some unnecessary problems for this. Therefore, the decision to only allow access to the current line in the parser might not have been the best, or it might have been better to add a step that evaluates these expressions before starting the emulation itself.

The parts that were tested provided to be quite helpful as these tests did catch bugs in the parser and lexer multiple times during the development. Some further system tests for the complete emulator with some provided action files and expected output would have also been helpful as there were multiple occasions in which a bug lead to the failure of some of the example programs that only was found a lot

of time later.

7. Conclusion

As shown in Chapter 6 the emulator is fulfilling its requirements and thereby the main part of this thesis was completed successfully.

The design decisions for the implementation of the emulator were also mostly good. But a few things should have been more thought out in the beginning, like the state object and the different execution modes. It also turned out that more testing, or to be more precise other tests than unit tests, would have helped catch bugs early on.

For the example programs, it can be said that these were not probably implemented but do show that the attack in the paper works for the second variant after the mistakes in the implementation were fixed. This thereby also shows that the emulator does fulfill its requirements.

Part of the idea of weird machines and IFSMs by Dullien is, thereby, also shown to work in a more realistic environment, as the different implementations of the IFSM fulfilled different security properties in the emulator.

For future work into weird machines, this emulator can be used with other programs to test if these are exploitable. It is also possible to use different operators and register amounts to try out if this changes something about the exploitability of weird machines.

A further possible extension for the emulator could be a way to visualize linked lists to simplify the understanding of the changes of these during the emulation. It would also be possible to integrate the emulator into a webpage to make it more accessible. For this only, the Node.js specific code would need to be replaced by code using the Web APIs.

A. Action Files

```
# Step 1
EXECUTE 30017
INPUT 100
EXECUTE 1
INPUT 200
EXECUTE 23
INPUT 101
EXECUTE 1
INPUT 201
EXECUTE 30
INPUT 102
EXECUTE 1
INPUT 202
EXECUTE 37
# Step 2
INPUT 99
EXECUTE 1
INPUT 999
EXECUTE 44
# Step 4
CHOOSEN_BITFLIP 18 30
CHOOSEN_BITFLIP 18 31
# Step 5
INPUT 202
EXECUTE 1
INPUT 1
EXECUTE 40
# Step 6
INPUT 19
EXECUTE 1
INPUT 1
EXECUTE 3
```

Listing 7: First attack of *variant2A.s*

```
# Setup of linked list
EXECUTE 30017
# Add one entry
INPUT 100
EXECUTE 1
INPUT 200
EXECUTE 23
# Add entry that should be attacked
INPUT 99
EXECUTE 1
INPUT 999
EXECUTE 30
# Increase pointer to our entry by one
CHOOSEN_BITFLIP 12 31
CHOOSEN_BITFLIP 12 30
CHOOSEN_BITFLIP 12 29
CHOOSEN_BITFLIP 12 28
# Input our secret to overwrite secret by the next free pointer
INPUT 200
EXECUTE 1
INPUT 1
EXECUTE 40
# Use the next free pointers value to read the secret
INPUT 13
EXECUTE 1
INPUT 1
EXECUTE 3
```

Listing 8: Simpler attack of *variant2A.s*

```
EXECUTE 30015
INPUT 100
EXECUTE 1
INPUT 200
EXECUTE 27
INPUT 101
EXECUTE 1
INPUT 201
EXECUTE 37
INPUT 102
EXECUTE 1
INPUT 202
EXECUTE 47
INPUT 99
EXECUTE 1
INPUT 999
EXECUTE 57
INPUT 102
EXECUTE 1
INPUT 1
EXECUTE 59
INPUT 101
EXECUTE 1
INPUT 1
EXECUTE 46
INPUT 103
EXECUTE 1
INPUT 203
EXECUTE 47
INPUT 104
EXECUTE 1
INPUT 204
EXECUTE 56
CHOOSEN_BITFLIP 12 30
CHOOSEN_BITFLIP 12 31
EXECUTE 1
INPUT 204
EXECUTE 1
INPUT 1
EXECUTE 72
INPUT 19
EXECUTE 1
INPUT 1
EXECUTE 21
```

