Symvex: A Symbolic Execution System for Machine Code

by

Mattias Rönn

LIU-IDA/LITH-EX-A-15/074-SE

2016-01-19
Final Thesis

Symvex: A Symbolic Execution System for Machine Code

by

Mattias Rönn

LIU-IDA/LITH-EX-A-15/074-SE

2016-01-19

Supervisor: Ulf Kargén {ulf.kargen@liu.se}
Examiner: Nahid Shahmehri {nahid.shahmehri@liu.se}
Abstract

This thesis is a part of an ongoing research project at Linköping University. The goal of the thesis work is to design and implement a prototype for a symbolic execution system that scales well with larger programs and is capable of performing symbolic execution on machine code. For this reason we have analyzed the current state of symbolic executors that are able to perform symbolic execution on machine code to see if we could use that implementation as base for our prototype. We wanted to know if any of the existing systems scaled well with large software. We found that neither of the existing systems worked well with the real life software in our evaluation. Furthermore, even if it would have been possible to fix one of the existing systems, the time required to figure out the faults in their implementation would most likely have been too great. For this reason we decided to create an implementation of our own from scratch. However, we did note that some approaches in the existing systems seemed to work better with large software. Specifically saving as little state as possible about the program seemed favorable. Armed with the knowledge gained from the analysis, we managed to implement a system that compared quite well with the existing systems. Our system was able to execute all the real-life programs used in our tests, but unfortunately had some issues with high memory usage for certain programs. In order to lessen the problem with high memory usage, we present and discuss several potential ways to mitigate this issue.
# Contents

1 Introduction  
1.1 Motivation .......................... 1  
1.2 Problem Formulation ................... 3  
1.3 Scope ................................. 3  
1.4 Assumptions on the Reader .............. 4  
1.5 Thesis Outline ........................ 4  
1.6 Abbreviations and Terms ................. 4  

2 Theory  
2.1 Bugs ................................ 6  
2.1.1 Classification ..................... 6  
2.1.2 Different Kinds of Bugs ............ 6  
2.2 Testing ............................... 7  
2.2.1 White Box Testing .................. 7  
2.2.2 Black Box Testing ................. 7  
2.3 Fuzz Testing .......................... 7  
2.3.1 Mutation Based .................... 8  
2.3.2 Generation Based ................. 8  
2.3.3 Issues with Fuzz Testing ............ 8  
2.4 Symbolic Execution ................... 8  
2.4.1 Small Example ..................... 8  
2.4.2 Symbolic Variables ................. 10  
2.4.3 Compiled Code ..................... 10  
2.4.4 Intermediate Representations ...... 11  
2.4.5 Constraint Generation .............. 11  
2.4.6 Symbolically Execute Code .......... 11  
2.4.7 Concolic Testing .................. 12  
2.4.8 Path Explosion ..................... 12  
2.4.9 Execution Strategies ............... 13  
2.5 SMT Solvers .......................... 14  

3 Method for Evaluation ................. 16  
3.1 Finding Existing Projects .............. 16  
3.2 Evaluation ........................... 16  
3.2.1 Test Setup ........................ 17  

4 Evaluation of Existing Systems ......... 19  
4.1 S2E ................................ 19  
4.1.1 Some Early Thoughts ............... 19  
4.1.2 Configuration Options .............. 19  
4.1.3 Method of Execution .............. 20  
4.2 FuzzBALL ............................ 20  
4.2.1 Some Early Thoughts ............... 20  
4.2.2 Configuration Options .............. 20  

v
CONTENTS

4.2.3 Method of Execution .............................. 21
4.3 BAP ............................................ 21
4.4 Pathgrind ........................................ 21
4.4.1 Some Early Thoughts ......................... 21
4.4.2 Configuration Options ....................... 22
4.4.3 Method of Execution ......................... 22
4.5 Other Tools ..................................... 22
4.6 Solvers in Use .................................. 22
4.6.1 STP ......................................... 22
4.6.2 Z3 ......................................... 23
4.7 Results ......................................... 23
4.7.1 Base Cost .................................... 24
4.7.2 Triangle Program ............................. 24
4.7.3 Making Symbolic Execution Fail .............. 25
4.7.4 Conclusion .................................... 26

5 Design and Implementation 27
5.1 Approach ........................................ 27
5.2 Top Level Design ................................ 27
5.3 Execution Tracing ............................... 27
5.3.1 Vextrace .................................... 29
5.3.2 Vexparse .................................... 30
5.4 Symbolic Execution of the Trace ............... 30
5.4.1 Memory Model ................................. 32
5.4.2 Approximations ............................... 34
5.5 Finding a New Path .............................. 36
5.6 Optimizing the Implementation ................. 36
5.6.1 VEX Resizing Data ........................... 36
5.7 Unrelated Constraint Elimination ............... 37
5.8 Controlling the Execution ...................... 38
5.9 Limitations ..................................... 38

6 Evaluation of Our Implementation 39
6.1 Test Setup ....................................... 39
6.2 Rerunning Evaluation Tests .................... 39
6.2.1 Base Cost .................................... 39
6.2.2 The Triangle Program ....................... 39
6.3 Real Life Programs .............................. 40
6.4 Breakdown of Execution Times ................ 42
6.5 Memory Breakdown .............................. 44
6.6 Problem with Diverging Paths ................ 44
6.7 Conclusion ..................................... 45
6.8 Further Work .................................... 45
6.8.1 Loop Detection .............................. 45
6.8.2 Iterative Constraint Solving ................. 45
# Contents

7 Conclusion and Discussion  
  7.1 Symbolic Execution and Larger Software  
    7.1.1 Existing Systems  
    7.1.2 The Preferable Approach  
    7.1.3 Status of Our Implementation  

8 Appendix  
  8.1 The Triangle Program
1 Introduction

This thesis work was conducted as part of an ongoing research project at Linköping University, aiming to develop methods for testing and analysis of binary code for security purposes. The goal of this thesis project is to develop a prototype symbolic execution system for use in future research efforts.

1.1 Motivation

The typical purpose of testing a piece of software is to make sure that it fulfills the requirement on the software [1]. However, from a security standpoint that is not the best approach, as regular testing will often not test the corner cases and unforeseen ways of using the software, which is not defined by the requirements. This can lead to missed security bugs in the software. An attacker could for instance exploit such bugs to gain remote code execution with the privileges of the attacked program. There exist several mitigations [2, 3, etc.] to combat the exploitation of security bugs. However, these mitigations should not be seen as a protection mechanism, but rather as a last line of defense in the event of a fault in the software. Therefore, finding the bugs and correcting them is a much better approach. There exist several techniques to find bugs in software, although the approaches varies depending on whether source code is available or not. The main focus of this thesis is when we only have access to compiled programs. In that case, a lot of information about the program is unavailable and thus, in order to find bugs, we need to utilize other techniques compared to when we have access to the source code. For instance, performing static analysis, a common technique used on source code, is harder as we do not have access to information about the control flow, data types for variables, nor variable names in compiled programs. Therefore, when we only have access to compiled code, we often have to resort to black box testing techniques. Black box testing techniques have the advantage that they do not need to take into account what language the program is written in, or what build system was used to build the software. Compiled code also have the advantage that we can perform dynamic analysis directly on it.

Black-box testing is performed by automatically generating test cases and checking if the program behaves as expected when run with the test cases. We could generate test cases manually by reasoning about the program, reading the manual, reading the specification, etc. However, it would be very hard to come up with test cases that gives us a high code coverage. Due to the difficulties in manually generating test cases, we may have to invest a fair amount of time in test generation, and that could also cost us, or a company, a lot of time and money. The difficulties in manually testing software can often account for up to 50% or more of the development cost [4]. Therefore automatic test case generation is of value. Automatic tools for testing compiled code with test case generation can be classified on a
scale from fully random based test case generation to not using any random input at all, or something in between.  

*Fully random based* tools, such as simple *fuzz testing*, can do a good job and are fairly easy to implement. However, a random based tool may have a hard time getting past code like that in listing 1. On a 32-bit system the comparison on line 4 would evaluate to true in 1 case out of $2^{32}$ possible cases.

**Listing 1:** Code example - Stopping a random input generator

```c
int main() {
    int i;
    scanf("%d", &i);
    if (i == 65) {
        "We don’t like the number 65 in this example"
        assert(false);
    }
    return 0;
}
```

One in about 4 billion is just not good enough. Often when we want to test a piece of software, we have access to some kind of well formed input. In this case we can mutate the input, e.g. by flipping random bits in the input, to try and find bugs in the software. This technique is often denoted mutation based fuzz testing. Another fuzzing technique is generation based fuzzing which works by utilizing knowledge of the input format of the program. This knowledge can then be used to generate input that is more likely to be a well formed input but containing unexpected data. The main limitation of fuzzing techniques is that they have no knowledge of the internal structure of the program and may thus be stopped by boundary values, exact comparisons, etc.

One way of solving this problem is to gather information about the code that is being run. Consider if-statements. Every if-statement contains a condition which will evaluate to true or false. If it evaluates to true we will take one path through the code and if false, then we will take the other path through the code. An if-statement is an example of a *branch point*. If we look at one branch point in the code, then we can try to find a value that will make the execution take the true branch, and one value that will make the execution take the false branch. If we then do this for every possible sequence of branch points in the program under test, then we will eventually have traversed every path in that program. This is a high level description of what is called *symbolic execution*. Being able to use symbolic execution
on all types of software, both our own and others\(^1\), with or without access to source code, would be of great value. However, some problems exist with symbolic execution today.

Symbolic execution is a great tool for testing small programs and functions. One problem is that the amount of paths in a program grows exponentially with the amount of branch points in the program. This problem is called *path explosion*. Large programs can have several million lines of code \(^5\) containing thousands if not hundreds of thousands branch points. This will make it practically impossible to get full path coverage in a large program. So in order to be able to run symbolic execution on a larger program, we need to make some tradeoffs. Instead of trying to achieve full path coverage, we want to achieve as high path coverage as possible given a timeframe. To do that, we can, for instance, use some heuristic to try to pick a path that will likely traverse through unexplored parts of the program. Another problem is that keeping track of all branch points and associated run time information may require a lot of memory. The main focus of this thesis is that latter problem; designing a symbolic execution system that can run for a long time without requiring too much memory, while still executing efficiently on larger programs.

1.2 Problem Formulation

The goal of the thesis work is to design and implement a prototype for a symbolic execution system that scales well with larger programs. The first step of the thesis work will be to evaluate existing symbolic execution systems for machine code. The approaches used by different systems will be compared, and the possibility to base our implementation on an existing system will be investigated. The design of our system will then be based on the findings from the evaluation. The questions this thesis seeks to answer are:

- How well do existing symbolic execution systems that runs on binary programs scale with larger software?
- Which design approaches are preferable for developing a symbolic execution system that scales with larger software?

1.3 Scope

We will limit the evaluation of existing systems to systems that are currently, spring 2015, in active development. Implementing a symbolic execution system is not a simple task. Therefore, the main goal of this thesis work

\(^1\)Using symbolic execution on software developed by someone else can possibly break some end use agreements, which needs to be looked into before testing their software. To our knowledge, there are no laws prohibiting us from using symbolic execution on any software, but since there are different laws in every country, care must be taken. In any case, any bug found should always be reported to the maintainer of the affected software.
will be to create a prototype for a symbolic execution system which can be extended later on. A secondary goal for this thesis work will be to optimize the implementation using established strategies from existing systems.

