Fakultät
Elektronik und Informatik

Master Thesis

über

Automatic Detection of Vulnerabilities in Black Box Applications Independent of Processor Architecture

Autor: Birk Kauer
Birk.Kauer@Kauer-Consulting.de

Erstgutachter: Prof. R. Hellmann
Zweitgutachter: Prof. Dr. M. Gelderie
Betreuer: Dr. O. Matula

Bearbeitungszeit: 27.07.2018-27.01.2019

I Abstract

Today, identifying and pinning down bugs (and security issues in particular) in closed source applications is mostly based on very time-consuming manual reverse engineering. This process is often supported by IDA-Scripts[31], or some rather special enterprise solutions like MAYHEM[29] from AllSecure, which can rarely be affordable by private persons, researcher institutions or small companies.

In this master thesis, we present a methodology to hunt for bugs independent of the CPU Architecture (e.g. ARM/X86/PPC/MIPS) using an intermediate language. The thesis covers different approaches for working with intermediate languages and handling various specific problems, such as compiler optimizations, for example. Furthermore, it will include case studies of already discovered vulnerabilities using the proposed methodology to underline its practicability.

Alongside the master thesis, we release a Framework with several plugins to detect security issues in compiled programs using the Binary Ninja[59] API. The Framework is built to be easily extendable by the community. The released plugins aim to find some particular vulnerabilities, such as Buffer Overflows, Integer Overflows, uninitialized variables, and some more even in the case of heavy compiler optimization. Further, the Framework can prioritize vulnerabilities by analyzing code coverage files and determine for which vulnerability already valid input exists.

II Content

1 Introduction

Searching automatically/programmatically for vulnerabilities or bugs in programs is already a common theme in the software industry with source code scanners such as HP Fortify [30] or IBM AppScan [15]. These scanners can analyze provided source code to discover potential problems.

However, often source code is not available for people outside of the developing corporation. Hence, the previously mentioned source code scanner can not aid in analyzing software for a security researcher. The following chapters describe several concepts and proof of concepts in a way to examine already compiled programs. It is worth mentioning that any automated approach can never be a complete solution to solve any known problems and it is not possible to tell if the software found every existing bug due to the theoretical proof of the halting problem[58]. Hence, the later-described techniques can always generate false positives.

To secure their IT infrastructure, companies rely on services that aid them in identifiy vulnerabilities. Howoever, often the process of identifiy vulnerabilties is limited by time, i.e. the industry pays the contractor for a pre-defined timeframe. This often includes a full analysis of multiple components where often some parts are overlooked when planning the timeframe. It is quite often the problem that some core parts of the software or system are more complex than expected or were not even considered in the planning phase. Hence, the fulfilling team or person tackle time constraints and often miss core implementations due to lacking time. The presented framework in this thesis is designed to aid massively in such time constrained scenarios. While it is not designed to solve every problem introduced by bugs, it is instead designed to help the auditor to present results in an easy to understand and prioritized fashion. This approach should enable the auditor to focus on particular parts of the software to either rule out false positives or find a relevant security issue in this system.

The automated vulnerability analysis of already compiled programs is and was always a rewarding challenge. Due to the currently increasing use of different architectures like ARM/MIPS/X86 and X86_64, such analysis is getting more and more complex to yield identical results for the same code base compiled to different architectures. However, it is needed to analyze such devices in a semi-automated manner due to the massive increase of IoT products on the market. These devices are often prone to very basic vulnerabilities such as stack buffer overflows. Such basic vulnerabilities often relate to the required low production costs and the lack of understanding of such existing vulnerabilities. Further, since the already mentioned source code scanners like Fortify and AppScan are costly,

they are often skipped for development of low budget IoT devices. However, since IoT-devices are often connected to the Internet as their name sugests, they can be a rewarding and easy target for attackers. Some of these devices are required to be exposed directly to the Internet which exposes them to a large user base of potentially malicious actors.

Due to the halting problem, it cannot be decided if such analyses identified all vulnerabilities and it is always required to distinguish between positive findings and false positives manually. Nevertheless, the presented framework in this thesis can achieve a speed advantage for a security researcher to idenitfy security vulnerabilities without manual reverse engineering. However, since IoT devices are often based on other architectures such as different variants of ARM or MIPS, it is preferable to analyze the software on such devices independently of their underlying architecture.

The remainder of this work is structured as follows: Chapter 2 discusses several core concepts required for this thesis. Besides introducing concepts for occurring vulnerability patterns in general, the different approaches for analyzing such patterns are introduced. This is followed by an introduction of the overall concept of intermediate languages (abbreviated IL in the following) and some of the current successful ILs such as LLVM[40]/BAP[16], especially in more detail the Binary Ninja IL. Chapter 3 highlights several approaches analyzing machine code to search for patterns with our proposed framework for vulnerability analysis which is described in Chapter 4. Essential aspects of the implementation are reviewed in Chapter 5 before the results of the performed analysis are presented in Chapter 6. The thesis finishes with a conclusion and discussion of further research topics in Chapter 7.

2 Background

This chapter introduces the technical concepts and terminology required for the rest of this thesis. Section 2.1 introduces different processor architectures and their characteristics. The chapter continues with chapter 2.2, which introduces different intermediate languages and a conclusion of these languages. After the introduction of Single Static Assignments in chapter 2.3 there is a brief chapter 2.4 on how disassembler detect functions with two selected algorithms for function detection. Finally in chapter 2.5 the usage of Abstract Syntax Trees is explained followed by the big chapter 2.7 which introduces the use of Concolic Execution.

2.1 Architectures

A core target of this thesis is to search for vulnerabilities on different platforms without much modification of the analysing code for each archicteture. Thus, it is required to introduce the most common used architectures by the time of writing this thesis. Section 2.1.1 describes some internals and key parts of the X86 and X86_64 architecture which are the most widely used architectures afterall, while ARM is described in Section 2.1.2 and the MIPS architecture is described in section 2.1.4. Finally, the PowerPC architecture is described in section 2.1.3.

2.1.1 X86/X86_64

X86 and X86_64 are Complex Instruction Set Computer (abbreviated CISC in the following) architectures where X86 refers to the 32-bit solution and X86_64 represents the 64-bit solution. Due to the CISC architecture, the instruction set is the most complex and can take multiple CPU cycles per executed instruction. It is by far the most common processor architecture due to AMD and Intel pushing their products into the personal computer market. However, the latest generations of Intel suffered from speculative execution attacks such as spectre and meltdown[11].

Registers. While the original X86 architecture had 16-bit registers, currently on X86 32-bit registers are standard, and on X86_64 these registers are 64-bit wide. The following table illustrates the original purpose for the registers. However, they can be used by arbitrarily by the compiler.

Register	Purpose
AL/AH/AX/EAX/RAX	Accumulator
BL/BH/BX/EBX/RBX	Base index
CL/CH/CX/ECX/RCX	Counter
DL/DH/DX/EDX/RDX	Extended precision of Accumulator
SI/ESI/RSI	Source index
DI/EDI/RDI	Destination index
SP/ESP/RSP	Stack Pointer
BP/EBP/RBP	Frame Base Pointer
IP/EIP/RIP	Instruction Pointer

Since RAX/RBX/RCX... are the representations for 64-bit registers and therefore only accessible in X86_64 the rest is addressable in X86 and X86_64. For example, RAX represents the full 64-bit register while EAX represents 32-bits and AX 16-bits. The only change is AL and AH where AH represents the higher 8-bits from AX and AL the lower 8-bits. The following figure displays the addressing of the register (Note: the "?" is used as a placeholder for the different register names):

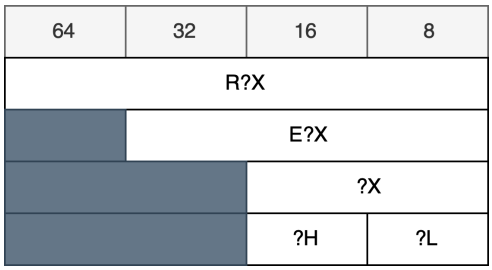

Figure 1: Addressing registers

Operating Modes. X86 supports the use of real and protected mode. The processor uses the real mode while booting and running instructions from the basic input/output system (BIOS) while the protected mode is used by the operating system later to protect application address spaces from corrupting each other.

Due to the 32-bit constraint in protected mode, the long mode was introduced to address 64-bit architectures in X86_64.

Alignment. X86 and X86_64 do not require any address aligment when executing instructions due to the CISC architecture.

2.1.2 ARM

Advanced RISC Machine in short ARM is based on the reduced instruction set computing architecture and is used for the maturity of IoT devices. This architecture slowly finds its ways to the consumer laptop market (e.g., Pinebook [1]) due to the small usage of power.

Registers. ARM with its 32-bit or respectively 64-bit wide registers contains 13 registers for general purpose and not allocated for a specific task as defined in X86 and X86_64. The following table displays a brief overview of the register in ARM and their purpose:

Register	Purpose
R0-R12	General-purpose registers
R13	Stack Pointer (SP)
R14	Link Register (LR)
R15	Program Counter (PC)

The special purpose registers which range from 13 to 15 are somewhat similar to the X86 Structure besides the Link Register. However, R13 is the stack pointer which is similar to ESP/RSP in X86 and X86_64 while R15, the Program Counter (PC) is similar to EIP or RIP. The only deviation from the X86 registers is the Link Register as noted earlier. This register is used for faster function calls. This register is used when a subroutine completes and returns to the caller. The method of using a register is faster than storing the return address on the stack.

Operating Modes. ARM, in general, is based on 32-bit wide instructions. To compress code density the Thumb mode was introduced in 1994 with ARM7TDMI[2] where the processor can operate in two different modes — the default ARM mode and Thumb mode where the instructions are 16-bit wide. The later Thumb-2 mode[3] was introduced to support both 32-bit and 16-bit wide instructions which made ARM a dynamic instruction length architecture.

Alignment. ARM requires the instructions to be aligned by 32-bits. While in thumb mode there is only a 16-bit alignment required as described above.

2.1.3 PowerPC

Performance Optimization With Enhanced (abbreviated RISC in the following) – Performance Computing in short PowerPC or often referred to as PPC and was mainly used by Apple and IBM in the past. Since Apple migrated to the Intel X86_64 architecture, it became a niche architecture in the server and personal computer market. However, it is still heavily used by embedded devices.

The most notable difference is the ability to switch endianness during execution.

Registers. The PowerPC architecture contains 32 general-purpose registers ranging from GPR0-GPR31 and 32 floating-point registers as displayed in the table below:

Register	Purpose
GPR0-GPR31	General-purpose registers
FPR0-FPR31	Floating-point registers
CR	Condition register
FPSCR	Floating-point status and control register
XER	XER register
LR	Link register
CTR	Count register

The condition register (CR) reflects results from arithmetic operations while the Floating-point status and control register contains all floating point exceptions in a bitmask. The XER register is used as a bitmask to indicate overflows or the number of bytes transferred by specific load or store instructions. The link register (LR) and the count register (CTR) are similar to the link and program counter (PC) register in the ARM (Section 2.1.2) architecture.

The notable difference is across different PowerPC generations the general-purpose registers are sometimes referred similarly to ARM with r0-r32. Further, some registers are flagged as volatile which results in a value loose during a function call.

Operating Modes. As briefly described earlier it is possible to switch endianness while executing. This is done by setting or unsetting a bit in the machine state register (MSR). By default, the processor starts in Big-endian mode and can be switched dynamically. The possible change of endianness needs to be accounted for while performing the different analyses.

Alignment. Since PPC is a RISC architecture, the instructions are 32-bit wide aligned.

2.1.4 MIPS

Microprocessor without Interlocked Pipelined Stages (abbreviated MIPS in the following) is also a RISC architecture. This processor architecture exists for 32-bit and 64-bit and is not as widely used as ARM or PowerPC, but it does occur on some embedded systems.