Listing 9: Attack of *variant2B.s*

B. Configuration Files

```
import { Config } from './lib/Config'

export default  {
  memorySize: 2 ** 16,
  registerCount: 6,
  useDefaultOperators: true,
  continueAfterAutomaticExecution: true
} as Config
```

Listing 10: Config of the machine in TypeScript

```
module.exports = {
  memorySize: 2 ** 16,
  registerCount: 6,
  useDefaultOperators: true,
  continueAfterAutomaticExecution: true
}
```

Listing 11: Config of the machine in JavaScript

C. Implementations of the Example Program

C.1. variant1A.s

```
.const firstIndex 6
.const lastIndex 6 + (5000*2)
BasicStateA:
    READ    r0              # Read p
    READ    r1              # Read s
CheckForNullSecret:
    JZ      r1, OutputErrorMessage
    JZ      r0, OutputErrorMessage
CheckForPresenceOfP:        # Run through all possible array entries.
    LOAD    firstIndex, r3
    LOAD    lastIndex, r4
CheckForCorrectP:
    ICOPY   r3, r5          # Load the stored p of the tuple
    SUB     r5, r0, r5      # Subtract the input p
    JZ      r5, PWasFound
    ADD     r3, 2, r3       # Advance the index into the tuple array.
    SUB     r3, r4, r5      # Have we checked all elements of the array?
    JNZ     r5, CheckForCorrectP
PWasNotFound:
    LOAD    firstIndex, r3
    LOAD    lastIndex, r4
SearchForEmptySlot:
    ICOPY   r3, r5
    JZ      r5, EmptyFound
    ADD     r3, 2, r3
    SUB     r3, r4, r5
    JZ      r5, NoEmptyFound
    J       SearchForEmptySlot
NoEmptyFound:
OutputErrorMessage:
    SUB     r0, r0, r0
    PRINT   r0
    J       BasicStateA
EmptyFound:
    DCOPY   r3, r0   # Write the password
    ADD     r3, 1, r3    # Adjust the pointer
    DCOPY   r3, r1   # Write the secret.
    J       BasicStateA
PWasFound:
    LOAD    0, r4
    DCOPY   r3, r4   # Zero out the stored p
    ADD     r3, 1, r3
```

```
43      ICOPY   r3, r5  # Read the stored s
44      PRINT   r5
45      J       BasicStateA
```

Listing 12: *variant1A.s* as provided in [1].