1.4 Assumptions on the Reader

This thesis is written with the assumption that the reader is at least familiar with programming in a C-like language, is familiar with operating systems and have a basic understanding about compiler technology. Most likely this applies to someone in their later years of a bachelor or beginning of a master in a Computer Science program.

1.5 Thesis Outline

The rest of the thesis will follow the following structure:

- Section 2, Theory, introduces all the relevant theory needed to understand this thesis work. However, a good understanding of computers and programming is assumed from the reader.

- Section 3, Method for Evaluation, discusses how the evaluation is performed of all the symbolic execution systems evaluated in this thesis.

- Section 4, Evaluation of Existing Systems, presents the results of the evaluation of the different existing tools for symbolic execution on machine code.

- Section 5, Design and Implementation, describes the how the symbolic execution system was designed and implemented.

- Section 6, Evaluation of Our Implementation, contains the evaluation of the symbolic execution system we have written.

- Section 7, Conclusion and Discussion, concludes this thesis with a discussion about the conclusions that could be made during this thesis work.

1.6 Abbreviations and Terms

The following is a list of abbreviations and terms used in this thesis. Some of them will be discussed in more detail in the report.

**Code coverage**

The percentage of the code that has been executed by test cases.

**Concrete value**

An actual value, the value stored in a memory area.
CPU
Central Processing Unit, the main processor in a computer.

CPU-time
The amount of time a task has been executing on the CPU.

Fork
Create a copy of the process that is executing, including its current state.

NP-hard problem
A problem that cannot be solved in polynomial time (with any known methods), but that can be solved in exponential time.

RAM
Random Access Memory, the main memory in a computer.
2 Theory

This section will introduce the theory needed to understand this thesis work. It will start by explaining in general terms what a bug is before moving on to testing techniques and more specifically topics related to symbolic execution.

2.1 Bugs

A bug is a fault in software or hardware, although we will focus on software faults. A fault is a defect in a software that may be triggered during the use of the software and may result in a failure.

2.1.1 Classification

Avizienis, et al. suggests that a fault can be classified in one of 31 subcategories [6], but for this thesis we are only interested in those which are introduced by a human during the development of the software. These faults are in most cases accidental without malicious intent. However, from a testing perspective, there is no difference between a fault of malicious or non malicious intent.

2.1.2 Different Kinds of Bugs

Bugs comes in many forms, some are harmless whereas others may crash a program, or possibly enable attacks on the program. As the focus of this thesis is security related, we are mainly interested in those bugs that may end up causing some kind of invalid memory accesses or memory corruption. Invalid memory accesses or memory corruptions can be used in information leak [7, 8] and remote code execution attacks such as buffer overflows [9]. However, it should be noted that not every security bug is a bug that affect memory, some stem from various logical errors that could for instance produce invalid results. Consider listing 2 and 3 as an example of a invalid memory access and memory corruption.

Listing 2: Code example - Memory corruption

```c
char buf[10];

// Will eventually go past end of buf
for (int i = 0; i < 100; ++i) {
    buf[i] = getc();
}
```

A memory corruption is when some data in memory is overwritten with something unexpected. Often this happen when we are not handling bounds checking correctly like in listing 2, or using a pointer that is uninitialized or invalid as in listing 3.

Listing 3: Code example - Variants of invalid memory access
```
char* uninitialized;
char* null = NULL;
// Kernelspace address on 32-bit
char* invalid_address = 0xC000000;
// We do not know where 'uninitialized' is pointing,
// likely a memoryspace not owned by us
*uninitialized = 'o';
// NULL-pointer dereference
*null = 'o';
// The same as uninitialized but accessing kernel space
*invalid_address = 'o';
```

Invalid memory accesses often result in crashes, or alternatively if the program does not crash, the data we accessed was not the data we intended to access.

### 2.2 Testing

As previously stated, there exist many different techniques to find bugs in software. Most techniques falls either into the category of white box testing or black box testing.

#### 2.2.1 White Box Testing

White box testing, also called structural testing, is a software testing technique where you take into account the internal structures of the software you are testing [10]. Normally this is the case when the tester have access to the source code of the software. However, that is not always true, some white box testing methods are applicable on compiled code. For example, in this thesis, we are exploring the use of symbolic execution on compiled code.

#### 2.2.2 Black Box Testing

Black box testing is when you only consider the input and output of the software under test. How the result was computed, or what happened during execution, is not of interest in black box testing [10]. One of the more famous black box testing methods is fuzz testing and its variants.

### 2.3 Fuzz Testing

Fuzz testing and symbolic execution (which is the thesis focus) are both techniques for automatic test generation. However, fuzz testing is still fundamentally different, so we will just briefly go through it here as the two techniques are often mentioned together. Fuzz testing is a testing technique that is actively used in the industry today. Fuzz testing is in its core a random based technique, although not every part of the generated input need
to be random. Two common fuzz testing techniques are mutation based and generation based fuzz testing.

### 2.3.1 Mutation Based
Mutation based fuzz testing starts with a well formed input and then randomly change(mutate) some part of the input at random places.

### 2.3.2 Generation Based
Generation based fuzz testing uses knowledge about the input format for generating test cases to the program under test. The knowledge could for instance be a grammar for the input. This knowledge can then be used to produce well formed input, but instead of using expected data, using extreme values such as very large integers, long strings, etc. as input.

### 2.3.3 Issues with Fuzz Testing
As we can see in listing 1, rather simple constructs may stop fuzz testing as it has no knowledge of the internal implementation of the code. If the grammar for some reason states that the number 65 should be in the input, then this would not be an issue. However, in other cases when the input is random, then fuzz testing would likely not execute the body of the if-statement. Therefore it is advantageous to be able to make decisions on the structure of code instead, even if it is compiled. How this is done will be discussed in the coming sections.

### 2.4 Symbolic Execution
Symbolic execution is a white box testing technique which works by marking certain variables we are interested in in the program as symbolic. We will start by going through a small example in order for the reader to easier understand what the goal is.

### 2.4.1 Small Example
Lets consider the code from listing 4.

**Listing 4:** Code example - Symbolic Execution

```plaintext
int main() {
    int i;
    scanf("%d", &i);
    i = i + 10;
    if (i >= 65) {
        // Path 1
        if (i < 66) {
            // Path 3
```

8
Here we have the variable 'i' defined on line 2. On the next line it gets assigned a value that is inputted from the user. Since it is controlled by the user that variable is interesting for test generation, but as explained before, just using randomly generated values as input will not be enough to test this program. What we can do here with symbolic execution is to mark 'i' as symbolic and follow 'i' through the program. Marking something symbolic is like saying that it is an unknown in an equation. On line 4, 'i' is updated to $i + 10$ and then on line 5 we reach an if-statement. The if-statement is interesting here as we have a choice of 2 branches, one if the predicate evaluates to true, and one if it evaluates to false. Lets name the true path 'path 1' and the false path 'path 2'. Each path here have a condition that needs to be satisfied in order to continue execution with that path, as shown in table 1.

<table>
<thead>
<tr>
<th>Path</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$i + 10 \geq 65$</td>
</tr>
<tr>
<td>2</td>
<td>$\neg(i + 10 \geq 65)$</td>
</tr>
</tbody>
</table>

If the condition for path 1 is satisfied, then we continue down that path. In doing so we encounter another if-statement. Since this if-statement is nested within the first if-statement, every previous if-statement must still evaluate to the same result in order for the execution to continue down a branch in the nested if-statement. A sequence of 1 or more dependent if-statements is called a *path condition*. We have the same situation with nested conditional
statements in the else-branch, so the full table, table 2, of path conditions for this small program is then the following.

<table>
<thead>
<tr>
<th>Path</th>
<th>Path Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>((i + 10 \geq 65) \land (i + 10 &lt; 66))</td>
</tr>
<tr>
<td>4</td>
<td>((i + 10 \geq 65) \land \neg(i + 10 &lt; 66))</td>
</tr>
<tr>
<td>5</td>
<td>(\neg(i + 10 \geq 65) \land (i + 10 == 100))</td>
</tr>
<tr>
<td>6</td>
<td>(\neg(i + 10 \geq 65) \land \neg(i + 10 == 100))</td>
</tr>
</tbody>
</table>

Not every path would be possible to reach. For instance, path 5 is not reachable as \(i + 10\) can not be both smaller than 65 and equal to 100. If a path condition is not solvable, then that part of the program is not reachable. Solving is done by using an SMT-solver. See section 2.5 for more information on SMT-solvers. By trying to solve every path condition, it is possible to generate an input for every reachable path in the program. By running every path in the program we can test if the program exhibits any unexpected behavior, like reaching the assert on line 9 in listing 4.

### 2.4.2 Symbolic Variables

A symbolic variable is a variable in a program that we have chosen to mark as symbolic. Marking something as symbolic means that we are not tracking its value, but rather saying that when variable 'x' is used, record its use as an expression. Marking something as symbolic can be done in several ways. One approach is to monitor the different ways the program can receive input, and mark the memory spaces(or variables) used to store the input as symbolic.

Which variables to chose to make symbolic varies on what parts of a program that you want to test. Often you want to mark all the input to some module or function as symbolic if you want to test that module or function. If you are testing a full program, then it could be interesting to mark those variables storing command line options or storing data read from files as symbolic.

### 2.4.3 Compiled Code

It is easy to think that when working with compiled code, only black box testing is available. However, any variable in a program is essentially just an address to some place in memory. There exist several tools for disassembling a binary file into assembly code, on which it is possible to implement symbolic execution. The downside is that assembly languages like x86 are very complex with a lot of side effects that are not immediately visible in
the syntax. One way around that is to lift the assembly code into an *inter-
mediate representation* which explicitly shows what is happening for each
line of code. This approach is chosen by FuzzBALL [11], parts of the S²E
execution engine [12], Mayhem [13], and others. Symbolic execution tools
that are working with source code are often also working with intermediate
representations, but instead of lifting assembly to it, compiling the source
code to it. This approach is used by KLEE [14], CUTE [15], and several
others.

### 2.4.4 Intermediate Representations

An intermediate representation, often abbreviated IR, is a representation of
the code that can be both independent of what language the code originally
was written in, and independent from the underlying architecture. When
compiling code, compilers often translate the code into an IR because then
they only need to implement an translation into IR for a language. The same
goes for translating the IR to a target architecture; only one translation from
IR to each architecture is needed. Symbolic execution is often done on an
IR as the IRs are quite explicit with what is happening, and can be language
and architecture independent. This makes the implementation simpler.

### 2.4.5 Constraint Generation

Symbolic variables play an important role when generating path conditions.
Every conditional statement in the code imposes constraints on the possible
paths. A constraint for a condition is generated by tracing the symbolic
variable that exists within the condition from the creation point up until
the condition. How to trace a variable is easiest shown with an example.