Registers. The MIPS architecture contains 32 general-purpose registers ranging from r0 to r31 where from register r28 they have special purposes:

Register	Purpose
r0	zero register (always zero)
r1-r25	general-purpose registers
r26-r27	$k0 and $k1 reserved by the operating system
r28	Global pointer ($gp)
r29	Stack pointer ($sp)
r30	Frame pointer ($fp)
r31	Return address ($ra)

It should be noted that the instruction pointer or instruction counter is not part of the default 32 direct accessible registers and is placed in the special-purpose registers as PC register where it historically resides with the HI and LO register which were used for multiplication and division. However, those two registers HI and LO were removed in the latest MIPS generations.

Alignment. Since MIPS is a RISC architecture, the instructions are 32-bit wide aligned.

2.2 Intermediate Languages

This chapter introduces the technical concepts and terminology for intermediate languages/representations.

The following definition of intermediate languages will be used within this thesis:

> An intermediate representation is a representation of a program "between" the source and target languages. A good IR is one that is fairly independent of the source and target languages, so that it maximizes its ability to be used in a retargetable compiler.[57]

During the compilation process, some compiles make heavy use of an IL such as LLVM, for example, as used by the clang compiler [46]. Converting the given source code to an IL is only one of the multiple steps in the compilation process. Further, compilers can use this intermediate language to emit assembly code for multiple architectures such as X86/ARM/PPC or MIPS or perform optimizations.

The following sections describe several intermediate languages/representations and a conclusion on the selected intermediate language to perform analyses with the introduced framework.

2.2.1 LLVM

The Low-Level Virtual Machine (abbreviated LLVM in the following) intermediate representation was initially developed and started by Vikram Adve and Christ Lattner. Later Apple hired Lattner to combine LLVM into the Apple developer system. It was initially designed to create dynamic compilation techniques for static and dynamic programming languages and consists of multiple front-ends (source code input) and several backends to emit assembly for different architectures.

The following high level architecture clang overview displays this process:

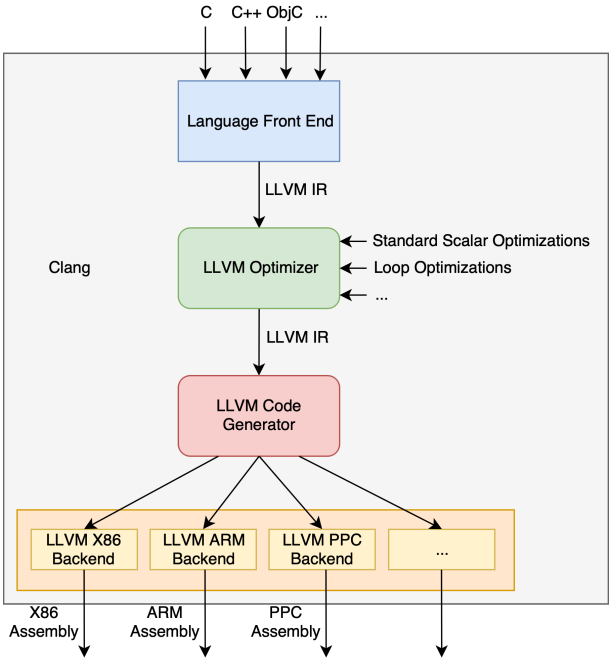

Figure 2: Clang Architecture

The previous picture displays the high-level architecture of the clang compiler. It is possible to feed clang with different sources (e.g., C/C++/ObjC and more). This initial source code is emitted to their LLVM intermediate language where the LLVM Optimizer can perform the code analysis and improve the performance. The optimized LLVM intermediate language is then passed to the LLVM Code Generator which emits based on the intermediate language input the target assembly which can be multiple architectures.

When looking at the architecture, it is interesting how the process of emitting the LLVM intermediate language works. The following will demonstrate the conversion from the C code given below to the LLVM language by clang:

```
1 #include<stdio.h>
2
3 int foo(int a, int b){
4   return a + b;
5 }
6
7 int main(){
8   foo(3,4);
9 }
```

Listing 1: Clang Compilation

The above given source code can be compiled with the following command. The command-line parameters force clang to emit the created LLVM assembly to a file with a ".ll" ending.

```
# clang main.cpp -emit-llvm -S
```

The following excerpt displays the generated LLVM assembly generated by clang:

```
[...]
; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @_Z3fooii(i32, i32) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = add nsw i32 %5, %6
  ret i32 %7
}

; Function Attrs: noinline norecurse nounwind optnone ssp uwtable
define i32 @main() #1 {
  %1 = call i32 @_Z3fooii(i32 3, i32 4)
  ret i32 0
}
[...]
```

Listing 2: LLVM IL

When comparing the original C code and the emited LLVM assembly it is notable that besides the several load and stores inside the "foo" function and function attributes (Attrs) most of the LLVM assembly is still very similar to the original C source code. However, compilers use ILs to perform heavy code optimizations. This code optimization is displayed in the following code example. The following C code is given to be optimized by clang:

```
#include <stdio.h>

bool is_sorted(int *a, int n) {
  for (int i = 0; i < n - 1; i++)
    if (a[i] > a[i + 1])
      return false;
```

```
7     return true;
8   }
9
10  int main(){
11    int list[5] = { 1, 3, 2, 7, 9 };
12    printf("Result = %d\n", is_sorted(list, 5));
13    return 0;
14  }
```

Listing 3: LLVM optimization

Since there will be not much to optimize in the main function the optimization from LLVM will focus on the is_sorted function which takes as a first parameter a pointer to a list/array of integers and the second parameter is the overall size of the integer array. The function is designed to determine whether the passed array is already sorted or not.

The following control flow graph displays the is_sorted function in the LLVM intermediate language without performing any code optimization (-O0) (Note: it is not required to understand the emitted LLVM assembly fully, the example solely displays the optimizations the compiler will perform on the intermediate language):

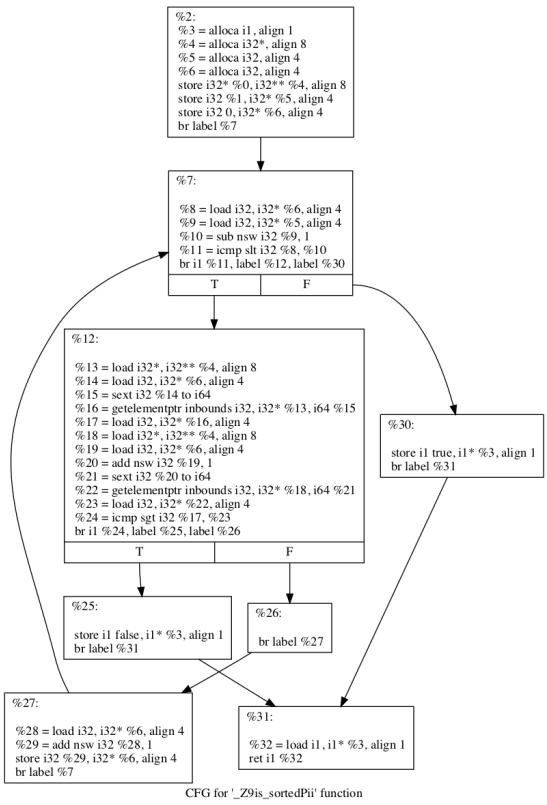

Figure 3: LLVM CFG no optimization

While the following control flow graph displays the best optimizations (-Ofast) performed on the LLVM intermediate language:

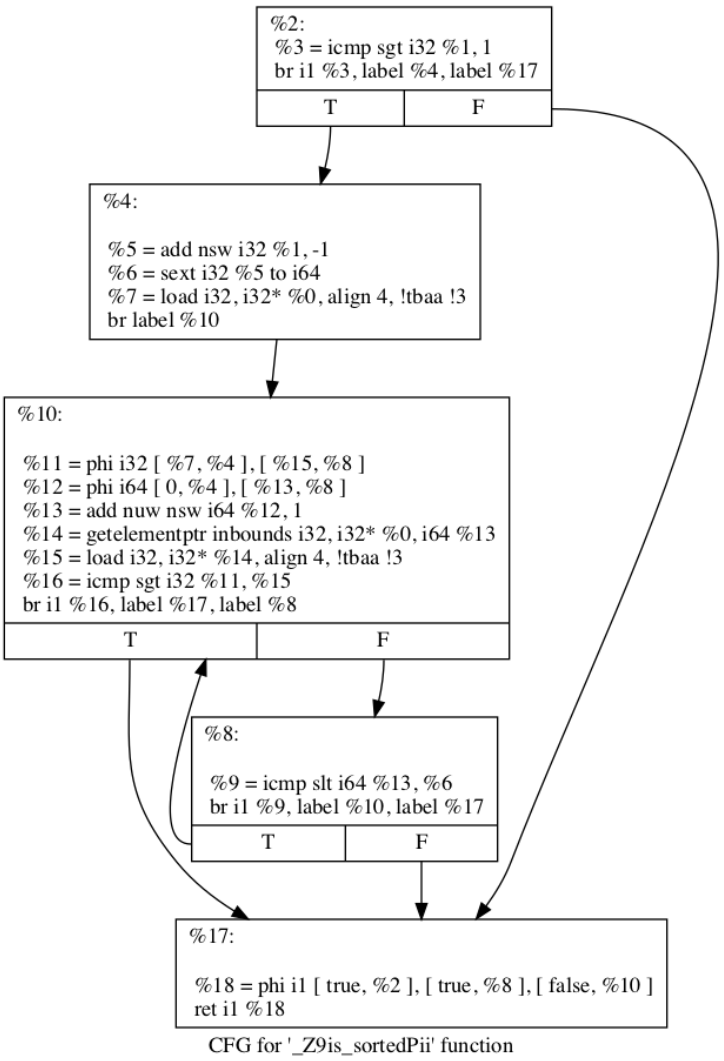

Figure 4: LLVM CFG full optimization

Comparing these two graphs results in a restructured code flow and a significant cut on instructions in the basic blocks inside the graph. The code will still perform the very same operations needed to return the correct result but was stripped down to the fastest way of running. One of the best hitting optimization which resulted in a significant amount of cut down of the LLVM code was performed by the Scalar Replacement of Aggregates (sroa)[47] transformation algorithm from LLVM. This optimization attempts to promote every alloca instruction (LLVM assembly) to a Single Static Assignment register (SSA is described in depth in section 2.3). Since a single alloca can now be promoted into multiple registers, it is possible for example to give up unnecessary stack variables.

Heavy compiler optimizations are a problem which needs to be dealt with in the automatic

analysis phase since some optimizations will change the code drastically.

For further understanding of the LLVM intermediate language, the official documentation and language reference guide is recommended [45].