C.2. variant2A.s

```
1  const free_head 5  # Head of the freelist.
2  const used_head 6  # Head of the used list.
3      J       InitializeFreeList
4  BasicStateA:
5      READ    r0              # Read p
6      READ    r1              # Read s
7      SUB     r2, r2, r2      # Initialize a counter for number of elements.
8  CheckForNullSecret:
9      JZ      r1, OutputErrorMessage # Zero secret not allowed.
10     JZ      r0, OutputErrorMessage # Zero password not allowed.
11     LOAD    used_head, r3 # The list consists of [p, s, nxt] tuples.
12 CheckForPresenceOfP:
13     JZ      r3, EndOfUsedListFound
14     ICOPY   r3, r4          # Load 'p' of the entry.
15     SUB     r4, r0, r4      # Compare against the password
16     JZ      r4, PWasFound # Element was found.
17     ADD     r3, 2, r3       # Advance to 'next' within [p, s, nxt]
18     ICOPY   r3, r3          # Load the 'next' pointer.
19     J       r3, CheckForPresenceOfP
20 EndOfUsedListFound:
21     LOAD    free_head, r3
22     JZ      r3, OutputErrorMessage # No more free elements available?
23     ICOPY   r3, r2        # Get the first element from the free list
24     DCOPY   r2, r0        # Write the [p, ?, ?]
25     ADD     r2, 1, r4
26     DCOPY   r4, r1        # Write the [p, s, ?]
27     LOAD    used_head, r0
28     ICOPY   r0, r1        # Load used_head to place it in 'next'
29     DCOPY   r0, r2        # Rewrite used_head to point to new element
30     ADD     r2, 2, r4     # Point to 'next ' field
31     ICOPY   r4, r2        # Load the ptr to the next free element into r2
32     DCOPY   r4, r1        # Write the [p, s, next]
33     DCOPY   r3, r2        # Write the free_head -> next free element
34     J       BasicStateA
35 PWasFound:
36     ADD     r3, 1, r2
37     ICOPY   r2, r1     # Load the stored secret.
38     PRINT   r1     # Output the secret.
39     ADD     r3, 2, r2  # Point r2 to the next field.
40     LOAD    free_head, r1
41     ICOPY   r1, r0  # Read the current pointer to the free list.
42     DCOPY   r2, r1  # Point next ptr of current triple to free list.
43     DCOPY   r1, r3  # Point free-head to current triple.
```

```
44      J       BasicStateA
45  InitializeFreeList:
46      LOAD    free_head, r0
47  LoopToInitialize:
48      ADD      r0, 3, r1        # Advance to the next element.
49      ADD      r0, 2, r0        # Advance to the next pointer inside.
50      DCOPY    r0, r1           # Write the next pointer.
51      ADD      r1, 0, r0        # Set current elt = next element.
52      SUB      r0, 5000*3+7, r2  # Have we initialized enough?
53      JNZ      r2, LoopToInitialize
54  TerminateFreeList:
55      SUB      r0, 1, r0
56      DCOPY    r0, r2 # Set the last next-pointer 0 to terminate
57                      # the free list.
58  WriteInitialFreeHead:
59      LOAD     used_head+1, r0
60      LOAD     free_head, r1
61      DCOPY    r1, r0     # Set the free_head to point to the first triple.
62      J        BasicStateA
63  OutputErrorMessage:
64      SUB      r0, r0, r0
65      PRINT    r0
66      J        BasiceStateA
```

Listing 13: *variant2A.s* as provided in [1].

C.3. variant2A_fixed.s

```
1   const free_head 5  # Head of the freelist.
2   const used_head 6  # Head of the used list.
3       J       InitializeFreeList
4   BasicStateA:
5       READ    r0              # Read p
6       READ    r1              # Read s
7       SUB     r2, r2, r2      # Initialize a counter for number of elements.
8   CheckForNullSecret:
9       JZ      r1, OutputErrorMessage # Zero secret not allowed.
10      JZ      r0, OutputErrorMessage # Zero password not allowed.
11      LOAD    used_head, r3 # The list consists of [p, s, nxt] tuples.
12      ICOPY   r3, r3
13  CheckForPresenceOfP:
14      JZ      r3, EndOfUsedListFound
15      ICOPY   r3, r4          # Load 'p' of the entry.
16      SUB     r4, r0, r4      # Compare against the password
17      JZ      r4, PWasFound # Element was found.
18      ADD     r3, 2, r3       # Advance to 'next' within [p, s, nxt]
19      ICOPY   r3, r3          # Load the 'next' pointer.
20      J       CheckForPresenceOfP
21  EndOfUsedListFound:
22      LOAD    free_head, r3
23      JZ      r3, OutputErrorMessage # No more free elements available?
```

```
24      ICOPY  r3, r2       # Get the first element from the free list
25      DCOPY  r2, r0       # Write the [p, ?, ?]
26      ADD    r2, 1, r4
27      DCOPY  r4, r1       # Write the [p, s, ?]
28      LOAD   used_head, r0
29      ICOPY  r0, r1       # Load used_head to place it in 'next'
30      DCOPY  r0, r2       # Rewrite used_head to point to new element
31      ADD    r2, 2, r4    # Point to 'next ' field
32      ICOPY  r4, r2       # Load the ptr to the next free element into r2
33      DCOPY  r4, r1       # Write the [p, s, next]
34      DCOPY  r3, r2       # Write the free_head -> next free element
35      J      BasicStateA
36  PWasFound:
37      ADD    r3, 1, r2
38      ICOPY  r2, r1     # Load the stored secret.
39      PRINT  r1     # Output the secret.
40      ADD    r3, 2, r2  # Point r2 to the next field.
41      LOAD   free_head, r1
42      ICOPY  r1, r0   # Read the current pointer to the free list.
43      DCOPY  r2, r0   # Point next ptr of current triple to free list.
44      DCOPY  r1, r3   # Point free-head to current triple.
45      J      BasicStateA
46  InitializeFreeList:
47      LOAD   free_head, r0
48      LOAD    used_head+1, r0
49      LOAD    free_head, r1
50      DCOPY   r1, r0     # Set the free_head to point to the first triple.
51  LoopToInitialize:
52      ADD     r0, 3, r1        # Advance to the next element.
53      ADD     r0, 2, r0        # Advance to the next pointer inside.
54      DCOPY   r0, r1           # Write the next pointer.
55      ADD     r1, 0, r0        # Set current elt = next element.
56      SUB     r0, 5000*3+7, r2  # Have we initialized enough?
57      JNZ     r2, LoopToInitialize
58  TerminateFreeList:
59      SUB     r0, 1, r0
60      DCOPY   r0, r2 # Set the last next-pointer 0 to terminate
61                     # the free list.
62  WriteInitialFreeHead:
63      LOAD    used_head+1, r0
64      LOAD    free_head, r1
65      DCOPY   r1, r0     # Set the free_head to point to the first triple.
66      J       BasicStateA
67  OutputErrorMessage:
68      SUB     r0, r0, r0
69      PRINT   r0
70      J       BasicStateA
```