**Listing 5:** Code example - Trace a variable

```plaintext
1 int a, b;
2 b = a * 5; // 'a' is in expression, so follow b too
3 a = a + 10; // 'a' is now a + 10
4 a = a * 2; // 'a' is now (a+10)*2
5 if (a < b) {
6     // code
7 }
```

Tracing the code in listing 5 will yield the constraint \((a + 10) \times 2 < (a \times 5)\)
for the conditional on line 7. Basically you are tracing every variable that
has interfered in some way with a symbolic variable.

### 2.4.6 Symbolically Execute Code

Two methods for symbolically executing code is to either traverse the code
without actually executing it on the CPU or execute the code on the CPU
at the same time as we are collecting constraints. The trouble with not actually executing the code is that if we reach a path on which we cannot solve the constraint, then we cannot continue down that path [16]. However, if we would execute the code and reach a constraint that we can not solve, then we could change the symbolic values into concrete ones and continue with the execution. By doing this we lose some precision and may miss some paths. The following section describes this approach in more detail.

2.4.7 Concolic Testing

When running the program on the CPU, then we have access to all runtime values for the symbolic variables as well as other variables. Concolic testing is when we intertwine concrete execution with symbolic execution. Concrete execution here means that we execute with the real values, just like a normal execution. As mentioned above, an advantage with intertwining concrete execution is that if we cannot symbolically reason about something, we could use its concrete value instead. By making a variable or constraint concrete we will lose some precision. This is due to when using a concrete value in place of a symbolic expression we assume that the concrete value is a solution to the symbolic expression. However, the solution to the symbolic expression is typically a model representing the solution, and that model may represent multiple values. This may lead to missed paths, but that is still better than not being able to continue at all.

2.4.8 Path Explosion

An issue with symbolic execution is a problem called path explosion. Path explosion refers to the fact that a program will have a very huge amount of possible paths. This is due to that the amount of paths will grow exponentially with the amount of branch points in the code.

Listing 6: Code example - Nested statements

```c
int main() {
    int i;
    scanf("%d", &i);
    if (i >= 65) {
        if (i < 66) {
            // "We don’t like the number 65 in this example"
            assert(false);
        } else {
            //Some code...
        }
    } else {
    //Some code..
    }
}
```
Consider listing 6, here we already have 2 constraints yielding 4 paths in about 10 lines of code. In a large program there can also be several paths leading to the same code, further increasing the amount of paths we could explore. As an example, in a report about the number of lines of code in software published at ETH in Zürich in 2010, the browser Firefox [17] had about 2.5 million lines of source code at that time [5]. With a code size like Firefox’s is it not practically possible to explore every path.

2.4.9 Execution Strategies

So far, we have not addressed what is actually done when reaching a branch point in the code. Branch point here in this section refers to a branch point that includes symbolic variables. We have said that the execution should, if possible, continue down both paths. There exist some design philosophies to realize this.

Forking One approach is to fork the execution at every branch point. An issue with this approach is that with many branches we will have many active processes, which may slow down the overall system and consume a lot of RAM. It uses a lot of RAM because every process will require some memory in addition to the memory required for the execution state. However, since each process can keep its own state at all times and can run in parallel with the other processes, this approaches will execute fast. Basically this means that the approach favor time of execution rather than amount of used memory and it is the approach chosen by KLEE [14].

Minimal states Another approach that exists is to always execute from start to the end and only store small states. Using this approach we only need to store the constraints for each state in the execution. However, since we do not store the full state for each branch point, we need to execute the program from the beginning every time. Using this strategy will reduce the impact on RAM, but increase the execution time. This approach is chosen by SAGE [18], Pathgrind [19] and FuzzBall [11].

Selecting the Next State Regardless of the execution strategy used, we need to address the problem of selecting the next state to continue with once a path is explored. On the one end we could follow a BFS pattern and explore all branch points in the beginning of the program and slowly making our way deeper into the code. On the other end, we could follow a DFS pattern for choosing the next path to quickly reach deep into the program, but following only a narrow path.

The choice of strategy varies with what you want to do. Using something BFS-like can lead to the issue that you get stuck for a long time in the early
parts of the program, whereas when using DFS you may get stuck in things like loops. However, in the end both BFS and DFS would execute the same amount of paths as they are both complete search algorithms, which also means they both suffer from the path explosion problem. Therefore, some implementations use heuristics to guess which state is most beneficial to continue with. For instance, the symbolic execution system SAGE from Microsoft Research prioritizes paths that has previously lead to the exploration of new code [18].

2.5 SMT Solvers

A SMT-solver (Satisfiability Modulo Theories solver) is a solver that given a formula computes whether it is satisfiable or not. Given that the formula is satisfiable, it can also return a solution to the formula. As the name suggest, these solvers are based on various theories. These theories are different approaches to various problems. Two of these theories are:

Linear Arithmetics
It is the theory behind linear arithmetics with integer or real numbers.

Bitvectors
It is the theory of arithmetics over bitvectors. For instance, a 32-bit integer can be viewed as a bitvector of length 32. It is different from normal arithmetics in the sense that it works with fixed width numbers and thus exhibits behaviors like overflow and underflow.

Some other theories that exist are the empty theory, nonlinear arithmetic, arrays, data types, and quantifiers.

One high level simplified approach used for solving SMT queries is to first consider the problem as a satisfiability problem using boolean expressions [20]. Given formula (1) below, we can substitute \(a \geq 3\) with \(p_1\) and \(a \geq 5\) with \(p_2\). This substitution gives us formula (2).

\[
\langle \neg(a \geq 3) \land (a \geq 3 \lor a \geq 5) \rangle \tag{1}
\]

\[
\langle \neg(p_1) \land (p_1 \lor p_2) \rangle \tag{2}
\]

Formula (2) can then be fed into a SAT-solver (satisfiability-solver) to check if it is even possible to solve. Here a possible solution model is \(p_1 = false\) and \(p_2 = true\). In other words, we are looking for a solution which satisfies equation (3) and (4).

\[
p_1 = false \iff a < 3 \tag{3}
\]

\[
p_2 = true \iff a \geq 5 \tag{4}
\]

This solution model is then passed onto the appropriate theory, in this case arithmetic theory. In this case there exists no valid solution to the model,
but given another input there might have been. This is a basic principle behind a SMT-solver, and should be enough in order to follow the thesis [20]. In practice, modern solvers uses techniques such as bit-blasting to replace the operators with their bit-level circuit equivalents, transforming the problems into boolean satisfiability problems. This will create small sub problems from the initial problem that can be more efficiently solved by SAT-solvers [21, 22].

There exist some standard languages for communicating with SMT-solvers. They are used to get a standard interface to a solver, although, most solvers do not support all formats. Three common formats are SMT-lib v2.0 [23], CVC [24], and DIMACS [25].
3 Method for Evaluation

This section describes how the analysis of the existing systems will be performed.

3.1 Finding Existing Projects

In order to find some material to perform the analysis on, the first step of the study was to find out what symbolic execution systems exist today. More specifically, we are searching for those systems that support running programs without modifying the program’s source code; it should be enough to only have the binary version of a program available. As a starting point, we investigated some well known projects in symbolic execution, namely KLEE, Mayhem, and S2E. All these projects have articles written about them by the developers and may contain references to other similar projects. Also, we checked if these well known projects have been cited by some less known project. For instance, ACM lists for each article in their database which articles the article cites and which articles the article is cited in. Also, searching in article databases such as Springer may yield some results. Some projects do not have an academic backing and these projects are probably easier found through Google and similar search engines. This is because people outside academia are not as prone to write an article about their work.

3.2 Evaluation

For the evaluation part we need some kind of unified approach that puts the symbolic execution system under equivalent tests. An issue here is that the different systems support many different options, which may not correlate nicely with another option in another system. Another issue is that not all options are documented. Such options will not be considered in tests, even if they may have been beneficial for the particular symbolic execution system. As we are only interested in those systems that can run without source code access, all other symbolic execution systems will be disqualified.

The interesting things to measure in these tests are memory usage and time consumption. It may be the case that the time consumption is too big to reasonably execute the program. In such cases we will limit testing time to 20 minutes. If an input file is needed the script in listing 7 is used to generate a random input file.

Listing 7: Code example - File generator

```bash
#!/bin/bash
dd if=/dev/urandom of=$1 bs=1 count=$2
```

In order to monitor peak memory usage, a script called ‘memusug’ will be used [26]. In order to avoid swapping and/or thrashing, if the program takes
more than 10GB of memory we will shut it off. Even if it would be possible
to continue running, swapping have a huge affect on performance making
it almost impossible to continue. Accessing data on a hard drive takes in
the 10 millisecond range, whereas accesses to RAM is typically in the 100
nanosecond range. Therefore, we could experience up to \(10^5\) time slower
execution if allowing swapping to be used.

All the tests will run on a computer with Linux Mint 17.1 with the
following specification:

**Table 3:** Computer used for tests

<table>
<thead>
<tr>
<th>Type</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel®Core™i7-3770 CPU at 3.40GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>16 GiB at 1600MHz</td>
</tr>
<tr>
<td>Harddrive</td>
<td>500GB, 7200rpm</td>
</tr>
</tbody>
</table>

### 3.2.1 Test Setup

Testing of the existing systems and the system we created is split into two
parts. First we measure the performance with a minimal program containing
an empty main function. With this test we can measure the startup
cost of each symbolic execution system. In the second part we will test with
programs that are used ‘in real life’. For the real life programs we will use
gzip [27], tar [28] and convert [29]. Gzip and tar are selected as they are
commonly used software for archiving files are simple to control for how long
they will execute by varying file sizes. Convert is selected to have a computationally heavy program to test with. It is a program converting images
from one format to another. The programs are run with the commands as
shown in table 4.

**Table 4:** Input to programs under test

<table>
<thead>
<tr>
<th>Program</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>-k -d input_file.gz</td>
</tr>
<tr>
<td>tar</td>
<td>xf input_file.tar</td>
</tr>
<tr>
<td>convert</td>
<td>input_file.png output_file.jpg</td>
</tr>
</tbody>
</table>

As backup for cases where we could not use ’real’ programs for some rea-
son, we used as a fallback option a classic program in testing; the triangle
program [30]. The original triangle problem is as follows:

"The program reads three integer values. The three values are interpreted
as representing the lengths of the sides of a triangle. The program prints a
message that states whether the triangle is scalene, isosceles, or
equilateral"[30]
The triangle program is useful due to having many different, but simple to understand, cases.

In order to be able to compare the different programs we will measure the amount of memory used and the number of paths explored, both in absolute number, and per time unit. The amount of memory used is interesting since we want to keep the memory usage as low as possible. The number of paths in absolute numbers is interesting since various symbolic execution systems use different kinds of approximations which may lead to missed paths. The time used per path is interesting in the same way as the memory usage; to be efficient, the time a tool spends on each path should be as low as possible.
4 Evaluation of Existing Systems

In this section we will first present those tools that we found and evaluated, before presenting the actual results.