2.2.2 BIL

The Binary Analysis Platform [16] (abbreviated BAP in the following) also contains an intermediate representation: The BAP intermediate Language (abbreviated BIL in the following). BAP is divided into front and backend components which are connected by the BAP intermediate Language. While the frontend is responsible for lifting the binary code from the supported architecture to the intermediate language, the backend implements different analyses. The following diagram describes the high-level BAP architecture:

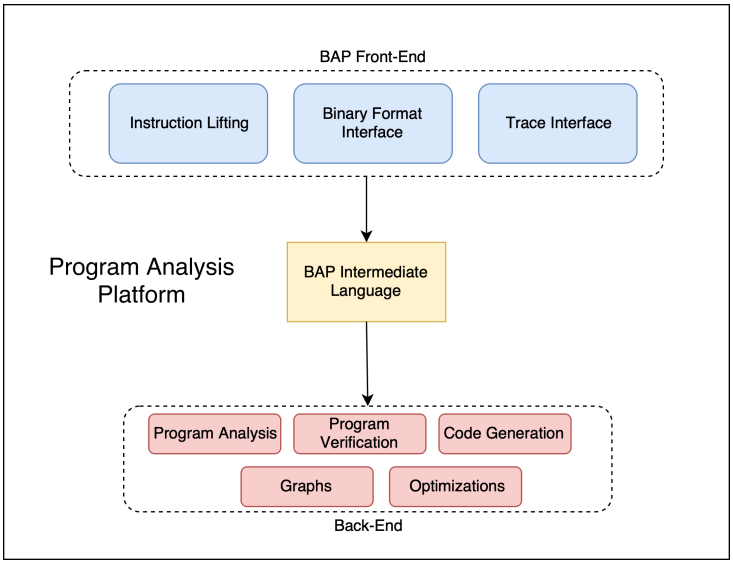

Figure 5: BAP Architecture

Lifting the instructions is implemented by using the linear sweep algorithm[53]. The linear sweep algorithm is described in detail in section 2.4.2.

Lifted BIL code is represented as an Abstract Syntax Tree (abbreviated AST in the following). Performing the analysis for vulnerabilities would be performed by walking the abstract syntax tree. BAP itself is written in OCaml[43] which is designed for functional programming. Hence, due to the design BAP exposes its application programming interface (abbreviated API in the following) mostly through OCaml and serves a limited Python/C and rust interface. Since OCaml is not as widely used as Python or C and the BAP API in other programming languages is limited, usage of the BAP API can be difficult. Hence, when evaluating which intermediate language/representation is the best for fast development and providing an accessible framework, BAP was not used within

this thesis.

However, to introduce different intermediate representations the following displays the BAP intermediate Language with the same example which was used in the LLVM example code (Listing 3) for optimizing loops.

The following graph displays the previous mentioned is_sorted function in the BAP intermediate Language:

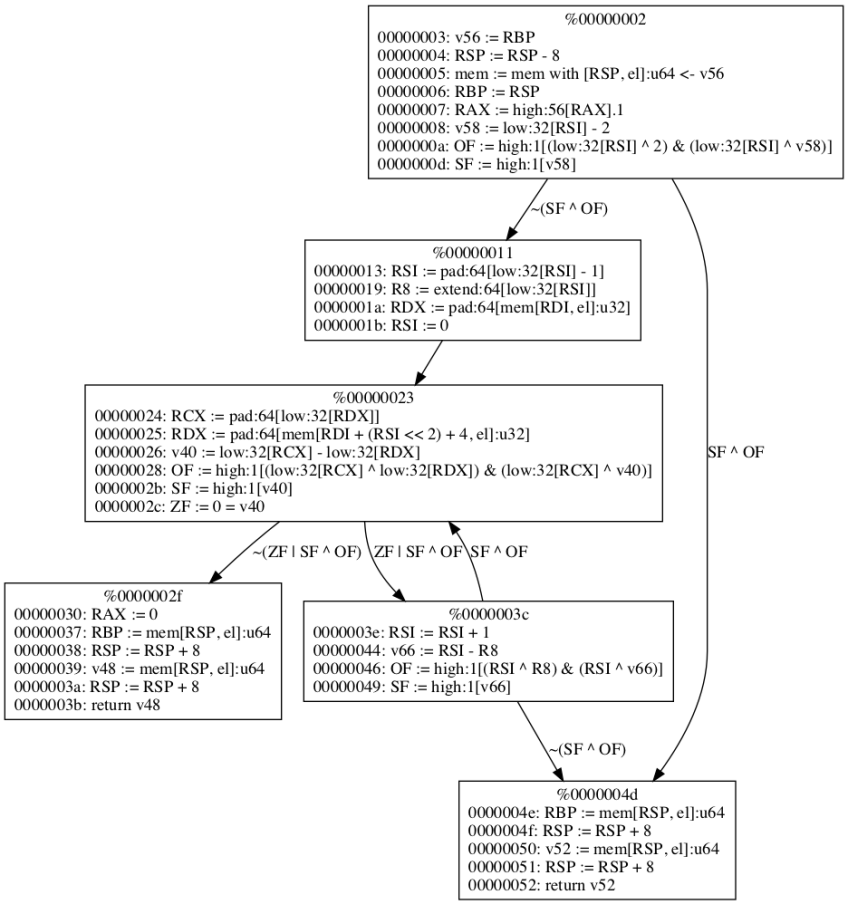

Figure 6: BIL CFG

Examining at the graph, it is a rather complex to understand intermediate language due to the very long number converting instructions.

2.2.3 VEX

VEX is a part of the dynamic instrumentation framework called Valgrind. Valgrind in itself is a virtual machine that provides a framework for building dynamic instrumentation software and is widely known for its memcheck tool. At the core, Valgrind uses its self constructed VEX intermediate language. VEX is used in a variety of other tools to perform the lifting to an intermediate language. The most notable tool which uses VEX is angr[54]. The angr framework will be a future milestone to be implemented in the proposed framework. Thus, it is worthy to introduce VEX as intermediate language as well. Again the very same function is_sorted is used to be represented in VEX as the following graph displays (Due to the extensive language the function prologue is skipped in the graph for better display):

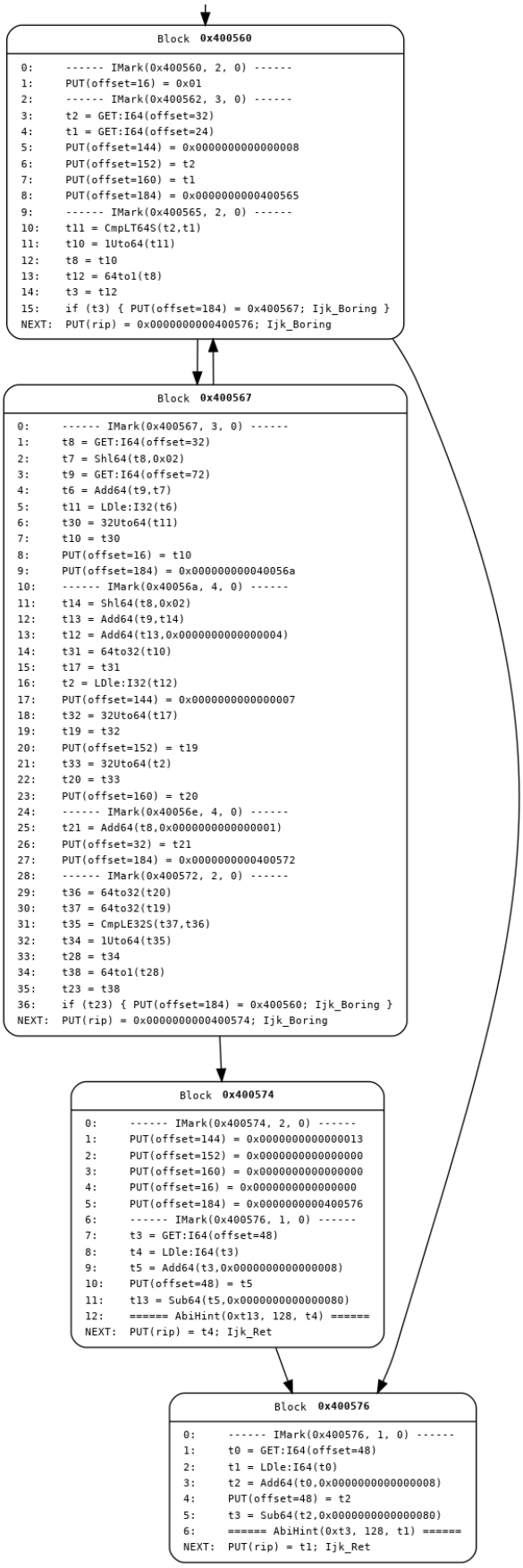

Figure 7: VEX CFG

VEX has some exciting features compared to the LLVM and BIL intermediate language since it performs a side-effect free lifting of the architecture-agnostic machine instructions. This feature makes the language hard to comprehend for a human reader. However, in computer-assisted analysis with angr, it can be used together with the Z3 theorem prover[50] from Microsoft to ensure that the lifting was performed without side effects and the input to Z3 is not missing anything essential after lifting.

2.2.4 Binary Ninja IL

The Binary Ninja intermediate language (abbreviated BNIL in the following) is, as the other ILs mentioned aboved, a representation of multiple architectures assemblies and is deeply integrated into the Binary Ninja reversing platform. It currently consists of two different representation, the Low-Level Intermediate Language (abbreviated LIIL in the following) and the Medium-Level Intermediate Language (abbreviated MLIL in the following). Both representations can be turned into a Single Static Assignment (SSA) representation (SSA is described in detail in section 2.3).

The following picture from the official Binary Ninja Documentation[17] displays the Binary Ninja intermediate language architecture:

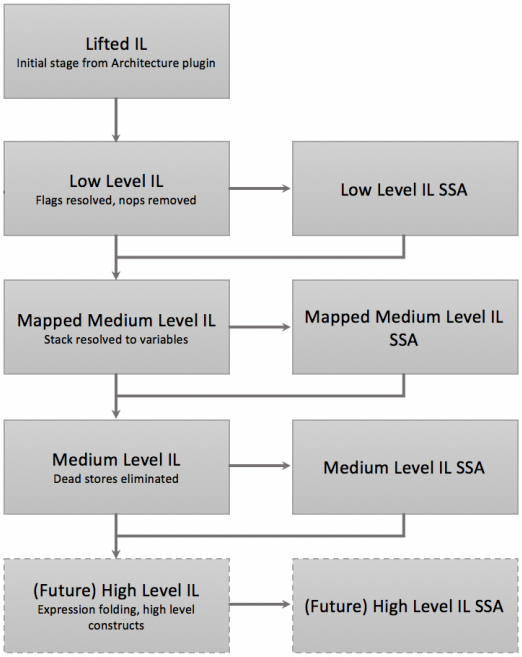

Figure 8: Binary Ninja Intermediate Languages[17]

When analyzing the program, Binary Ninja starts to lift the assembly code to it´s Lifted IL which then is optimized to the Low-Level Intermediate Language. Further, after lifting to the Low-Level Intermediate Language Binary Ninja performs several analyses to lift the

current representation to the Medium-Level Intermediate Language. Thus, such minor steps are referenced in the figure 9 as Mapped Medium Level Intermediate Language. Currently, Binary Ninja only lifts until the Medium-Level Intermediate Language, but in 2019 the very first releases of the High-Level IL are about to be released. The following bullet points will describe the different intermediate languages from Binary Ninja in detail:

Low-Level Intermediate Language. The Low-Level Intermediate Language was already shipped with the first public release of Binary Ninja. The following figure displays again the very same function is_sorted as BNIL LLIL:

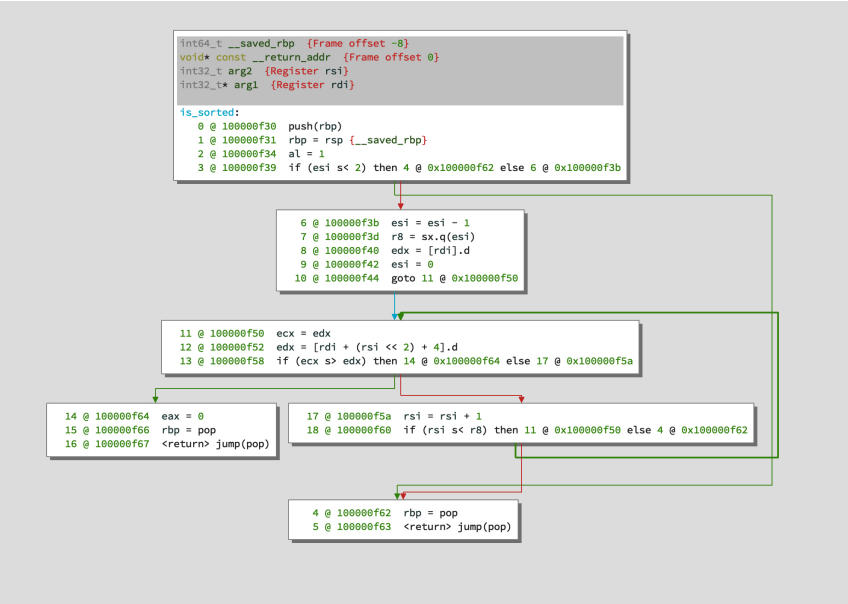

Figure 9: Binary Ninja Intermediate Languages I

It is notable that the registers are named after the assembly architecture (X86_64) which is similar to the BAP IL in section 2.2.2. However, it is already in a very human understandable form compared to the BIL language.