Listing 14: Working version of *variant2A.s*.

C.4. variant2B.s

```
1  const free_head 5  # Head of the freelist.
2  const used_head 6  # Head of the used list.
3      J       InitializeFreeList
4  BasicStateA:
5      READ    r0              # Read p
6      READ    r1              # Read s
7      SUB     r2, r2, r2      # Initialize a counter for number of elements.
8  CheckForNullSecret:
9      JZ      r1, OutputErrorMessage # Zero secret not allowed.
10     JZ      r0, OutputErrorMessage # Zero password not allowed.
11     LOAD    used_head, r3 # The list consists of [p, s, nxt] tuples.
12     ICOPY   r3, r3
13 CheckForPresenceOfP:
14     JZ      r3, EndOfUsedListFound
15     ICOPY   r3, r4          # Load 'p' of the entry.
16     SUB     r4, r0, r4      # Compare against the password
17     JZ      r4, PWasFound # Element was found.
18     ADD     r3, 2, r3       # Advance to 'next' within [p, s, nxt]
19     ICOPY   r3, r3          # Load the 'next' pointer.
20     J       CheckForPresenceOfP
21 EndOfUsedListFound:
22     LOAD    free_head, r3
23     JZ      r3, OutputErrorMessage # No more free elements available?
24     ICOPY   r3, r2        # Get the first element from the free list
25     DCOPY   r2, r0        # Write the [p, ?, ?]
26     ADD     r2, 1, r4
27     DCOPY   r4, r1        # Write the [p, s, ?]
28     ADD     r4, 1, r4
29     ICOPY   r4, r1        # Load the value of the old next pointer
30     DCOPY   r3, r1        # Write the free_head -> next free element
31     LOAD    0, r1
32     DCOPY   r4, r1        # Write [p, s, 0]
33     LOAD    used_head-2, r1 # Load the used_head (-2 as 2 will be added in the next step)
34 FindEndOfUsedList:
35     ADD r1, 2, r0 # go to the pointer
36     ICOPY   r0, r1 # load the target of the pointer
37     JNZ     r1, FindEndOfUsedList # Check if the end of the list is reached
38     DCOPY   r0, r2 # Write the beginning of the last cell into the pointer of the previous
39     J       BasicStateA
40 PWasFound:
41     ADD     r3, 1, r2
42     ICOPY   r2, r1      # Load the stored secret.
43     PRINT   r1      # Output the secret.
44     ADD     r3, 2, r2  # Point r2 to the next field.
45     LOAD    free_head, r1
46     ICOPY   r2, r4  # Store the old next element
47     ICOPY   r1, r0  # Read the current pointer to the free list.
48     DCOPY   r2, r0  # Point next ptr of current triple to free list.
49     DCOPY   r1, r3  # Point free-head to current triple.
50
```

```
51     LOAD   used_head-2, r1  # The used head is the start of the search
52 FindLinkingUsedElement:
53     ADD    r1, 2, r2    # Go to the next pointer
54     ICOPY  r2, r2       # Get the value of the next pointer
55     SUB    r3, r2, r2   # Compare pointer target with our cell number
56     ADD    r1, 2, r0    # Go to the next pointer of the element we just checked
57     ICOPY  r0, r1       # Use the previous next pointer to get to the next element
58     JNZ    r2, FindLinkingUsedElement # Have we not found the one targeting our cell?
59 FoundLinkingUsedElement:
60     DCOPY  r0, r4       # Set the earlier stored value as the new target
61     J      BasicStateA
62 InitializeFreeList:
63     LOAD   free_head, r0
64 WriteInitialFreeHead:
65     LOAD    used_head+1, r0
66     LOAD    free_head, r1
67     DCOPY   r1, r0     # Set the free_head to point to the first triple.
68 LoopToInitialize:
69     ADD     r0, 3, r1        # Advance to the next element.
70     ADD     r0, 2, r0        # Advance to the next pointer inside.
71     DCOPY   r0, r1           # Write the next pointer.
72     ADD     r1, 0, r0        # Set current elt = next element.
73     SUB     r0, 5000*3+7, r2  # Have we initialized enough?
74     JNZ     r2, LoopToInitialize
75 TerminateFreeList:
76     SUB     r0, 1, r0
77     DCOPY   r0, r2 # Set the last next-pointer 0 to terminate
78                    # the free list.
79     J       BasicStateA
80 OutputErrorMessage:
81     SUB     r0, r0, r0
82     PRINT   r0
83     J       BasicStateA
```