4.1 S2E

$S^2E$, Selective Symbolic Execution, is a symbolic execution platform that is based on KLEE [14] and QEMU [31]. It is using QEMU to virtualize a host operating system and translating the basic blocks that QEMU generates into LLVM [32] IR. Then it uses KLEE for the symbolic execution. KLEE is designed to work with LLVM IR. A downside is that $S^2E$ is using a fairly old version of KLEE and thus also LLVM IR. KLEE uses STP [33] to solve the generated constraints. $S^2E$ only exposes a subset of all options KLEE offers, however, since $S^2E$ is using binary translation into LLVM IR, $S^2E$ supports symbolic execution on compiled programs. $S^2E$ is developed by a research group at École Polytechnique Fédérale De Lausanne, although there has not been any commits to its repository in the last 6 months. Since there have not been any official statement that the project is dead, we will consider it still live, but with reservation.

4.1.1 Some Early Thoughts

$S^2E$ promises much and from the start looks like a really good project, though the documentation could be much better. We had to spend a considerable amount of time reverse engineering $S^2E$ just to figure out how to write the configuration file needed so $S^2E$ could start without warnings.

4.1.2 Configuration Options

$S^2E$ have options for where to allow for forking, however, enabling those options did not seem to have any effect. The possible options are to only allow forking in usermode, or in the process code section. Those would have been useful in order to select which modules of a program to test. Another useful option that exists but seem not to have any effect is the ability to limit the memory usage. To be able to use concolic testing with $S^2E$ you need to specify `-concolic`. This will make $S^2E$ augment symbolic variables with concrete values, so $S^2E$ does not need to call the constraint solver for every condition. This is since the concrete values can be used to know which path will be taken at a branch point. The concolic flag will also make $S^2E$ use DFS rather than BFS for state search and make $S^2E$ run the program to completion every time [34]. This means that $S^2E$ will never call the constraint solver during execution.
4.1.3 Method of Execution

S²E works by running a program in QEMU. Then for every basic block that QEMU translates, S²E will lift it to LLVM IR and do symbolic execution on the LLVM IR. QEMU can then continue to translate the basic block into the target architecture machine code. This means that the LLVM IR is never translated into machine code. Since S²E is using KLEE as its symbolic execution engine, S²E is forking at every branch point that the program encounters. The new entity resulting from the fork is referred to as a state. This means that every branch point will have a relatively substantial memory impact. Each forked execution will then check if the new path is feasible to continue with. If the new path is unexplored and it is possible to solve the path constraint, S²E will continue with that path. S²E will not continue with the new path right away, but rather when that state is chosen as the next to be executed. The default for concolic mode is that S²E executes one state from start to end before choosing a new state to continue with. However, it is possible to chose another strategy for choosing states, such as a breadth first manner instead of the depth first manner that is the default.

4.2 FuzzBALL

FuzzBALL is a symbolic execution engine that is executing on Vine IL [35], that was developed with the BitBlaze project [36]. Vine IL is a intermediate language that does not contain any implicit side effects for any expressions. Before doing symbolic execution, FuzzBALL decompiles x86 into Vine IL. Vine IL supports a subset of the x86-family of instructions. However, it lacks support for SIMD-instructions that every new Intel processor today supports as it is part of x86-64, restricting usage with modern programs. FuzzBALL is maintained by Stephen McCamant at University of Minnesota. FuzzBALL is still receiving sporadic updates so we will consider it as an active project.

4.2.1 Some Early Thoughts

FuzzBALL presents us with a lot of configuration options, though the documentation for most options consists of the small help message printed with the option if you invoke FuzzBALL with –help. While doing some initial testing, many options did not seem to have an effect, or required, without stating it, some other option to be enabled to work.

4.2.2 Configuration Options

In FuzzBALL it is possible to set a start and an end address for symbolic exploration, although we did not use it as it would require us to manually figure out the start and end for the program under test. The start address is a point in the program from where FuzzBALL should start to symbolically
execute the program. No symbolic variable may have been referenced before that point. The end address is where FuzzBALL should stop the symbolic execution. It is also possible to configure FuzzBALL to use different solvers. Per default it uses STP, though FuzzBALL can be configured with Z3 [37], CVC [38], and others.

### 4.2.3 Method of Execution

FuzzBALL works by lifting the code into VEX, an intermediate language used by Valgrind [39], and then translating the VEX code into VINE IL. Symbolic execution is then performed on the VINE IL. This is similar to how concolic testing was described in section 2.4.7, although to perform concrete execution of the code, the VINE IL is interpreted to the target architecture rather than running directly on the CPU.

### 4.3 BAP

BAP [40] - Binary Analysis Platform, is technically not a symbolic execution engine by itself, but rather a platform designed to simplify analysis of binary programs. Most notably, BAP consist of tools for transforming binary code into BIL, an intermediate language designed with symbolic execution in mind. In earlier versions, BAP contained tools useful for symbolic execution, although they have been deprecated for a to us unknown reason. The symbolic execution system Mayhem [13] is using BIL as its language to perform symbolic execution on. BAP is an interesting project for us, but due to the discontinuation of the tools for symbolic execution, the authors unfamiliarity with OCAML(which BAP is written in), and the lack of documentation for BIL, we did choose not to continue with BAP for further reviews. Considering learning OCAML for BAP as a negative thing is purely due to lack of time for this project, and not something that is inherently bad with BAP.

### 4.4 Pathgrind

Pathgrind [19] is an automated path explorer for compiled programs based on Sogeti ESEC Lab’s Fuzzgrind [41]. Pathgrind is using symbolic execution to generate new input files to programs. It is only available for 32-bit programs. Pathgrind is using Valgrind to lift the binary code into an intermediate language.

#### 4.4.1 Some Early Thoughts

Pathgrind is not straight forward to get started with the \(-help\) option does not really help and gives no hint on how to actually run anything. However, the author of Pathgrind has written a blog post to help people to get started [42]. The downside is that that blog post seem to be all documentation there
is for Pathgrind. Although Pathgrind only supports 32-bit programs, it does not produce an error if run with a 64-bit program, instead reporting that there were no paths to explore. This can fool a user into believing that Pathgrind works with 64-bit programs, even though it does not.

4.4.2 Configuration Options
Pathgrind does not expose many configuration options. Those that exist are program arguments, max and min bound for the symbolic execution, and max number of paths. There also exists a checker option, though its use is undocumented.

4.4.3 Method of Execution
Pathgrind works by instrumenting the lifted code that Valgrind produces. First Pathgrind runs the program with an initial input file in Valgrind. Valgrind lifts the binary code into an intermediate language called VEX one basic block at a time, making that basic block available for instrumentation. Pathgrind then does symbolic execution on the VEX code following those instructions that are interesting, i.e. instructions handling symbolic data. Pathgrind marks data read from the input file as symbolic. Once the entire program is executed, Pathgrind chooses a new path to explore and solves its path condition to obtain a new input file that is used for the next execution. Just like FuzzBALL, every execution starts from the beginning running to the end of the program.

4.5 Other Tools
Other tools like Mayhem and SAGE would have been interesting to test, but those tools are proprietary and not available for public use. KLEE, CUTE, and others were disqualified due to their requirement to have the program source code available.

4.6 Solvers in Use
Two solvers were the most common encountered during our testing, namely STP [33] and Z3 [37]. Both are released under the MIT License [43].

4.6.1 STP
STP is in our experience the most commonly used solver. It is the default solver in all tested projects in this thesis. STP supports bitvector theory and array theory, making it good for use with symbolic execution. It is possible to interface with STP through native interface or using SMT-Lib V2 or the CVC language.
4.6.2 Z3

Z3 is a theorem prover developed and maintained by Microsoft. Z3 is also possible to use with FuzzBALL. Z3 supports several theories, including bitvectors, arrays and linear and non-linear arithmetics. It is possible to interface with Z3 through a native interface or using SMT-Lib V2 or the DIMACS language.

4.7 Results

Comparing the performance of $S^2E$, Pathgrind and FuzzBALL gave some mixed results. Notably, FuzzBALL and Pathgrind failed to run properly on the ‘real life’ programs we had planned to use for testing, see chapter 3. $S^2E$ only manages to run till completion for really small files, already at 1 symbolic byte we had execution times at over 1 minute and memory usage at over 2GB. With more than 10 bytes of symbolic data, $S^2E$ suffered within minutes from swapping on a computer with 16GB memory and testing was aborted. $S^2E$ spawns several thousand states each minute, and each unterminated state consumes somewhere around 0.5 to 1 megabyte of memory. In Pathgrind it was also difficult to see whether the system worked at times, as Pathgrind never reported errors. Pathgrind just silently quits and reports 0 paths explored when running into something it cannot handle. Therefore, we had to skip the programs we had planned to use for testing and we decided to fall back on the triangle program. this also made it easier for us to have full control over the tests.

FuzzBALL, $S^2E$ and Pathgrind is run with the following commands:

**Listing 8:** Commands for FuzzBALL and $S^2E$

```
1 # S2E command line
2 LD_PRELOAD=./init.env.so ./program \ 
3 --concolic --select --process --code
4 ./s2ecmd kill 0 "done"
5
6 # Fuzzball command line
7 time memusg ./fuzzball --solver --stats --trace --solver \ 
8 --trace--iterations --linux--syscalls --time--stats \ 
9 --periodic--stats 10000 ~/triangle/a.out \ 
10 --smtlib--solver--type z3 \ 
11 --solver smtlib --solver--path 'which z3' \ 
12 --solver--timeout 300 \ 
13 --timeout--as--unsat --save--solver--files \ 
14 --no--fail--on--huer \ 
15 -- ~/triangle/a.out file
16
17 # Pathgrind command line
18 time memusg fuzz/fuzz.py triangle
```
4.7 Results

FuzzBALL is run with the Z3 solver here as it performed best. Running FuzzBALL with STP was slightly slower.

4.7.1 Base Cost

All three tested symbolic execution tools do a translation into another language, and $S^2$E also runs in a virtual machine. Therefore it is interesting to see how much time will be spent for just starting up. Also, the base memory usage will be presented. It is important to note, however, that the startup time penalty is only incurred at the first run for $S^2$E, but every time for FuzzBALL and Pathgrind. This is because the biggest time of the $S^2$E startup is spent starting up the VM, whereas FuzzBALL and Pathgrind reruns the entire program from the start for each new input to the program.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Time(seconds)</th>
<th>Memory(Kilobytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^2$E</td>
<td>20.4</td>
<td>2112044</td>
</tr>
<tr>
<td>FuzzBALL</td>
<td>2.2</td>
<td>62736</td>
</tr>
<tr>
<td>Pathgrind</td>
<td>0.02</td>
<td>77112</td>
</tr>
</tbody>
</table>

Table 5: Base cost

We can see in table 5 that to start up $S^2$E a fair amount of work has to be done. Not shown in the table above, the startup time for $S^2$E if we do not have access to kernel virtualization [44] is about 10 times slower compared to if we can use it. Most of the time used for FuzzBALL is in the loading and linking part of the startup of a program. Pathgrind excels in this test case, with next to none startup cost.