Medium-Level Intermediate Language. Binary Ninja improved the lifting to the Medium-Level Intermediate Language in the 1.0.776 release in June 2017. This representation improved the redability of the code drastically by resolving stack variables and eliminating dead storages. Such dead storages can be displayed as follows:

```
1 rax = 10
2 rbx = rax
3 printf('%i', rbx)
```

Listing 4: Eliminating Dead Storages I

The Medium-Level Intermediate Language will resolve this dead storage as follows:

```
1 printf('%i', 10)
```

Listing 5: Eliminating Dead Storages II

This makes the code way more readable and understandable which is displayed again with the is_sorted function but lifted to the Medium-Level IL:

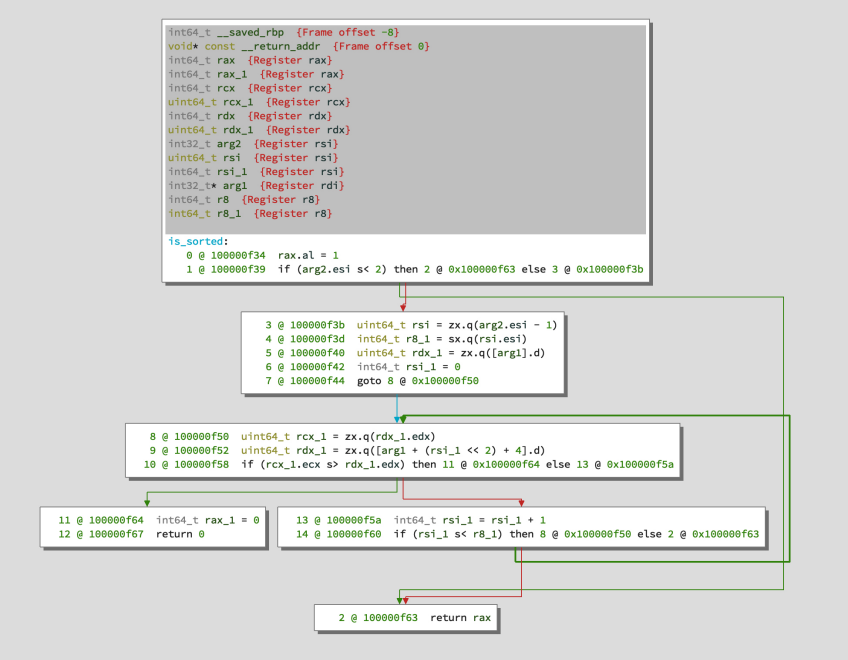

Figure 10: Binary Ninja Intermediate Languages II

The Medium-Level Intermediate Language of the is_sorted function is much easier to read than the Low-level Intermediate Language. However, there is still a step to take to match the IDA Pro decompiler[31].

High-Level Intermediate Language. The High-Level Intermediate Language is at the writing of this thesis still under massive development and was announced to land in the Binary Ninja release around 2019 to 2020. This intermediate representation is trying to match the IDA Pro decompiler Hex-Rays.

2.2.5 Conclusion Intermediate Languages

This chapter focused on different intermediate languages which can be used for automatic vulnerability discovery. The main decision criteria for an intermediate language to perform the analysis is the possibility for manual auditing after the analysis. Hence, the IL is required to have a very human-friendly representation. This criterion eliminates from the author´s perspective the VEX (described in section 2.2.3) and the BIL (described

in section 2.2.2) intermediate language due to their hard to read intermediate language. When comparing the leftover intermediate languages LLVM and the Binary Ninja Intermediate Language both are readable, but the Binary Ninja Intermediate Language has quite an edge over LLVM. Another important criteria is how accessible the language is for different programming languages through an API. Since the proposed framework language is Python, BAP was eliminated from the evaluated intermediate languages due to the limited accessibility for Python through the API.

The last remaining solution for the framework, the Binary Ninja Intermediate Language, provides the key requirements, i.e. readability and excellent bindings for Python. Another bonus point is the active development of the platform and that they are still improving their intermediate language and adding new architectures. The downside of the Binary Ninja platform is the payment fact which needs to be paid annually. However, compared to the competitor of IDA Pro the annual payment is significantly lower and still affordable for students and smaller companies.

2.3 Single Static Assignment

Single Static Assignment (abbreviated SSA in the following) is a property of an intermediate language and requires that each variable is assigned exactly once and is defined before it is used. This requires variables where a value is assigned multiple times to be split into multiple versions. Using the SSA Form, it is possible to eliminate dead code and perform loop optimizations and further transformations to emit a more efficient code. To briefly introduce the term dead code, sometimes developers have the habit of using unnecessary assignments or additional statements in their source code. Sometimes, this is owed to make the code more readable, but this habit would lower the performance of the software if the compiler would not optimize it. The following code excerpt illustrates such unnecessary assignments.

```
1  int add(int a, int b){
2    int result;
3    result = a + b;
4    return result;
5  }
```

Listing 6: Unnecessary Assignments

The function displayed in figure 6 can be optimized heavily by removing the unnecessary variable "result" and simply returning the final result of the addition from variable "a" with the variable "b".

To illustrate the usefulness of the SSA form the following pseudo-code displays how the SSA form helps in eliminating dead code and therefore increasing the overall speed of the software. The following pseudo-code is given:

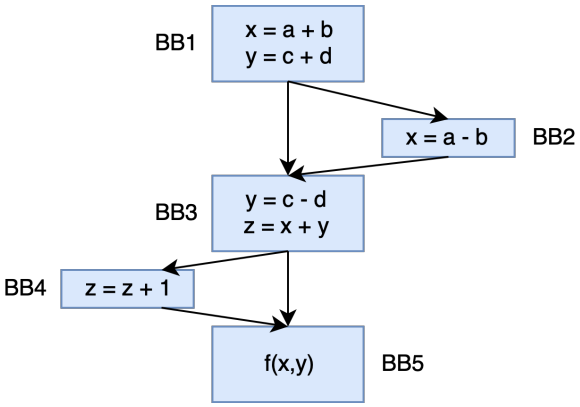

Figure 11: SSA dead code optimization I

Since this is quite a simple example, it is possible to spot some dead code blocks already. However, in very complex scenarios it is no longer possible to spot such dead code paths for developers or auditors. The very first step is transforming the graph in the SSA form which is displayed below:

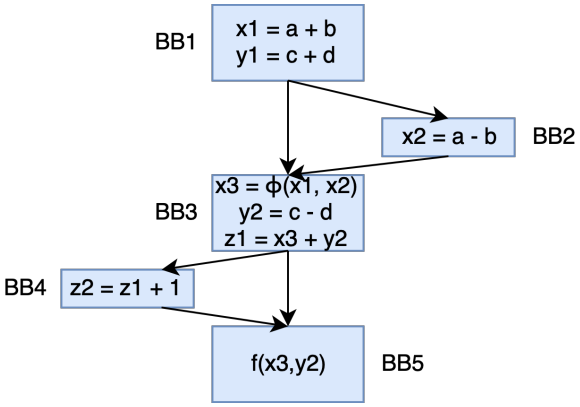

Figure 12: SSA dead code optimization II

The very first notable change is that every variable is now only assigned once and if it was assigned previously multiple times the versions are iterated counting from 1 upwards. The next notable change is the introduction of a new symbol in basic block 3 which is the greek symbol Phi (Φ). Phi is used whenever multiple control paths are resulting in the basic block where two or more blocks contain assignments to a variable used in the ending basic block. In this example, basic block 1 (BB1) and basic block 2 (BB2) booth assign values to the x variable which is used in basic block 3 (BB3) when assigning the z variable. Hence, it is unclear beforehand which x will be used for assigning z, and thus,

x3 is assigned with phi from x1 and x2.

Observing the resulting SSA graph, it is already possible to tell that, for example, the variable y1 in BB1 is never used and the whole BB4 with the assignment of z2 is not used anymore, and thus the assignment of z1 can also be deleted since it is dead code. The following graph displays the graph after the dead code optimization step:

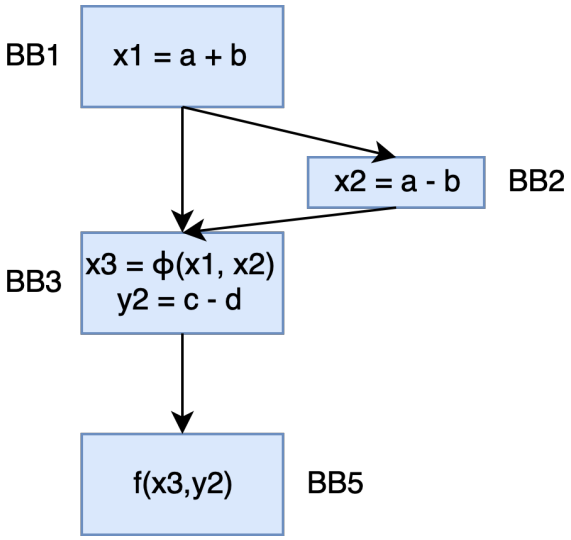

Figure 13: SSA dead code optimization III

Using the SSA form will be later one of the core features of the framework when analyzing compiled software.

2.4 Function Detection

A crucial part for analyzing black box binaries is to pinpoint where a function starts and where it ends precisely. Finding the beginning of a function is as most of the problems previously described also an undecidable problem. Due to this fact, there can not be a single algorithm which solves this problem. However, since Binary Ninja is already handling this problem quite well through their recursive decent and linear sweep algorithm, it is handy to know a few of these algorithms. Hence, they are briefly described in the following sections. It is noted that Binary Ninja includes a few more algorithm to detect functions such as Call Target Analysis and Tail Call Analysis. However, they are not covered in the following:

2.4.1 Recursive Descent

Early versions of Binary Ninja included solely the Recursive Descent algorithm for function detection and was limited due to this fact.

The Recursive Decent algorithm starts to parse from the defined entry point of the binary and follows all function calls to detect functions accurately. This approach, however, is quite limited due to possible member function calls from classes which frequently occur in C++ applications. Such calls are implemented by the C++ compiler using Virtual Tables (VTables).

Further, some functions can be disconnected from the direct control flow or performing tail calls where the return address is dynamical constructed. Thus, the Recursive Decent algorithm cannot find them.

2.4.2 Linear Sweep Algorithm

The Linear Sweep algorithm is a straightforward way to approach function detection. The basic idea is to scan every as an executable marked section from top to bottom and identify possible function prologs. This is usually a heuristical approach where the different function prologs of the available calling conventions are matched against the disassembly. Due to this approach, the missing functions from the previously introduced Recursive Descent algorithm can be caught using the Linear Sweep approach.