Listing 15: Implementation of the second variant with the linked list linked as described in [1].

List of Abbreviations

API - Application Programming Interface

BNF - Backus Naur Form

BNFC - BNF Converter

CLI - Command Line Interface

GUI - Graphical User Interface

IDE - Integrated Development Environment

IFSM - Intended Final State Machine

REPL - Read Evaluate Print Loop

Bibliography

[1] Thomas F Dullien. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing*, 2017.

[2] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[3] James F Ziegler and William A Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

[4] John F Reiser. Software emulating hardware for analyzing memory references of a computer program, April 6 2004. US Patent 6,718,485.

[5] Jennifer Paykin, Eric Mertens, Mark Tullsen, Luke Maurer, Benoıt Razet, and Scott Moore. Weird machines as insecure compilation.

[6] Stephen A Cook and Robert A Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

[7] ECMA. *ECMA-262: ECMAScript 2018 Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), ninth edition, June 2018.

[8] Alice E Fischer and Frances S Grodzinsky. *The anatomy of programming languages*. Prentice Hall, 1993.

[9] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[10] Zheng Gao, Christian Bird, and Earl T Barr. To type or not to type: quantifying detectable bugs in javascript. In *Proceedings of the 39th International Conference on Software Engineering*, pages 758–769. IEEE Press, 2017.

[11] Feross Aboukhadijeh. Javascript standard style. `https://github.com/standard/standard#why-should-i-use-javascript-standard-style`, 2019.

[12] Michael Rambeau Sacha Greif, Raphael Benitte. The state of javascript 2018. `https://2018.stateofjs.com`, November 2018.

[13] Aarne Ranta. *Implementing programming languages. An introduction to compilers and interpreters*. College Publications, 2012.

[14] Markus Forsberg and Aarne Ranta. The labelled bnf grammar formalism. *Department of Computing Science, Chalmers University of Technology and the University of Gothenburg*, 2005.

[15] BNFC Contributors. Bnf converter. `https://github.com/BNFC/bnfc`, 2019.

[16] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.