4.7.2 Triangle Program

As explained before, we decided to fall back on the triangle program. However, we decided to modify the triangle program slightly. We added the possibility for the program to return ”invalid” for input that does not represent a triangle. This was to get a meaningful result for these inputs. The source code for our program can be found in Appendix A. The base line for FuzzBALL, Pathgrind and $S^2$E without any symbolic input are as follows:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Time(seconds)</th>
<th>Memory(Kilobytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S^2$E</td>
<td>21.9</td>
<td>2137340</td>
</tr>
<tr>
<td>FuzzBALL</td>
<td>3.79</td>
<td>62736</td>
</tr>
<tr>
<td>Pathgrind</td>
<td>0.03</td>
<td>78060</td>
</tr>
</tbody>
</table>

Table 6: Triangle program base results
As we can see, there is not that much difference compared to the empty program, which is expected as we have not added that much new code. If we continue with running with symbolic data, then we get more interesting results.

Table 7: Triangle program results

<table>
<thead>
<tr>
<th>Tool</th>
<th>Time (seconds)</th>
<th>Peak memory usage (Kilobytes)</th>
<th>Paths explored</th>
<th>Average time per path (seconds/path)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S²E</td>
<td>26.5</td>
<td>2188500</td>
<td>26</td>
<td>1.02</td>
</tr>
<tr>
<td>FuzzBALL</td>
<td>71.4</td>
<td>79176</td>
<td>26</td>
<td>2.75</td>
</tr>
<tr>
<td>Pathgrind</td>
<td>14.3</td>
<td>78104</td>
<td>25</td>
<td>0.56</td>
</tr>
</tbody>
</table>

Even though S²E has a quite big startup cost, S²E will be faster than FuzzBALL after a certain point. This is likely due to that QEMU can translate and run the code much more efficiently than FuzzBALLs can interpret VINE. Pathgrind is taking the victory here as well in terms of speed, however, it is reporting 1 less path explored. This is probably due to some approximation Pathgrind is doing, making Pathgrind miss one path.

4.7.3 Making Symbolic Execution Fail

As we have seen we can execute the triangle program without too much hassle. During the testing, by accident, we did leave a debug printf statement in the code containing a printout of a symbolic variable. This mistake is a good example to show how ‘easy’ symbolic execution could get stuck at something that is for us uninteresting. For example, Pathgrind now needed about 8 minutes and 30 seconds and a memory usage around 80MB to finish executing all paths. That single statement slowed Pathgrind down about 8 times. The total amount of paths that Pathgrind explored was 328, making it about 1 path per 1.5 seconds.

FuzzBALL on the other hand, could not execute every path now, FuzzBALL was left running during one weekend (about 60 hours). In that time, FuzzBALL could only execute 277 paths. That is only about 4 paths an hour compared to about 1 path every 3 seconds in the case without the printf. The time to solve 1 path condition many times ended up in the several minute range. Why Pathgrind could solve its path conditions so much faster than FuzzBALL, even though they operate in similar ways, is to us unknown. It could be due to FuzzBALL possibly generating much bigger path conditions or solving unnecessary path conditions or some other reason we have not considered.

S²E solves its path conditions incrementally, as S²E has all information about the path in memory, including previously solved constraints. Therefore, S²E manages to execute about 150 paths in 2.5 minutes, making it slightly faster than Pathgrind. However, due to that S²E forks for each
branch and thus save its state for each new path, the memory usage is a problem. Already after 2 minutes 30 seconds it consumes about 7GB of memory, and the memory usage increases linearly with the amount of paths. We didn’t reach a memory usage high enough to see the effects of swapping for this program due to triggering an assert in $S^2E$, however, running gzip with a big enough input file made the computer freeze once swapping started. Normally a computer does not completely freeze due to swapping. The problem was likely due to the combination of high memory and CPU usage and hardware virtualization.

4.7.4 Conclusion

All tested systems are interesting projects. A major downside for FuzzBALL and Pathgrind is that they do not seem to work well with ‘real’ programs. Also, the fact that FuzzBALL and Pathgrind is lacking support for some sets of instructions that is likely to exist in modern programs is worrying. Notably, neither of them have support for any x86-64 instructions. Fixing this limitation would, just as in the case for BAP, require too much time in getting to know the unfamiliar environment of OCAML(for FuzzBALL) and a unknown code base. Therefore will we not move on with FuzzBALL and Pathgrind, even if they have several good properties, most notably their low memory usage.

$S^2E$ on the other hand performed fast, although it consumes so much memory so that it can not really be used efficiently for large programs. We also encountered a number of weird bugs and assertions, which combined with the lack of documentation and a to us foreign code base makes us feel that basing our implementation on $S^2E$ would require too much time to even get started.

Even if we will not base our implementation on neither FuzzBALL, Pathgrind or $S^2E$, there are several lessons to be learned. If we want to work with large programs, the approach chosen by FuzzBALL and Pathgrind is better as it does not require as much memory. $S^2E$ showed us that a startup overhead may not be bad, as long as translating to the target architecture is fast.
5 Design and Implementation

This section will describe what approach was used for coming up with a design for our symbolic execution system. The remainder of the section describes the actual design and implementation of our symbolic execution system. Our implementation will be referred as Symvex.

5.1 Approach

To decide which tools to use for the implementation an evaluation was performed. This evaluation can be found in chapter 4. The result of the evaluation showed that we need to implement a symbolic execution tool of our own.

The implementation will be divided into two parts. The first part will be implemented by the supervisor of this thesis, and the second part will be implemented by the writer of this thesis. The first part handles recording an execution, and the second parts handles the symbolic execution.

5.2 Top Level Design

The top level design is illustrated in figure 1. The design is aimed at dividing the implementation into separate units so one unit can easily be exchanged if needed at a later stage.

This top level design can be broken down into 3 main parts.

1. Trace the execution.
2. Process the trace.
3. Find a new path.

During the execution, one pass through all 3 parts is labeled one iteration. Tracing the execution is the part where the program under test is executed and all instructions that were executed are recorded. How that is done is presented in chapter 5.3. Processing of the trace is the part where each recorded instruction is symbolically executed. This part is presented in 5.4. The last part is were the results of the symbolic execution have been recorded and we need to find a new input to drive the execution forward. The last part is presented in chapter 5.5.

5.3 Execution Tracing

This system is aimed at x86 and x64_64 [45] assembly. One issue with working with x86 assembly is that the instructions can have side effects that are not visible. For instance, the instruction set reference from Intel [46] states for the add instruction:
5.3 Execution Tracing

The ADD instruction performs integer addition. It evaluates the result for both signed and unsigned integer operands and sets the OF and CF flags to indicate a carry (overflow) in the signed or unsigned result, respectively.

The SF flag indicates the sign of the signed result. [46]

For this reason, we need a tool for lifting assembly code into something that is easier to work with. The implementation of the lifting tool is made by
the supervisor of this project. It is made from two parts.

**Vextrace** Vextrace is a tool that records the execution into a trace file.

**Vexparse** Vexparse is a source code package for parsing a trace file.

### 5.3.1 Vextrace

Vextrace is a tool written for use with Valgrind [39]. Valgrind is a tool for instrumenting code as it is being executed. As part of the instrumentation, the native code is lifted into an intermediate language called VEX [39]. The advantage of lifting the code into an intermediate language is that most side effect are made explicit. Exposing the side effects makes the symbolic execution easier later on. This technique was used by Pathgrind and FuzzBALL with mixed results. One big advantage as we saw in the evaluation is that the base memory cost is fairly small.

FuzzBALL is using an online decision procedure, meaning FuzzBALL solves the constraints as they come in order to decide which branch to take. This complicates the execution of the programs under test, but simplifies deciding the next path to continue with. However, FuzzBALL was very slow, likely because FuzzBALL executes the program under test by interpreting the intermediate representation. Using interpretation is generally slower than compiling the intermediate representation and running the compiled code, a method used by Pathgrind. Another method Pathgrind uses is an offline decision procedure. Using an offline decision procedure means that Pathgrind just serializes the lifted code into a text file, and does the symbolic execution in another program. By using an offline decision procedure, Pathgrind can execute the programs faster. However, Pathgrind needs to spend more time between iterations. The approach chosen by Pathgrind seems to work much better, and therefore we will choose that path too. However, we will not translate the binary VEX into a text representation, but rather keep it as is for more efficient processing.

While it is possible to implement everything in the Valgrind tool, Valgrind only exposes a limited subset of the C standard library and do not allow linkage with external libraries. Therefore we will keep the Valgrind tool as small as possible, in order to make the rest of the implementation easier. For this reason we will take the VEX-code and serialize it into a trace file. Since code normally do not contain the actual data but rather instructions doing some operation on the operands, during the serialization step we will also include concrete values of read and written data. Also, during the instrumentation we can include custom instructions to help with the symbolic execution, such as indicating at which memory addresses data from files are stored.
5.3.2 Vexparse

Vexparse is the other half of the execution tracing tool. It is responsible for deserializing the serialized VEX-code from the trace file. It is the interface to the recording tool and is what the implementation created by the authors of this article will use.

5.4 Symbolic Execution of the Trace

The symbolic execution of the trace is done by considering every VEX instruction one by one. For each iteration, we can build a graph over the data flow in the program. We are only tracking the symbolic data in the data flow graph. The rest of the data is viewed as concrete values.

Listing 9: Code example - A simple program

```c
int a, b, c;
a = file_read();
c = file_read();
b = a * 5;  // 'a' is in expression, so follow b too
a = b + a;
a = c − a;
if (a < b) {
  // code
}
```

Given the code in listing 9 the corresponding data flow graph would look like the graph in figure 2. The graph in generated by following the symbolic

![Data flow graph](image)

**Figure 2**: The data flow graph corresponding to the code in listing 9. The values in brackets correspond to the source line they are created. The values in parenthesis correspond to the latest definition of a symbol.
data in the following manner. The numbering below corresponds to the
source lines in listing 9.

1. Definition of variables does not exist in the underlaying assembler
code, they are just a region somewhere in memory.

2. When we read data from a file to memory, the memory region where
the data from the file is stored is marked as symbolic. A memory
region marked as symbolic is considered 'tainted'.

3. Same as for line 2, we are now tracking 2 symbolic variables.

4. Since 'a' is used in the calculation, the memory region for b is tracked
as well. 'b' is not an symbolic variable, it is just tracked memory. In
the data flow graph, that memory region is represented as an opera-
tion, with backlinks to the operands.

5. In this expression we both have 'a' and 'b'. Looking in the graph,
we can find the latest definition of them and create backlinks to 'a'
and 'b'. Since we are overwriting the memory region for 'a', we are
overshadowing the old definition of 'a' with this new one. This means
that the next time 'a' is referenced, 'a' will look like ”'b' plus the old
'a' ”.

6. We always use the last known definition of a variable, or memory
region, so we get the symbolic value of 'a' from line 5, and the symbolic
variable 'c'. This operation overwrites 'a'.