2.5 Abstract Syntax Tree

An Abstract Syntax Tree represents syntax of the source code in a hierarchical tree structure. Compilers mainly use this approach. However, decompilers are using the same approach in a reverse order to lift the code from assembly to their intermediate representation. This approach is heavily used by the Binary Analysis Platform which was introduced in chapter 2.2.2. To display how an Abstract Syntax Tree is structured the following simple source code is given:

```
1  if (x < y) {
2    z = 1;
3    return z;
4  } else {
5    z = 2;
6  }
```

Listing 7: AST Source Example

The following graph displays a generated AST from the previously shown source code:

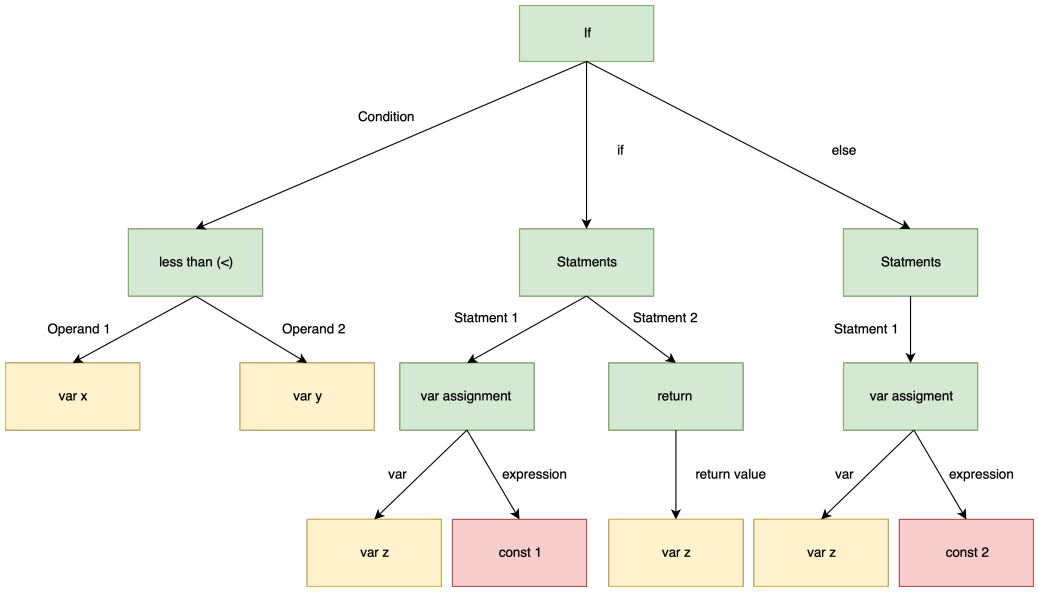

Figure 14: AST Example

Such Abstract Syntax Trees can be walked quite efficiently concerning a compiler to emit code to either another language or to emit assembly code while for decompiler reconstructing an AST can be beneficial for graph vulnerability modeling.

2.6 Depth-First Search Algorithm

The depth-first search algorithm (abbreviated DFS in the following) is an algorithm used for traversing a graph. The DFS search algorithm tries to explore first as far as possible along a branch before it performs the backtracking. It should be noted that this is the opposite to the breadth-first search (BFS) algorithm.

How the DFS algorithm works is illustrated on the following graph:

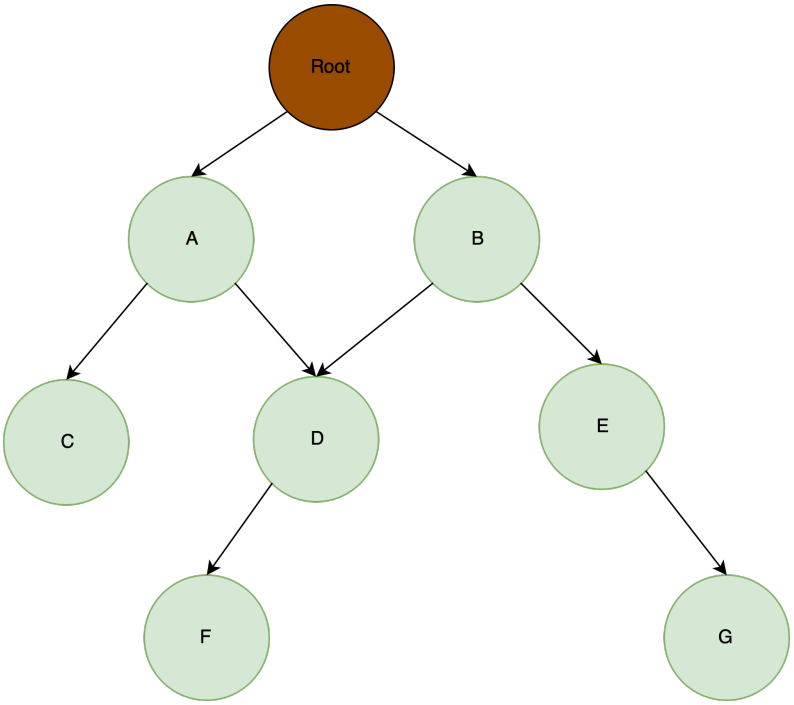

Figure 15: DFS Example

The algorithm tries to dive first deep into the branches. On which side the algorithm starts is usually depends on the implementation, but usually, the left branch is the starting branch. Then the algorithm tries to work to the left edge as deep as possible and then starts to backtrack and take the other branch if possible. Thus, sticking to the example displayed in figure 15 the output of the DFS search starting at the "Root" node would be the following set (Root, A, C, D, F, B, E, G) when the algorithm prevents tracking a previous walked path (This is used to prevent being stuck in a loop).

This DFS algorithm is used in the proposed framework for several different plugins to perform analysis on a given function graph.

2.7 Concolic Execution

Due to the massive prize pool of the Cyber Grand Challenge which was hosted by the Defense Advanced Research Projects Agency (DARPA) pushed the development of concolic execution frameworks such as angr [54] and MAYHEMs[29] Concolic Execution Client (CEC) and Symbolic Execution Server (SES). Due to the free availability of angr the following examples of concolic execution is described on how angr handles it. Concolic execution combines both symbolic and concrete execution while the basic idea here is that the concrete execution drives the symbolic execution. Angr uses for the concolic

execution engine the intermediate language VEX which was introduced in chapter 2.2.3. Usually, a program needs some input to branch into different parts, but in the case of concolic execution, the symbolic execution engine tries to fulfill with calculated input every possible path. The downside effect of this concolic execution is, of course, an immense path explosion on very complex programs. However, the following example from the angr FOSDEM presentation [14] is used to make the use of concolic execution more clear:

```
1 x = int(input())
2 if x >= 10:
3   if x < 100:
4     print "You win!"
5   else:
6     print "You lose!"
7 else:
8   print "You lose!"
```

Listing 8: Concolic Execution Example

To achieve the "You win!" condition in the above-listed pseudo code, it is required to input a number between 10 and 99. This appears to be evident for a human reader, but this example can be performed in very complex calculations where it stops the be easily understandable for humans. Thus, Symbolic Execution can take over and solve equations to generate valid input to hit the code path for a searched condition. The following graphs display how the Symbolic Execution engine deals with this case.

When executing the above-given pseudo-code in line 1 the symbolic Execution engine generates a symbolic variable since the input is unclear at this time and there are no underlying constraints to this input at that time. The below graph displays the generation of such a symbolic variable:

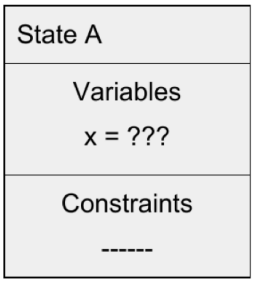

Figure 16: Initial State

The second row of the pseudo code is an if statement which is a branch statement. When a

branch is possible the Symbolic Execution engine generates a new state for every possible branch. Since this, if condition branches on a specific condition on the "x" variable the engine set a constraint on the generated symbolic variable. The following figure displays the branching and the constraint on the symbolic variable:

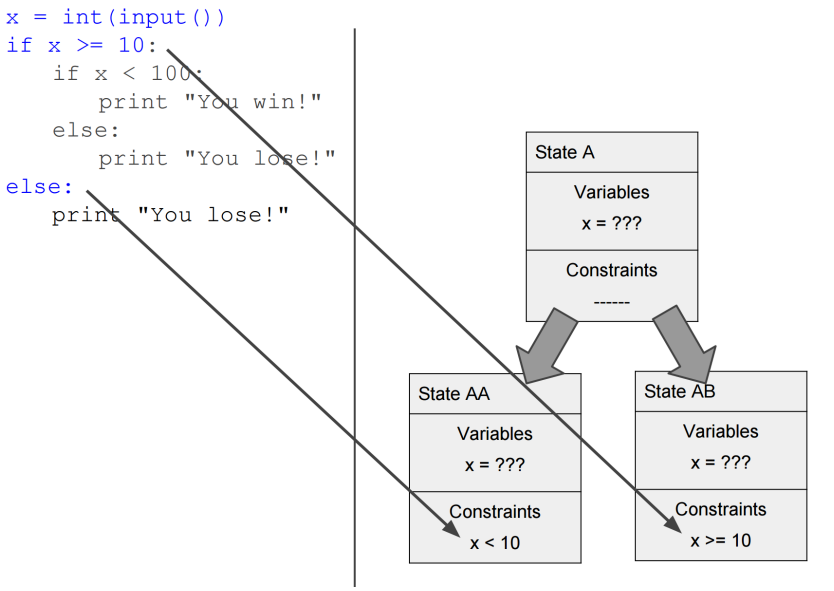

Figure 17: First Branch State

The third line of the pseudo-code branches the code again with another if statement and again applying constraints to the "x" variable. Referencing the above state graph. The state "AB" needs to be branched again. Thus, when branching and applying the constraints accordingly to the if condition the following state graph is the result:

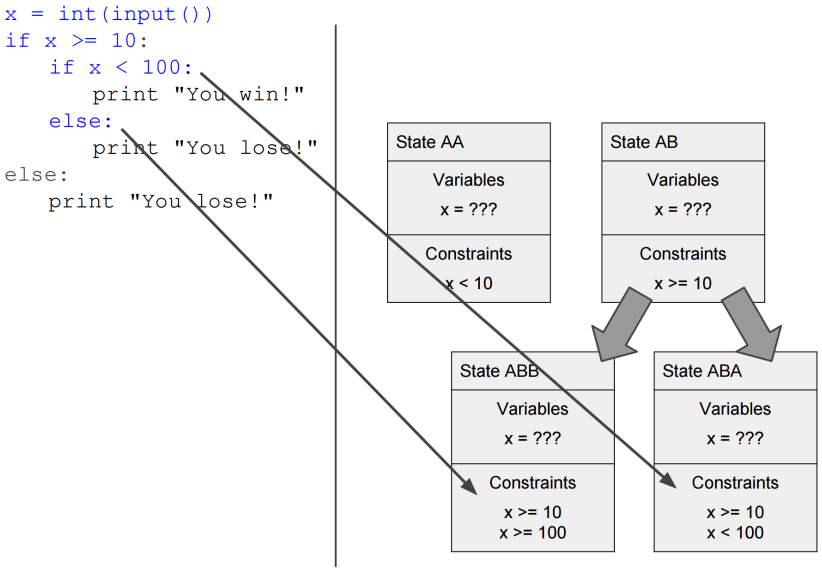

Figure 18: Second Branch State

When the concolic execution reached the desired end branch or code line, it is possible to concretize a symbolic value to solve the input which is responsible for taking a path through the program. Angr does concretize these values with the help of Microsofts Z3 [50]. The following graph displays an exemplary value when concretizing a symbolic variable with Z3 (remember a value between 10 and 99 is possible):

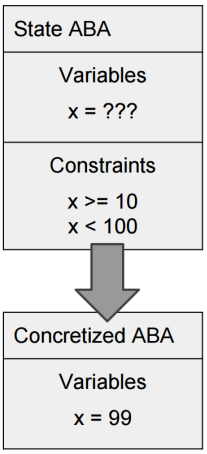

Figure 19: Concretized State

3 Analysis

This chapter introduces the concepts and approaches used for analyzing software and discovering vulnerabilities. Section 3.1 describes different analysis techniques and evaluates them which technique fits for the proposed framework. Finally, chapter 3.2 introduces Taint Analysis while chapter 3.4 describes several vulnerability patterns.

3.1 Approaches for Vulnerability Discovery

Common approaches to analyse software for security vulnerabilities are such as source code reviews, static analysis, and fuzzing. Since, this thesis focuses on already compiled software source code reviews can not be performed, instead reverse engineering is needed which can result in a very tidious and time consuming task. In this section the above described most popular approaches are described and evaluated whether it is appliaple considering the advantages and limitations.

3.1.1 Reverse Engineering

Reverse Engineering can be defined as:

> "Reverse engineering is the process of comprehending software and producing a model of it at a higher abstraction level, suitable for documentation, maintenance, or reengineering"[56]

A classic approach for finding vulnerabilities is to analyse the compiled software by reverse engineering it. Reverse engineering is a very time consuming method since due to compiling there is often a loss of quite some information such as comments inside the code and symbols when the software is delivered in a stripped state.

This forces the analyst to roughly follow this process.

Understanding the Overall Architecture. Since there is often only a user and no developer documentation it is necessary to understand the technical architecture by trying to trace where the processing of the input is and how input is passed to the software and where different trust zones between components are implemented.

Recovering Function Names. If the software is stripped, it is a necessary step to try to recover function names to generate a better understanding of the control flow of the program. This can sometimes be achieved due to the left in debug output otherwise the analyst has to name the function according to his ability to understand what the function does.

Recovering Structs and Classes. depending if the program is compiled from c or c++ it is a tedious step to recover the used structs and classes. It is possible to support this step with automated tools in the case of c++ due to Run-Time Type Information (RTTI) Software. However, the step of recovering structs and classes goes hand in hand with the following process.

Analyzing Control Flow and Hunting for Bugs. This step requires the previous steps to successful hunt for bugs. Here it is necessary to analyze functions in the control flow for processing input in depth. This is the most challenging and time-consuming task.

Overall, manual reverse engineering is a time-consuming task where time is often the problem to find vulnerabilities in software. Therefore, reverse engineering will always be a way to find vulnerabilities in a time-limited environment, but it is not well suited for this thesis object.

3.1.2 Fuzzing

Fuzzing has variable definition such as:

> "highly automated testing technique that covers numerous boundary cases using invalid data (from files, network protocols, API calls, and other targets) as application input to ensure the absence of exploitable vulnerabilities better"[44]

Writing such Fuzzers can range from very easy and fast to implement fuzzers to very complex programs which can run fully automated and can even submit discovered problems directly to the vendors such as Syzkaller [4].

Fuzzer can currently be split into three categories where there are also existing hybrid fuzzers between the various categories. The first category is dumb fuzzers, where the fuzzer does not know anything about the input and internals of the program whatsoever. They are also sometimes referred to as black box fuzzers. Such dumb fuzzers only contain algorithms to mutate on any given data. A mutation of a dumb fuzzer which contains more knowledge about the input is, for example, a fuzzer called Radamsa [5]. Radamsa can get a grasp of different values inside any given input such as integers or strings and mutates the recognized patterns more accurately.

In contrast, a grammatic based fuzzer requires information on the structure or grammar used for the inputs. It generates new input based on a given grammar and mutates the generated input. A grammatic based fuzzer which had success in the past is for example domato[6]. This grammatic based fuzzer generates valid input for a browser to

display a webpage (Document Object Model) and was responsible for quite some security vulnerabilities[6].

In the past few years, another more complex way to fuzz software was introduced. These fuzzers are often called genetic or feedback-driven fuzzer. Most notably fuzzers are American Fuzzy Lop[7] and Honggfuzz[8]. These fuzzers traverse through the program and generate more specialized input over time. This intelligent generation is achieved due to the feedback the software delivers back to the fuzzer on every input cycle. These genetic fuzzers are responsible for a bulk amount of vulnerabilities in the past years. A further improvement of genetic fuzzers is the combination with a symbolic execution engine such as angr[54] which can help a stagnating fuzzer in finding new input by resolving input for a specific path which was undiscovered by the genetic algorithm. This approach when combining genetic fuzzers and symbolic execution was presented in the driller whitepaper [55] by shellphish.

Big companies such as Google with ClusterFuzz [9] already include such fuzzers into their lifecycle and promoting their development.

The described success of fuzzers seems a good approach for finding security vulnerabilities in software. However, there are downsides of fuzzing:

Stateful Interface. Fuzzers have a hard time to maintain a state which requires a sane way of initialization. As long as the fuzzer does not understand the initialization process entirely a potential security issue cannot be found. These problems have of course been bypassed in the past but requires heavy adjusting of the fuzzer.

Missing Certain Vulnerability Types. Fuzzers can miss out on different vulnerability types such as race conditions or where, due to a state in the application, different input needs to be send in the correct order to trigger a vulnerability such as a Use-After-Free or Double-Free.

In summary, fuzzing is an excellent way of finding vulnerabilities and should always be considered first, but it is not well suited for every vulnerability.

3.1.3 Static Analysis

An alternative approach to manual analysis is static analysis. Static analysis supported by tools to find security vulnerabilities is already mentioned in IEEE papers and can be defined as follows:

> The tool basically scans through a file looking for syntactic matches based on several simple "rules" that might indicate possible security vulnerabilities (for example, use of strcpy()should be avoided). Much better approaches exist. [41]

While compilers usually use static analysis to perform compile-time optimizations, the same technique can be performed to find security vulnerabilities. Usually, this static analysis is used on the source code, but it is also possible to use this technique on compiled software which is described in detail in this thesis. However, in practice, the loss of information when compiling a software makes hunting for bugs in already compiled software extremely difficult and complicated. Furthermore, when performing taint analysis, it is possible to run into certain problems such as state explosions. While this problem needs to be dealt with more heavily when adding symbolic execution to the analysis, it can also occur during the standard static analysis.

Further, it should be mentioned that due to state explosions and the overall complexity static analysis is not a valid approach for large and complex targets such as a browser. Even though they will always contain security related bugs these problems are insanely hard to spot using static analysis. The following to examples of current browser bugs demonstrate the problem of automatically spotting these bugs. A wide-spread bug class for browsers is the Use-After-Free vulnerability. Besides their name, these vulnerabilities usually originate from the nature of JavaScript to trigger synchronous events on changes to objects for example. Such relations are sometimes overlooked by the software developer, who does not expect a change of the object while the code is running. Such a misassumption can then result in a Use-After-Free bug. Integrating a book-keeping mechanism into the static analysis approach of all points where callbacks to a JavaScript handler or proxy are triggered is not feasible since the tool would need to consider every single edge case.

A second example would be the nature of bugs residing in the just in time compiler inside browsers. Browsers use the functionality of a (Just in Time compiler) to improve performance when executing JavaScript. If a function is called quite often and thus the amount of calls hits a threshold the browser will determine this function to be jitted resulting in a compilation from JavaScript to assembly instructions. It is not possible to analyze such jitted function statically before they are compiled. Thus, the only way of

analyzing this automatically would require to analyze the algorithms how the JavaScript code is jitted which is again arbitrarily complex and results again in the halting problem.

Besides the described downsides, the static analysis approach can discover more bugs compared to runtime discovery (e.g., Fuzzing). Since it is not required to execute the software, more unique and edge case code paths are analyzed while performing the analysis. Since running the analysis on the final compiled software, it is possible to discover vulnerabilities which are introduced during the compile time which are extremely hard to spot performing a manual source code review. Such vulnerabilities are often race conditions, for example, Time-of-Check-Time-of-Use (TOCTOU) or double fetches problems where the software developer forgot to declare the given input as a shared resource (volatile).

To discover such bugs in complex scenarios, it is required to write several different analyses. The writing of such patterns on an intermediate language is part of this thesis to discover a variety of security problems in components which are often not audited such as IoT devices.

3.2 Taint Analysis

The following definition will be used for the dynamic taint analysis:

> Dynamic taint analysis (DTA), also called data flow tracking (DFT), taint tracking, or simply taint analysis, is a program analysis technique that allows you to determine the influence that a selected program state has on other parts of the program state. [13]

Taint Analysis can be performed statically or dynamically by executing the software. While both have their advantages and disadvantages, it is useful when modeling vulnerabilities.

Static Taint Analysis. The advantage of the static taint analysis is as the name suggests that everything is performed statically which is in the case of malware a safer approach compared to the dynamic analysis. Further, due to the static analysis, every possible code path can be considered instead of only the executed path while the disadvantage can be primarily found in C++ compiled programs. Due to inheritance and polymorphy, it is often not possible to follow the execution flow for the taint analysis statically.

Dynamic Taint Analysis. As previously mentioned advantage of the DTA is the possibility to trace the control flow containing inheritance and polymorphism. However, executing arbitrary binary code can be hazardous when dealing with malware.

Since the drastic improvement of symbolic execution engines (described in detail in section

2.7) it is possible to emulate a dynamic execution better and reach more code paths with the taint.

Since binary ninja provides the API for the static analysis, the framework will deal with the Static Taint Analysis. However, it is planned to include the angr framework [54] into the analysis which enables the use of symbolic execution to discover the control flow in C++ software.

3.2.1 Sources and Sinks

Sources. Taint sources can be defined as follows:

> Taint sources are the program locations where you select the data that's interesting to track. For example, system calls, function entry points, or individual instructions can all be taint sources. [13]

Further sources are for example network or direct user input. In some cases, environment variables need to be considered as well. This is often the case in so-called Common Gateway Interface (CGI) scripts where the input is delivered to the application from the web server via environment variables or can be directly manipulated in privilege escalation scenarios.

Sinks. Sinks in the context of vulnerabilities are common problematic functions such as memcpy or malloc. The memcpy function call exposed by libc or vc++ is quite often prone to a wrong usage which results in an exploitable condition. Further, if an attacker has the possibility to arbitrary change the bytes allocated for an object it can also lead to exploitable conditions in the later usage. Hence, tracking possible input from sources to sinks can discover bugs in software.

Taint Analysis combined with sources and sinks models the control flow from an identified source until it finds a sink. This approach can, of course, be performed backwards where first a possible sink is identified and then sliced backwards until it finds a valid source.

3.3 Code Coverage

Code Coverage is a typical term when developing software and is used to measure how much of a particular program is executed for given inputs. Test teams try to aim for as much code coverage as possible to discover bugs and problems. However, when it is not possible to access the source code to test for the code coverage dynamic binary instrumentation (abbreviated DBI in the following) frameworks are needed.

The following describes three possible approaches to gather code coverage with dynamic binary instrumentation frameworks (It should be noted that more frameworks exist).

DynamoRIO.

The following quote is from the official DynamoRIO[28] documentation.

> DynamoRIO is a runtime code manipulation system that supports code transformations on any part of a program, while it executes. DynamoRIO exports an interface for building dynamic tools for a wide variety of uses: program analysis and understanding, profiling, instrumentation, optimization, translation, etc.

Intel PIN.

The following quote is from the official PIN[33] documentation.

> Pin is a dynamic binary instrumentation framework for the IA-32, x86-64 and MIC instruction-set architectures that enables the creation of dynamic program analysis tools. [...] As a dynamic binary instrumentation tool, instrumentation is performed at run time on the compiled binary files. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code.

Frida.

The following quote is from the official Frida[49] documentation.

> Frida is a dynamic code instrumentation toolkit. It lets you inject snippets of JavaScript or your own library into native apps on Windows, macOS, GNU/Linux, iOS, Android, and QNX.