7. The if-statements on the following lines are not represented in the data
flow graph.

This data graph represents the dependencies between the operations on sym-
bo lic variables executed in the program. Whenever during the processing of
the trace when we encounter a branch point, then we take last known defini-
tion of the variables in the condition and transforms them into a SMT-query
together with the conditional operator. The if-statement on line 7 in listing
9 would be transformed into the following query:

Listing 10: Code example - A SMT-query

(assert (bvult (bvsub (bvadd (bvmult 5 a) a) c) c))

This query represent the condition for the path so far. It can be read like,
”make sure that the relationship between a and c follows this model”. Once
we have a SMT-query for a branch point, it is stored in a tree, representing
all different execution paths in the code. It is an idea we borrowed from
FuzzBALL. Each branch point we pass during the symbolic execution will
be added to the tree, and later used for finding new paths to explore.
5.4 Symbolic Execution of the Trace

5.4.1 Memory Model

Our model of the memory consists of the data flow graph together with several structures handling parts of the execution. An illustration of our memory model can be seen in figure 3. Together they form our shadow memory. It is the memory used in the symbolic execution and is a model of the memory of the program under execution. The various structures handles different parts of the program memory and as well as Valgrind specific data storage constructs.

**CPU Registers** This part of our shadow memory models the registers found in the CPU, such as stack pointers, general purpose registers, etc. Valgrind is modeling them with a separate set of load and store instructions, 'Get' and 'Put'. They are mostly used for temporarily holding data.

**Temporary Memory** This part models the temporary variables used by VEX. When translating a basic block, VEX will use a unique set of tempo-
rary variables for that block. These variables are used to hold intermediate results and data that have not yet been written back to memory or some register.

**Tainted Memory Handler** The tainted memory handler is used to keep track of what memory regions are currently tainted. When a currently tainted region is overwritten by non-tainted data, it is removed. The tainted memory handler also manages the mapping between a tainted memory region and the file where it was read from.

**The Memory** During the symbolic execution, whenever we write or read from memory, we need to keep track of the physical memory so we do not lose track of data. Whenever a store to memory happens where the data stored is symbolic, we write it to a separate memory store. When a read from memory happens, we check if we are tracking that address in the memory store. If we are tracking the address, then we return what is stored in our memory, otherwise the concrete value is returned. In our ‘memory store’ we keep a handle to a graph node representing a model of the data stored at a certain address. Refer to the example in section 5.4. We view the memory as a series of 8 byte chunks consisting of 8 ‘byte blocks’, where each byte block can be written to or read from. Each ‘byte block’ contain information of its own size, so one ‘byte block’ may overshadow other blocks. A write happens in the following way, the numbers match the numbers in figure 4.

1. Find the 8 byte chunk that contains the address to be written to.
2. Identify what stored data will be affected by the write and remove that data. Then update the byte block containing metadata about the stored data with the new size. If a byte block is overwritten, but not all stored data, then create a new byte block with the size of the data that was not overwritten. A byte block is illustrated as a filled block in the figure, and the bytes it overshadows are in lighter color compared to the starting block. The new byte block will be placed at the position after where the new data will be written.
3. Write the new block into the cleared out area.

This means we are actually only writing a node into one 1 byte block, and that node contain information about what size it has. This size information is then used to make sure that the other blocks are cleared or updated if needed. If the data that is to be written is too big to fit into 1 chunk, then 2 special nodes are created that represents different parts of the same data. The first node represent as many bytes of the original data that would fit into the first chunk. The second node represents the rest of the data, and a call is then made to write the second node at the next 8 byte aligned address. Reads from memory are handled similarly to writes, shown in figure 5 and mapped to the numbered list below.
5.4 Symbolic Execution of the Trace

Figure 4: An illustration of how data is written to memory.

1. Find the 8 byte chunk that contains the address we will read from and identify which bytes are to be read.

2. For those data items that will not be fully read, create extract nodes extracting the bytes to be read.

3. Concatenate all participating nodes from number 2, and create a result node.

If the read is of the same size as a stored data item, then the handle to that data items will immediately be returned, skipping point 2 and 3. Much like for write, if a read spans several byte chunks, then the read will be split into several read operations. The result from each read operation is then concatenated into the final result. This gives us a good an accurate memory representation as data can not be written per bit in x86 [46], but rather per byte as a minimum.

5.4.2 Approximations

We only consider integer operations during the symbolic execution. In all other cases the values resulting from the operations are concertized. This concretization will mean that we do not have an accurate view of computations involving non-integer arithmetics. We also concertize any addresses...
computed from symbolic data, as this greatly reduces the amount of symbolic variables. Furthermore, concertizing addresses also have the benefit that we do not get symbolic array indexes. Symbolic array indexes may complicate SMT-queries making them more difficult to solve. The downside is that some operations will not be tracked. For instance, if symbolic input is used to calculate the offset in an array or lookup table, then we will not track the data in the array or lookup table.
5.5 Finding a New Path

Once the symbolic execution is completed we need to find a new input to the program so the execution will go down another path during the next execution. At this point in the execution we now have a tree representing all branch points encountered so far. To choose the next path we do a depth first search in the tree looking for a node were we have not explored both the ”branch taken” path and the ”branch not taken” path. Once a node is found that is not fully explored, then the conditions stored in each node from the found node to the root node represents the path constraint. Refer to chapter 2.4.5 about constraint generation for more information about constraints. When we have a path constraint, finding a new path is done by negating the last constraint.

This gives us a representation of a new path. This representation represents almost the same path as the path found in the tree. The exception is the negated node. This means that the new path representation will represent a path were the execution will not go the same way at the very last branch point. In order to get an input that will take that new path, we will transform the path constraint from our internal format into SMT-lib 2.0 format [23]. Doing that enables us to use a solver to generate a solution to the problem. If a solution is found, then it represents what needs to be updated in the inputs to the program under test in order for the next execution to take our new path.

Once the input files have been updated we can trace a new execution and restart the process. If a solution cannot be found, then we can just repeat the process until we find a path that is possible to take. If we run out of potential paths, then we have in theory explored all paths and we are finished.

However, just because a solution is not found does not mean it is impossible to go down that path, just that our system is unable to. For instance, the solver may time out if it takes to long time to solve a query. If that happens, then we view it as if the solver returned that the query was unsatisfiable. Another reason this may happen is due to approximations, see section 5.4.2. When an approximation is used the SMT-queries may not accurately model the path we are looking for. So when solving them, the solver may return the correct result according to the model, but in reality it is the wrong result. In these cases we will detect the error since we never reached the branch point we were looking for. Since we can not reach the branch point with the information at hand, we mark it as unsatisfiable.

5.6 Optimizing the Implementation

5.6.1 VEX Resizing Data

One thing with VEX that is a bit of an annoyance is that VEX resizes data a lot. This happens even if the result is just 1 bit. Valgrind will then not store
the result as 1 bit, but rather a byte as minimum, keeping with alignment rules of the target platform. This gives the result that, for instance, the result of a comparison get the behavior illustrated in listing 11 below.

Listing 11: Code example - VEX resizing

1 /* Store 1 bit in t18 */
2 t18 = CmpEQ64(t7,0xFFFFFFFFFFFFFFFF:I64) [<1:1I1>]
3 /* Expand bit to 64 bits */
4 t17 = 1Uto64(t18) [<0x1:I64>]
5 /* Copy variable */
6 t15 = t17 [<0x1:I64>]
7 /* Shrink the expanded bit down to 1 bit again */
8 t19 = 64to1(t15) [<1:1I1>]
9 /* Copy variable */
10 t10 = t19 [<1:1I1>]
11 /* Check if we should take the branch */
12 if (t10) { PUT(184) = 0x40092FF:I64; exit - Boring }

This created a lot of extra SMT-queries when we translated the VEX-code directly, making the query files unnecessary large. To work around that problem, we designed the symbolic executor to always check if we are converting to the original size before inserting a size converter statement. If we are, then we can just use the original untransformed variable in further computations. This optimization reduced the SMT-query files with well over 50%. This is a Valgrind specific optimization and may not be applicable if a project uses some other way of tracing the code.

5.7 Unrelated Constraint Elimination

Another optimization that is implemented is ”Unrelated Constraint Elimination” [15]. It is an optimization often found in symbolic execution systems [18]. It works by keeping track of which symbolic variables exist in the negated constraint. Once we know what variables exist in the negated constraint, then we do not need to add those constraints that do not share any symbolic variables with the negated constraint to the full path constraint. We can do that since they will have no impact on the new result. Consider the equations below.

\[ y = kx + m \quad (5) \]
\[ 2 \times x = x + 5 \quad (6) \]
\[ 2 \times z = z + 3 \quad (7) \]

The solution to equation 6 influences equation 5. However, the equation 7 would still solve into the same result. For that reason, we can set 'z' to the value 3, rather than having to solve 'z' again when modifying 6. The same goes when modifying a constraint, for unrelated constraints we can just reuse the solutions from the previous execution.
5.8 Controlling the Execution

Symvex supports the following options related to symbolic execution.

**indirect_taint**
Also taint memory accessed with a symbolic address. This is useful for lookup tables, but tend to require a lot more memory.

**taint FILE**
Select which files should be considered tainted. Only data from tainted files will be tracked by the symbolic execution. A minimum of one file is required.

**taint FILE[low:high]**
See 'taint FILE' above with the addition that only the bytes from low to high is considered.

**solver_timeout VALUE**
Set in seconds the maximum time the solver may try to solve a query. The query will be considered as unsatisfied if the solver times out. Default is 300 seconds.

5.9 Limitations

Symvex is limited to the available memory. When reaching an out of memory condition the exploration ends and statistics will be printed. Symvex can also not handle execution in subprocesses to the program under test. This can cause issues if the subprocess does something that the execution in the main process depends on.
6 Evaluation of Our Implementation

In this chapter, our system is compared to the systems tested in the evaluation chapter, chapter 4. Then we will continue with some more in depth tests with or system that we could not perform with the other systems. Our implementation will be referred to as Symvex.

6.1 Test Setup

All tests are run on the same computer as the existing systems were tested on, see table 3. The tests were run with a maximum solver time of 20s for one path constraint. 20s is enough so no timeouts happens in the simpler cases; like the triangle program. We have also limited the available memory for the program to 10GB using ulimit.

6.2 Rerunning Evaluation Tests

6.2.1 Base Cost

When running only an empty program our implementation performs quite as expected as shown in table 8, placing itself among the other implementations using similar approaches.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Time(seconds)</th>
<th>Memory(Kilobytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S²E</td>
<td>20.4</td>
<td>2112044</td>
</tr>
<tr>
<td>FuzzBALL</td>
<td>2.2</td>
<td>62736</td>
</tr>
<tr>
<td>Pathgrind</td>
<td>0.02</td>
<td>77112</td>
</tr>
<tr>
<td>Symvex</td>
<td>0.36</td>
<td>50800</td>
</tr>
</tbody>
</table>

6.2.2 The Triangle Program

Symvex performed well in the triangle program, outperforming all other implementations using similar approaches. For programs of this size it also outperforms S²E, although counting S²E startup cost S²E will probably catch up at some point. The results are illustrated in the table 9.