The above described DBI frameworks are all very different, but they all have in common a plugin to export the collected code coverage to the drcov[27] file format.

The proposed framework got the ability to parse these coverage files using the lighthouse[38] drcov parser [39]. This feature enabled the prioritization of discovered vulner-

abilities when a given input hit the vulnerable function/instruction which was logged by a DBI framework.

3.4 Vulnerability Patterns

This chapter introduces different vulnerability patterns beginning with Buffer Overflows following with numeric over- and underflows. Next, problematic type conversions are described and, finally, the abuse of uninitialized data is introduced.

3.4.1 Buffer Overflows

The following reference defines the term buffer overflow:

> a buffer overflow is a software bug in which data copied to a location in memory exceeds the size of the reserved destination area. When an overflow is triggered, the excess data corrupts program information adjacent to the target buffer, often with disastrous consequences. [52, p.180]

Buffer overflows can appear anywhere in a program and the exact root cause may vary between different occurrences such as integer overflows and type confusions. It is not unlikely that a buffer overflow is the result of a different bug such as an Integer overflow (described in chapter 3.4.2) or a conversion issue (described in chapter 3.4.3). Further, complex problems like a Type Confusion bug can as well introduce a buffer overflow. However, besides the just described issues buffer overflows also can have their unique pattern. Some of the well-known patterns are introduced in the following paragraphs.

memcpy. The exposed memcpy function is quite often a sink for buffer overflows. The function call looks like this according to the manpages:

```
1 void* memcpy(void *restrict dst, const void *restrict src, size_t n);
```

Listing 9: memcpy

The call is very simplistic and copies n amount of bytes from the given source pointer (*src) to the given destination pointer (*dst). However, besides its straightforward purpose, it is often prone to vulnerabilities since either the src buffer can be potentially bigger than the destination buffer, and the calculation for the n parameter is based on the source buffer instead of the destination buffer or sometimes the calculation for the n parameter can be either merely wrong or contain a conversion problem. Since the function expects a size_t it implicitly converts the input parameter to an unsigned integer, which often results in the mentioned conversion problem. The behavior of conversion errors is analyzed in chapter

3.4.3. Finally, the memcpy call exposes very often an exploitable bug by off-by-one errors where the calculation for the n parameter is exactly wrong by one byte. An example for exploiting such an off-by-one error was presented by j00ru in a Windows 10 Paged Pool off-by-one overflow [35].

Given the above descriptions the following defined example is a typical real world pattern for a misuse of memcpy (Note: in the following example the password is user controlled):

```
1  static void add_password(AUTH_HDR *request, unsigned char type, CONST
     char *password, char *secret)
2  {
3      MD5_CTX md5_secret, my_md5;
4      unsigned char misc[AUTH_VECTOR_LEN];
5      int i;
6      int length = strlen(password);
7      unsigned char hashed[256 + AUTH_PASS_LEN];
8      unsigned char *vector;
9      attribute_t *attr;
10
11     if (length > MAXPASS) {
12         length = MAXPASS;
13     }
14
15     if (length == 0) {
16         length = AUTH_PASS_LEN
17     } if ((length & (AUTH_PASS_LEN - 1)) != 0) {
18         length += (AUTH_PASS_LEN - 1);
19         length &= ~(AUTH_PASS_LEN - 1);
20     }
21
22     memset(hashed, 0, length);
23     memcpy(hashed, password, strlen(password))
24     [...]
25 }
```

Listing 10: memcpy Vulnerability

Reading through the above-given source code appears first a bit elaborate. However, when inspecting the last line 23 with the occurring memcpy, instead of passing the maximum length of the destination buffer the length of the source buffer is taken. This of course results in a buffer overflow if the password is longer than 256 + AUTH_PASS_LEN. This exact pattern is an often occurring mistake by developers and does have critical effects on the application.

strcpy. Nowadays the strcpy function does not play a significant role concerning vulnerabilities anymore due to widespread knowledge of being a vulnerable function. Nevertheless, since this function introduces severe vulnerabilities in the past, we will analyze it here. The following code displays the function prototype according to the manpages:

```
1  char* strcpy(char * dst, const char * src);
```

Listing 11: strcpy

The strcpy function will copy byte by byte from the given source (*src) string to the destination buffer (*dst) until the function finds a null byte in the source buffer. Hence, if a source buffer is given to the function which is bigger than the destination buffer, strcpy writes over the bounds of the destination buffer.

Further, another typical problem are off-by-one errors for the strcpy function. Where for example the source buffer is not appropriately terminated by a null byte. For a program that uses many stack operations, it is possible that the strcpy function overwrites the destination buffer either with random data or some confidential data such as stack cookies which might leak through a buffer.

Since explained earlier strcpy is not that common anymore to find in a vulnerable state. Compilers also do usually emit warnings when using the unsafe strcpy function. However, sometimes it is possible to find strcpy like functions which have been written by the developers to perform a specific task. For example, instead of copying bytes until a null byte is found, they might want to split a string based on a specific character. The following example displays such a strcpy like function:

```
1  if (recipient == NULL && Ustrcmp(errmess, "empty address") != 0)
2  {
3    uschar hname[64];
4    uschar *t = h->text;
5    uschar *tt = hname;
6    uschar *verb = US"is";
7    int len;
8    while (*t != ':') *tt++ = *t++;
9      *tt = 0;
```

Listing 12: strcpy like Function

The presented source code presents a straight forward buffer overflow. When the source buffer "h->text" is bigger than "hname" (64 bytes) without containing a ":" char it will overflow the local stack variable.

Similar problems can also be found in conversion or expansion functions.

Note: Dealing with such manually written function is covered in a particular case in the buffer overflow plugin for the framework which is described in chapter 5.3.2.2.

sprintf. Working on format strings with functions such as sprintf can have multiple pitfalls. First, defining no format string can result in a format string bug. However, those are not very common nowadays. Second, using non-delimited format string modifiers can often lead to buffer overflows such as using "%s" on user-controlled input. Even though compilers usually emit warnings on these dangerous use of "%s" it is still possible to miscalculate delimiter lengths like "%12s" when chaining multiple together.

Given the above descriptions, the following example is a typical real-world pattern for misuse of sprintf. This designed example was found similarly in a malware emulation software and was reversed manually. The following function should emulate when the malware tries to connect to an IRC command and control server. (Note: in the following example the nick_name variable is user controlled):

```
1 char buf[1024];
2
3 sprintf(buf, ":server 1 %s : Welcome %s %s!%s@%s\r\n",
4         nick_name,
5         destination_IP,
6         nick_name,
7         source_IP
8 );
```

Listing 13: sprintf Vulnerability

If the attacker malware chooses an IRC nickname which is longer than 512 bytes (not counting the possible length of source and destination IP address), this results in a straight stack-based buffer overflow.

There are quite some buffer overflow patterns. However, the covered patterns should give a basic idea about them.

3.4.2 Numeric Overflows and Underflows

Numeric overflows, also called Integer Overflows, are a well known problem for software developers. However, they are sometimes very subtle and hard to handle especially in C and C++ applications since C requires the programmer to track quite some values such as memory allocation lengths and buffer lengths. Thus, it is quite common to find such

vulnerabilities in a program when it is required to perform arithmetic operations. The following definition does pinpoint the impact of such numeric issues.

> The incorrect result of an arithmetic operation can undermine the application's integrity and often result in a compromise of its security. A numeric overflow or underflow that occurs early in a block of code can lead to a subtle series of cascading faults; not only is the result of a single arithmetic operation tainted, but every subsequent operation using that tainted result introduces a point where an attacker might have unexpected influence. [52, p.225]

The following real world vulnerability from the OpenSSH server in version 3.1 displays the problem when handling allocation values (Note: The value in "nresp" is user controlled):

```
1  u_int nresp;
2  [...]
3  nresp = packet_get_int();
4  if (nresp > 0) {
5    response = xmalloc(nresp * sizeof(char*));
6    for (i = 0; i < nresp; i++)
7      response[i] = packet_get_string(NULL);
8  }
9  packet_check_eom();
```

Listing 14: OpenSSH Integer Overflow Vulnerability

The C code given above exposes a critical vulnerability. The function tries to allocate the value calculated from the user given input "nresp" and multiplies it with the size of a char pointer which is on 32 bit systems 4 bytes (8 bytes on 64bit systems). Hence, when the value in "nresp" is bigger than 0x40000000, the result will wrap around. When the value in "nresp" is, for example, 0x40000010 the wrap around result is 0x40 bytes (nresp is defined as an unsigned integer), and xmalloc does allocate these 0x40 bytes. However, the following loop will iterate 0x40000010 times and copies given user input bytes to the 0x40 bytes allocated buffer. This integer overflow results in a very critical heap overflow which was exploitable in the past.

3.4.3 Type Conversions

The following reference describes the term type conversions:

> There are two forms of type conversions: explicit type conversions, in which the programmer explicitly instructs the compiler to convert from one type to another by casting, and implicit type conversions, in which the compiler does "hidden" transformations of variables to make the program function as expected. [52, p.237]

Since C and C++ are so flexible it is possible to let different datatypes interact with each other. However, there are some basic rules when performing these interactions that are often ignored or simply overlooked. Due to this the resulting vulnerabilities are often very subtle but also very critical.

The following bulletpoints do introduce the very basics of conversion rules for numbers:

Value-Preserving Conversions. A value preserving conversion is straightforward. For example, when converting a signed char to a signed integer the compiler performs a value preserving conversion since a signed char can range from -128 up to the value of 127 while a signed integer can contain values from -2147483648 up to 2147483647. Thus, any value which the signed char contains can exist in the signed integer.

Value-Changing Conversion As the name suggests, this conversion rule is applied when the converted number does not directly exist in the target type. For example the unsigned integer type ranges from 0 to 4294967295. Thus, when converting it to a signed integer it is impossible to preserve for example the value 4294967295. Such type conversion issues result in a number of interesting vulnerabilities over the years, where the problem was overlooked, for example, within implicit conversions.

The complete ruleset of these conversions can either be looked up online in the official documentation or in the recommended book: The Art of Software Security Assessment[52].

The following example code displays a very common problem when dealing with type conversions:

```
1 int read_user_data(int sockfd) {
2   int length, sockfd, n;
3   char buffer[1024];
4   length = get_user_length(sockfd);
5   if(length > 1024){
6     error("illegal input, not enough room in buffer\n");
7     return 1;
8   }
```

```
9   if(read(sockfd, buffer, length) < 0){
10    error("read: %m");
11    return 1;
12  }
13  return 0;
14 }
```

Listing 15: Type Conversion Vulnerability

In line 2 the length variable is defined as a signed integer and in line 4 user controlled input is converted to the signed integer. Line 5 performs a security check whether to length variable contains a number bigger than 1024. Finally in line 9 the read operation reads the amount of bytes defined in the length variable to the buffer. At first this looks fine.

However, since length is defined as a signed integer it is possible to pass a negative number into the variable. This will pass the check if length is bigger than 1024 since it is negative.

The problematic type conversion happens in the read function. The following excerpt displays the read function prototype:

```
1 ssize_t read(int fildes, void *buf, size_t nbyte);
```

Listing 16: read Prototype

While it is not directly visible but the last parameter "nbyte" is from the size_t type which links directly to an unsigned integer. Hence, when passing a variable to this parameter, it will be implicitly converted to an unsigned integer. Since this conversion is a value changing conversion, it can result in a huge stack-based overflow. When supplying the value "-1" in the length variable, it will be converted to the large number of 4294967295 inside the read function. Thus, resulting in a stack-based overflow.