Printing in the Triangle Program While doing the evaluation of the existing software we noted that a single printing statement could change the execution times a lot. Our implementation fared quite well in this unintended test as illustrated in the table 10.
### 6.3 Real Life Programs

One of the issues with the existing systems was that they had troubles running real world programs. Symvex can handle real world programs, however, verifying that we are finding the expected amount of paths is virtually impossible. This is due to manually reasoning about how many paths we should find is not practically possible, and since the existing systems didn’t work so well, we can not use them as reference either. Also, real world programs tend to have so many possible paths so exploring them all is not possible in reasonable time. In some examples below, we will only consider a part of the input file in order to compare equal input sizes between programs. To only consider a part of the input file is done via our functionality to only

---

**Table 9:** Triangle program results

<table>
<thead>
<tr>
<th>Tool</th>
<th>Time (seconds)</th>
<th>Peak memory usage (Kilobytes)</th>
<th>Paths explored</th>
<th>Average time per path (seconds/path)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S²E</td>
<td>26.5</td>
<td>2188500</td>
<td>26</td>
<td>1.02</td>
</tr>
<tr>
<td>FuzzBALL</td>
<td>71.4</td>
<td>79176</td>
<td>26</td>
<td>2.75</td>
</tr>
<tr>
<td>Pathgrind</td>
<td>14.3</td>
<td>78104</td>
<td>25</td>
<td>0.56</td>
</tr>
<tr>
<td>Symvex</td>
<td>12.2</td>
<td>117404</td>
<td>26</td>
<td>0.47</td>
</tr>
</tbody>
</table>

**Table 10:** Results from printing in the triangle program

<table>
<thead>
<tr>
<th>Tool</th>
<th>Time (seconds)</th>
<th>Paths explored</th>
<th>Average time per path (seconds/path)</th>
<th>Relative slowdown²</th>
</tr>
</thead>
<tbody>
<tr>
<td>S²E</td>
<td>150</td>
<td>150</td>
<td>1.00</td>
<td>0.98</td>
</tr>
<tr>
<td>FuzzBALL</td>
<td>259200</td>
<td>277</td>
<td>935.74</td>
<td>340</td>
</tr>
<tr>
<td>Pathgrind</td>
<td>510</td>
<td>328</td>
<td>1.56</td>
<td>2.79</td>
</tr>
<tr>
<td>Symvex</td>
<td>95</td>
<td>156</td>
<td>0.61</td>
<td>1.30</td>
</tr>
</tbody>
</table>

The results from this test look both promising and worrying. The bad news is that we explored much fewer paths than Pathgrind. Internal statistics collected from Symvex shows that we end up with many diverging paths. As many as 1 out of 3 paths were diverging. For a possible explanation as to why, see section 6.6. On more positive note, we can see that the average execution time per path only went up with about 30% per path explored. Compared to Pathgrinds 179%, a 30% increase can be considered a good result.

²The average time with print divided by the average time without print(from table 7)
³S²E triggered an assert so the result is incomplete.
⁴FuzzBALL never finished so the result is incomplete.
consider some bytes in the input file as symbolic.

Running Symvex on gzip trying to unpack the compressed string "0123456789" yields the result seen in table 11. Symvex finds many paths through the program and eventually we had to stop running since it was taking so long time. However, after manual inspection a big number paths seem to come from enumerating all possible values each character in the start string could be changed to, so the amount of path would have been more than $256^{10}$.

<table>
<thead>
<tr>
<th>Size of symbolic input</th>
<th>Time in seconds</th>
<th>Memory usage</th>
<th>Paths found</th>
<th>Unsatisfied constraints</th>
<th>Paths timed out</th>
</tr>
</thead>
<tbody>
<tr>
<td>43 bytes</td>
<td>3000</td>
<td>211 MB</td>
<td>5170</td>
<td>7539</td>
<td>0</td>
</tr>
</tbody>
</table>

Another real life program that is similar to gzip is tar. In this case, we used a file only containing the text "1", which was compressed to a tar-file. The first item in table 12 represents running tar with the entire file as symbolic input, the second represents running tar with a slice of the file as symbolic input. The slice is as big as the file used in the gzip case.

<table>
<thead>
<tr>
<th>Size of symbolic input</th>
<th>Time in seconds</th>
<th>Memory usage</th>
<th>Paths found</th>
<th>Unsatisfied constraints</th>
<th>Paths timed out</th>
</tr>
</thead>
<tbody>
<tr>
<td>10240 bytes</td>
<td>3600</td>
<td>5 496 MB</td>
<td>122</td>
<td>1826</td>
<td>111</td>
</tr>
<tr>
<td>43 bytes</td>
<td>16</td>
<td>388 MB</td>
<td>3</td>
<td>96</td>
<td>0</td>
</tr>
</tbody>
</table>

We can see that when running Symvex on 'tar' the memory usage is much higher, though comparing equal size of symbolic data we can see that the difference is about 54%. This likely come from that tar executes more time with symbolic data. With more symbolic data the SMT-queries will be bigger and take longer to solve. This can be clearly seen when comparing the number of paths and unsatisfied constraint between gzip and tar. Even though we ran tar for longer, we found fewer paths. In the tar case there are even several times where the constraint solving times out.

Moving on from packaging utilities and taking a look at Symvex performance in a computationally intensive program; converting a png file to
a jpg file. Doing this requires both decoding one format and encoding another. It is done by using the 'Convert' [29] program found on some Linux distributions. In this case we do not perform too well, as illustrated by table 13.

Table 13: Results for Convert

<table>
<thead>
<tr>
<th>Size of symbolic input</th>
<th>Time in seconds</th>
<th>Memory usage</th>
<th>Paths found</th>
<th>Unsatisfied constraints</th>
<th>Paths timed out</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 bytes</td>
<td>53</td>
<td>4 036MB</td>
<td>4</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>9 bytes</td>
<td>53</td>
<td>10 000MB</td>
<td>10</td>
<td>94</td>
<td>0</td>
</tr>
</tbody>
</table>

Already with 3 symbolic bytes we are using over 4GB of memory. Already at 10 symbolic bytes we reach our threshold of 10GB used memory, where roughly 9.2GB is used by the data flow graph. Such high usage in the data flow graph indicates that the symbolic bytes are heavily used in computations.

6.4 Breakdown of Execution Times

For our implementation we have computed a more fine grained breakdown of the execution times. For a small program like the triangle program most time is spent tracing the execution as can be seen in table 14. The times may not add up to the triangle results in table 9 since they do not take startup and tear-down operations into account.

Table 14: Triangle program breakdown

<table>
<thead>
<tr>
<th>Component</th>
<th>Time in seconds</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracer</td>
<td>7.943</td>
<td>Includes forking and tracing, but excludes reading in the trace.</td>
</tr>
<tr>
<td>Symbolic Executor</td>
<td>3.479</td>
<td>Includes reading the trace file and the symbolic execution.</td>
</tr>
<tr>
<td>Path Tree</td>
<td>0.008</td>
<td>Includes finding the new path and generating the constraints for the path.</td>
</tr>
<tr>
<td>Constraint Solver</td>
<td>0.692</td>
<td>Includes writing queries to file, forking, solve time and reading solution from file.</td>
</tr>
<tr>
<td>File updating</td>
<td>0.000</td>
<td>Includes updating the input files.</td>
</tr>
</tbody>
</table>
Different programs can have fundamentally different areas where the execution time is spent. Looking at the breakdown in table 15 for the execution illustrated in table 11, we can see that the tracer still takes the most time. However, compared to the triangle case above, we now spend more time in the solver.

**Table 15:** Gzip unpack breakdown, the first 50 minutes

<table>
<thead>
<tr>
<th>Component</th>
<th>Time in seconds</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracer</td>
<td>1729</td>
<td>Includes forking and tracing, but excludes reading in the trace.</td>
</tr>
<tr>
<td>Symbolic Executor</td>
<td>852</td>
<td>Includes reading the trace file and the symbolic execution.</td>
</tr>
<tr>
<td>Path Tree</td>
<td>30</td>
<td>Includes finding the new path and generating the constraints for the path.</td>
</tr>
<tr>
<td>Constraint Solver</td>
<td>339</td>
<td>Includes writing queries to file, forking, solve time and reading solution from file.</td>
</tr>
<tr>
<td>File updating</td>
<td>0.002</td>
<td>Includes updating the input files.</td>
</tr>
</tbody>
</table>

If we then continue with looking at the tar example illustrated in table 12 we can see that the solver time is now totally dominant. This tar-file was the single character '1' packed. The file is 10KB when packed.

**Table 16:** Tar unpack breakdown, the first 60 minutes

<table>
<thead>
<tr>
<th>Component</th>
<th>Time in seconds</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tracer</td>
<td>54.242</td>
<td>Includes forking and tracing, but excludes reading in the trace.</td>
</tr>
<tr>
<td>Symbolic Executor</td>
<td>66.325</td>
<td>Includes reading the trace file and the symbolic execution.</td>
</tr>
<tr>
<td>Path Tree</td>
<td>38.292</td>
<td>Includes finding the new path and generating the constraints for the path.</td>
</tr>
<tr>
<td>Constraint Solver</td>
<td>3338.5</td>
<td>Includes writing queries to file, forking, solve time and reading solution from file.</td>
</tr>
<tr>
<td>File updating</td>
<td>0.071</td>
<td>Includes updating the input files.</td>
</tr>
</tbody>
</table>
In table 16 we can see that the constraint solving time corresponds to close to 93% of the execution time. In this example we left Symvex running for approximately 60 minutes. In this time we only found 122 satisfiable paths and 1826 unsatisfiable paths. This distribution is very different compared to the triangle program which had 26 satisfiable paths and 18 unsatisfiable paths. The reason for that is due to `tar` performing most of its operations on the symbolic data in a loop. Due to the looping and that Symvex finds new paths in a DFS manner, execution tends to get stuck in unwinding the loop backwards, resulting in very many unsatisfiable constraints. Implementing loop detection, see section 6.8.1, could be very useful in cases like this. Combining loop detecting with iterative solving of constraints, see section 6.8.2, could potentially improve performance a lot. This is because if we know that a constraint in a loop is not solvable, then all following iterations of the loop will not be solvable either. This is because the constraint that is not solvable will be a part of the path constraint for all the later constraints in the loop. With that information we can skip trying to explore every path in the loop and up to the unsolvable part. The unrelated constraint elimination optimization that we implemented performs very poorly in loops as the loop will consider the same data over and over again.

6.5 Memory Breakdown

The memory usage in Symvex consists of mainly 2 parts, the graph and the path tree. For small programs, the graph is totally dominating because it is using preallocating memory. The main usage of the graph is as discussed in the design part to keep track of the current execution. The main usage of the path tree is to keep track of branch points between executions, where most memory is allocated to keeping track of the constraints we have already found. The memory usage is a combination between two main factors, execution time and the amount of symbolic input. The longer the execution time is, the more branch points we will pass, which will increase the amount and size of the constraints we keep track of. The graph will also grow in size. However, not every branch point is interesting as that depends on if symbolic input were in the condition for the branch point. The more bytes that were initially tainted the bigger is the chance, that a byte will be in a condition making it symbolic.

6.6 Problem with Diverging Paths

A diverging path comes from when a path condition have been solved and the solution does not correctly model the path we expected to take. This will make the next execution go down a unexpected path. The reason for this, could either due to approximations or a bug in our implementation. It is not possible without a great effort, to verify that when we encounter a diverging path, it is not due to a bug in the implementation.
6.7 Conclusion

We managed to create a system that works fairly well in comparison to the systems that did exist. However, we share some of the memory issues with those systems, although those memory issues stem from the inherent memory usage that comes with symbolic execution. They are most noticeable in computationally heavy programs. One way to mitigate the memory usage problem is to store the data flow graph on disk instead of in the RAM. [47] Also we are possibly experiencing some issues with diverging paths. On the upside, with our system is it possible to run real programs, as long as there is memory available. Our system is missing most possible optimizations. Implementing those would most likely make our system compare even better to the other systems we looked at. Two useful optimizations is presented in section 6.8

6.8 Further Work

As we saw in the results for our system there is still work to do. We need to implement more optimizations, such as iterative constraint solving. We also were able to reason that being able to spot loops could potentially be useful too, especially in combination with iterative constraint solving.

6.8.1 Loop Detection

Some programs perform much of their execution in a loop. When executing in a loop, the same branch points are visited many times. Being able to detect that could help us avoid unnecessary work. One way to detect a loop could be to look at the path constraints and look at what branch points we have in the path constraint. This could be done with the help of the Floyd’s cycle-finding algorithm [48] or Brent’s algorithm[49].

6.8.2 Iterative Constraint Solving

The iterative constraint solving optimization can optimize away several unneeded calls to the solver. If a constraint is unsatisfiable, then any path containing that constraint will have an unsatisfiable path constraint. This means we can skip trying to explore said paths. Especially loops were an issue in some of our tests as we were unwinding them backwards. With this optimization we could discover early in the loop that we cannot find any new paths in the loop. This could help us avoid unwinding loops unnecessarily. The issue with loops was very obvious in the example illustrated in table 16.
7 Conclusion and Discussion

At the start of the project we defined the following questions.

1. How well do existing symbolic execution systems that runs on binary programs scale with larger software?

2. Which approach is preferable when you want to implement a symbolic execution system that scales with larger software?

Together with two goals.

Therefore as this thesis work’s main goal will be to create a prototype for a symbolic execution system which can be extended later on. A secondary goal for this thesis work will be to optimize the implementation with the help of strategies from existing systems.

7.1 Symbolic Execution and Larger Software

7.1.1 Existing Systems

We found 3 systems that are capable of doing symbolic execution of compiled programs. These systems are FuzzBALL, S²E, and Pathgrind. We found that neither system worked well with larger software. S²E was the one that performed best, and was able to execute real world programs. The downside with S²E was that the memory usage was just too great to do anything meaningful.

7.1.2 The Preferable Approach

We have seen, if a symbolic execution system choses the approach to fork at every branch point, then it will suffer from a very high memory usage. This was the case for S²E. The reason for the high memory usage is that forking saves too much state at every branch point. The approach with running the programs from start to the end and only saving a minimal state consumed much less memory, at the cost of execution speed. That approach was chosen by FuzzBALL and Pathgrind. However, to be able to run long programs, low memory usage is more useful than extra speed. For that reason, we argue that the approach chosen by FuzzBALL and Pathgrind is the better approach in this use case. Therefore will we base our implementation on that approach.

7.1.3 Status of Our Implementation

Our implementation is based mainly around ideas from FuzzBALL and Pathgrind. This has enabled us to keep the memory usage fairly low, while still having a decent execution speed. The implementation reached quite a bit on the way of becoming a working system, however not the entire
way. We still have issues with high memory usage when dealing with too much symbolic input or running too long programs. We also run into divergent paths quite often which we do not know if they are just result of the approximations we make or due to some bug in the implementation.

Even though we have some problems, the performance of our system compares favorably in comparison to the systems that use a similar approach to our system, such as Pathgrind and FuzzBALL. In terms of memory usage we also compare well with S²E, however, looking at the speed of execution, S²E is still faster. We have made some suggestions on what can be done to increase the performance of our implementation. These suggestions can be found in section 6.8.
References

[1] Eushiuan Tran, Carnegie Mellon University, Verification/Validation/Certification, Spring 1999,
Available at: http://users.ece.cmu.edu/koopman/des_s99/verification/index.html
Last accessed: 2015-05-12

[2] Yarom Yuval, Method of relocating the stack in a computer system for preventing overrate by an exploit program
United States Patent No. 5,949,973

[3] HP, Data Execution Prevention
Available at: http://h10032.www1.hp.com/ctg/Manual/c00387685.pdf
Last accessed: 2015-05-07

DOI=http://dx.doi.org/10.1145/1499799.1499863

[5] Arno Wagner, ETH, (S)LOC Count Evolution for Selected OSS Projects,
Last accessed: 2015-05-12

DOI=http://dx.doi.org/10.1109/TDSC.2004.2

[7] Sans Institute, Data Leakage - Threats and Mitigation
Available at: http://www.sans.org/reading-room/whitepapers/awareness/data-leakage-threats-mitigation-1931
Last accessed: 2015-06-08

DOI=http://dx.doi.org/10.1109/SP.2013.45

REFERENCES

Proceedings DARPA Information Survivability Conference and Exposition, Hilton Head, SC, January 2000; 119-129. DOI:http://dx.doi.org/10.1109/DISCEX.2000.821514


[17] Mozilla, Download Firefox,
Available at: https://www.mozilla.org/en-US/firefox/new/
Last accessed: 2015-05-12

Queue 10, 1, Pages 20 (January 2012), 8 pages. DOI=http://dx.doi.org/10.1145/2090147.2094081


Commun. ACM 54, 9 (September 2011), 69-77. DOI=http://dx.doi.org/10.1145/1995376.1995394

Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.453.4433&rep=rep1&type=pdf
Last accessed: 2015-11-02


[23] SMT-Lib, SMT-LIB The Satisfiability Modulo Theories Library
Available at: http://smtlib.cs.uiowa.edu/standard.shtml
Last accessed: 2015-05-11

Available at: http://stp.github.io/cvc-input-language/
Last accessed: 2015-05-11

[25] DIMACS, Center for Discrete Mathematics and Theoretical Computer Science,
Available at: http://www.cs.ubc.ca/~hoos/SATLIB/Benchmark-s/SAT/satformat.ps
Last accessed: 2015-05-11
[26] Jaeho Shin, Memusg
Available at: https://gist.github.com/maroneze/24cee559d5ce5b812ea
Last accessed: 2015-11-01

[27] Jean-loup Gailly, Mark Adler, Gzip - GNU Project - Free Software
Foundation
Available at: http://www.gnu.org/software/gzip/gzip.html
Last accessed: 2015-11-01

[28] The GNU project, GNU tar - Summary
Available at: http://savannah.gnu.org/projects/tar
Last accessed: 2015-11-01

[29] ImageMagick Studio LLC, Command line tools: Convert
Available at: http://www.imagemagick.org/script/convert.php
Last accessed: 2015-09-28


[31] QEMU, open source process emulator,
Available at: http://wiki.qemu.org/Main_Page
Last accessed: 2015-06-08

[32] The LLVM Compiler Infrastructure Project,
Available at: http://llvm.org/
Last accessed: 2015-06-08

[33] Vijay Ganesh. 2007. Decision Procedures for Bit-Vectors, Arrays and
Integers.
Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) David L. Dill. AAI3281841.

[34] Vitaly Chipounov, Analyzing Large Programs Using Concolic Execution
Available at: https://github.com/dslab-epfl/s2e/blob/master/docs/Howtos/Concolic.rst
Last accessed: 2015-06-21

[35] David Brumley, Berkeley, VinE Project Documentation,

[36] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min
DOI=http://dx.doi.org/10.1007/9783540898627_1


[45] Intel, *Intel®64 and IA-32 Architectures Software Developer Manuals* Available at:
REFERENCES

Last accessed: 2015-05-12

[46] Intel, *Instruction Set Reference, AM*
Volume 2a, page 95
Available at: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64ia32architecturessoftwaredevelopervol2amanual.pdf#page=95
Last accessed: 2015-09-02


[48] Floyd’s Cycle Detection Algorithm (The Tortoise and the Hare) - Siafoo
Available at: http://www.siafoo.net/algorithm/10
Last accessed: 2015-10-07

In BIT Numerical Mathematics, June 1980, Volume 20, Issue 2, pp 176-184, DOI=http://dx.doi.org/10.1007/BF01933190
8 Appendix

8.1 The Triangle Program

Listing 12: Code example - The triangle program

```c
#ifndef TRI_H
#define TRI_H
typedef enum {INVALID, EQUILATERAL, ISOSCELES, SCALENE} TriangleType;

void swap(int s[], int i, int j);
TriangleType getType(int sides[], int sides_length);
const char* TypeToString(TriangleType type);
#endif

#include "triangle.h"
#include <stdlib.h>

const char* TypeToString(TriangleType type) {
    switch(type) {
    case EQUILATERAL:
        return "Equilateral";
        break;
    case ISOSCELES:
        return "Isosceles";
        break;
    case SCALENE:
        return "Scalene";
        break;
    }
    return "unknown triangle type";
}

TriangleType getType(int sides[], int sides_length) {
    if(sides_length != 3) {
        return INVALID;
    }

    int s[3];
    TriangleType result = INVALID;
    for(int i = 0; i < sides_length; i++)
```
\[ s[i] = \text{sides}[i]; \]

\[
\text{if } (s[0] > s[1]) \\
\quad \text{swap}(s, 0, 1);
\]

\[
\text{if } (s[0] > s[2]) \\
\quad \text{swap}(s, 0, 2);
\]

\[
\text{if } (s[1] > s[2]) \\
\quad \text{swap}(s, 1, 2);
\]

\[
\text{if } (s[0] \leq 0 \text{ || } s[2] - s[0] \geq s[1]) \\
\quad \{ \\
\qquad \text{return } \text{INVALID}; \\
\quad \}
\]

\[
\text{if } (s[0] == s[2]) \\
\quad \{ \\
\qquad \text{result} = \text{EQUILATERAL}; \\
\quad \}
\]

\[
\text{else if } (s[0] == s[1] \text{ || } s[1] == s[2]) \\
\quad \{ \\
\qquad \text{result} = \text{ISOSCELES}; \\
\quad \}
\]

\[
\text{else} \\
\quad \{ \\
\qquad \text{result} = \text{SCALENE}; \\
\quad \}
\]

\[
\text{return } \text{result};
\]

\[
\text{void swap} (\text{int } s[], \text{int } i, \text{int } j) \\
\quad \{ \\
\quad \text{int } \text{tmp} = s[i]; \\
\quad \text{s}[i] = s[j]; \\
\quad \text{s}[j] = \text{tmp}; \\
\quad \}
\]

// main.cpp : Defines the entry point for the console application.

#include "triangle.h"
#include <stdio.h>

\text{int } \text{main} (\text{int } \text{argc}, \text{char}* \text{argv}[])
\begin{verbatim}
86  
87      int exit_code = 0;
88      int sides[3];
89      FILE* fd;
90      char data[5];
91      fd = fopen("file", "r");
92      fread(data, sizeof(char), 5, fd);
93      sides[0]=data[0] - 48;
96
97      TriangleType tt = getType(sides, 3);
98
99      fclose(fd);
100     return exit_code;
101  
102  }
\end{verbatim}
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsmann i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet – or its possible replacement – for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

©Mattias Rönn