3.4.4 Uninitialized Data

The following quote defines uninitialized data well:

> Standalone variables of simple types such as char or int, as well as members of larger data structures (arrays, structures and unions) remain in an indeterminate state until first initialized, if they are stored on the stack (formally under automatic storage duration) or heap (allocated storage duration). [34]

The concept of uninitialized data is simple. When defining variables, they should always be initialized.

Forgetting to initialize data is often referred to as not a security problem. While this is true in most cases, there are edge cases in different scenarios which can impact the overall security. Attackers need specific primitives to exploit uninitialized data such as the possibility to trigger a particular function several times or the possibility to grow either the stack or the heap in a very controlled fashion. Since this limits the applicability to, for example, OS Drivers/Hypervisors/Browsers and PDF reader, these bugs are often overlooked.

The impact of exploiting uninitialized data can sometimes lead to severe vulnerabilities which detour the control flow to execute arbitrary code. The following real-world example from the Pwn2Own[48] contest submitted by 360 security[10] in 2018 allowed the breakout from a virtual machine (Guest) to the hypervisor (Host) [32].

The following function in the Super Video Graphics Array (SVGA) interface suffers from an uninitialized variable:

```
1  void __fastcall sub_1401524B0(int a1)
2  {
3      DWORD *v1; // rax
4      char v2; // [rsp+20h][rbp-58h]
5      unsigned int v3[2]; // [rsp+88h][rbp+10h]
6
7      if ( a1 == 1 )
8      {
9          sub_140535530(v3, &unk_140CA1178);
10         while ( v3[1] )
11         {
12             v1 = (DWORD *)sub_140535060(v3, (__int64 *)&unk_140CA1178);
13             if ( v1 )
14             {
15                 if ( v1[2] & 4 )
16                 {
17                     sub_140151C20(1, *v1, 1i64, (__int64)&v2);
18                     sub_1403B69B0(&v2);
19                 }
20             }
21         }
22     }
23 }
```

Listing 17: VMware Breakout I[32]

In this extraordinary case the variable "v2" should be pointing in the normal case to the

beginning of a struct and is used to call function "sub_140151C20" as a parameter and directly beneath used as the first parameter to the function "sub_1403B69B0". Usually the function "sub_140151C20" initializes the "v2" variable. However, if it fails "v2" is not touched and remains uninitialized when passed to the function "sub_1403B69B0".

The following displays the function "sub_1403B69B0" where the uninitialized variable "v2" is passed as an argument:

```
void __fastcall sub_1403B69B0(__int64 a1)
{
    int v1; // eax
    unsigned int v3; // edi
    __int64 v4; // rbp
    __int64 v5; // rsi
    __int64 v6; // r12
    signed __int64 v7; // rax

    v1 = *(_DWORD *)(a1 + 12);
    if ( v1 == 1 )
    {
        if ( *(_QWORD *)(a1 + 24) >= 0xFFFFFFFFFFFFFFF8ui64 )
        {
            if ( *(_BYTE *)(a1 + 40) & 2)
            {
                if ( *(void **)(a1 + 16) != qword_140DA2DD8)
                {
                    sub_140014FD0(139i64, 1i64);
                    free(*(void **)(a1 + 16));
                    [...]
```

Listing 18: VMware Breakout II[32]

In the last line of the presented function, the free function is called on an offset from the "v2" variable. It might not be directly apparent what exploit primitive this brings, but this can be leveraged to force a free on a referenced object. Hence, this can be changed into a Use-After-Free primitive. Exploiting this bug was not easy since it requires meticulous stack grooming with calling different exposed functions from the driver to leave a valid pointer to an object on the stack for freeing. Thus, it will result in a type confusion when another function allocates on the previous free object. This was first used to create an arbitrary leak to bypass protection mechanism like ASLR and later turned into an arbitrary write to gain code execution on the hypervisor.

Explaining the exact exploitation steps mentioned above and the different terms of this

primitive are out of the scope of this thesis. However, it points out how forgetting to initialize a single variable can have disastrous effects on the security and result in a compromise of the hypervisor.

4 Design

Based on the evaluation in chapter 3, we concluded that static analysis is a promising approach when discovering vulnerabilities in different architectures in IoT devices when using the Binary Ninja Intermediate language. In this chapter the proposed design of our framework to discover different vulnerabilities is presented. In the next section a description of the Framework is given together with a walkthrough on how the proposed framework executes the different analysis plugins.

4.1 Architecture

The following Figure displays a high-level overview of the architecture of the proposed framework:

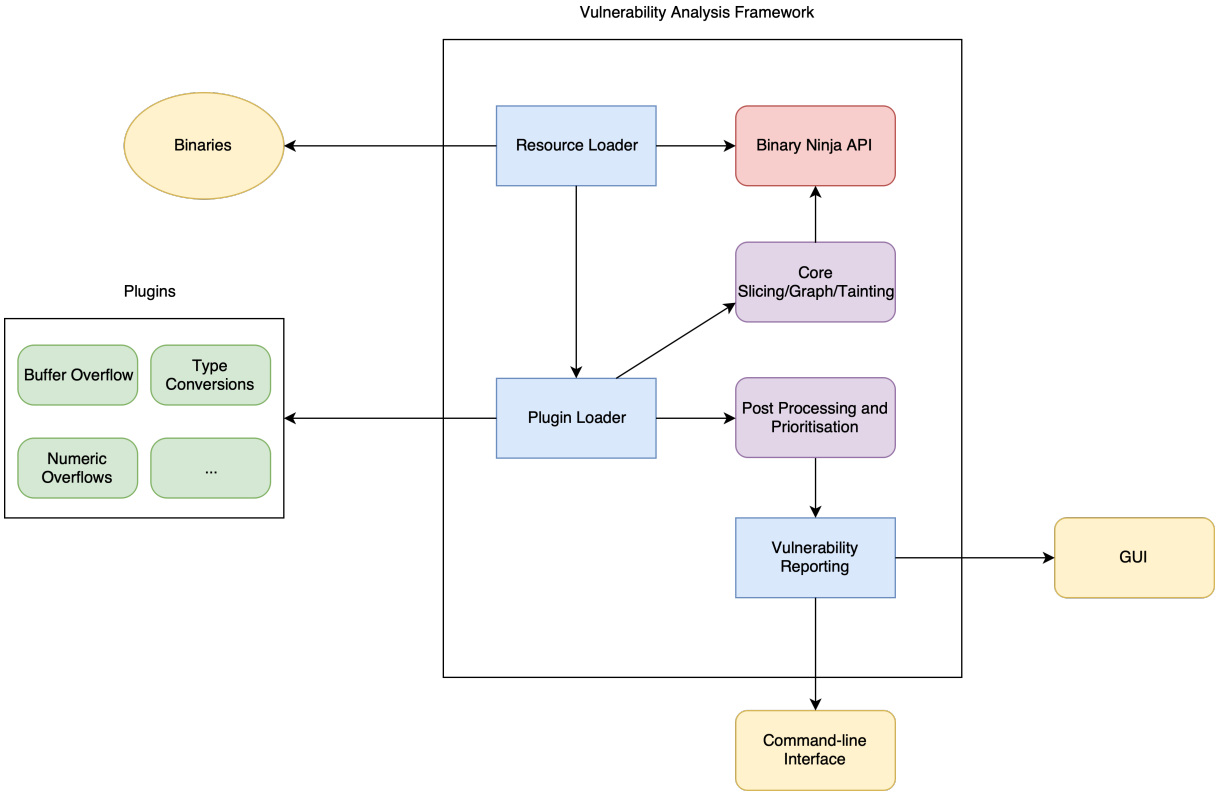

Figure 20: Vulnerability Analysis Framework Architecture

The following describes the different parts of the proposed architecture in the called order:

Resource Loader. The Resource Loader is the very first thing which can be either called over the command-line interface or the GUI. It parses the arguments and loads the corresponding binaries. The disassembling and lifting performs the Binary Ninja API. The returning "Binary View" class is then forwarded to the Plugin Loader.

Plugin Loader. The Plugin Loader looks in the plugin directory for the fitting plugins to

perform the analysis. The loaded plugin receives a reference to the Binary View and the Core (described below). Now they can perform the implemented analysis on the binary.

Core. The Core module includes some additional functionality in addition to the Binary Ninja API such as Slicing/Graph handling and Tainting which was developed during this thesis.

The Core module usually calls the Binary Ninja API directly to perform its tasks.

Post-Processing and Prioritization. Whenever the plugin spots a potential vulnerability, it can report this potential vulnerability to the Vulnerability Reporting module. However, before it lands there, some post-processing is performed depending on settings. For example, it sorts the vulnerability by the given rating and further it can prioritize by given code coverage files (Code Coverage is described in chapter 3.3).

Vulnerability Reporting. The Vulnerability Reporting module handles the reported vulnerabilities from the different Plugins. It can display it to the Binary Ninja GUI or print it to the command-line interface. It is designed to print a detailed output which enables the auditor to perform further analysis.

4.2 Analysis Plugins

The different plugins need to be designed uniformly such that there is no difference when the Plugin Loader starts to load the plugin. Hence, they are designed to be an inheritance of a single plugin class. In this class, different interfaces are designed and contain already defined function prototypes for the Plugin Loader module. It is later required to report the vulnerabilities through a standardized vulnerability class to the Vulnerability Reporting module.

4.3 Walkthrough

This section describes an exemplary vulnerability discovering session with the proposed design, starting with the initial loading over the analysis with the plugins to the final reporting.

1. The Analysis Framework is loading and tests if it is possible to reach the Binary Ninja API.
2. It parses the given arguments.
3. The Resource Loader is called and the path to the binary is extracted from the arguments. Further, the Resource Loader checks whether it is a valid file and then

passes it to the Binary Ninja API for the disassembling and lifting to the Medium-Level Intermediate Language.

4. The Plugin Loader tries to load the plugins from the default plugin directory. It does this one at a time. Blacklisting Plugins for specific tasks is possible. This is needed since sometimes a particular analysis is not necessary depending on the attack surface.
5. The loaded plugin performs its analysis.
6. If the plugin finds a potential vulnerability, it creates a new vulnerability occurrence.
7. The Post Processing and Prioritization module perform its analysis when a plugin finishes. This includes the sorting of potential vulnerabilities with several factors.
8. The Vulnerability Reporting module receives the potential vulnerabilities and emits them to either the attached GUI or to the command-line.
9. The next plugin is executed.

An essential advantage of this design is that the plugin emits the possible vulnerabilities directly at step 8. Hence, a manual analysis can start directly while the other plugins still performing the analysis.

4.4 Limitations

Several limitations need to be kept in mind when comparing the presented approach to different designs and when evaluating the discovered potential vulnerabilities. These limitations and their impact on the analysis results are discussed in the following.

False Positives. The framework can only be as good as the written plugins. Hence, it is possible to generate a lot of false positives. These occurrences need to be manually worked through. However, since this framework is designed for vulnerability researchers, they are usually willing to work through false positives to find one correct vulnerability. This process also helps the auditor to get to know the software better.

Massive Output of Vulnerabilities. When more and more plugins are added it will create a massive output on potential vulnerabilities. Thus, it is necessary to load only the modules that match the potential attack surface. Further, when supplying a big set of code coverage files the framework can more easily display the relevant parts to the user.

Memory Exhaustion. Using plugins with a very high memory usage might exhaust the memory on a small system and results in a shutdown of further analysis. This can

occur very fast when combining the static analysis with a concolic execution engine as described in the concolic execution chapter 2.7.

4.5 Conclusion

The design of our proposed framework allows modeling every possible vulnerability via the plugin mechanism. Thus, when the framework is accepted by the community and plugins are under active development it can combine multiple spread single scripts into a single framework to optimize the vulnerability discovery approach. This framework idea can be viewed similarly to the Metasploit framework[12] which was accepted by the community and is now the largest and most used penetration testing framework.

5 Implementation

Based on the design in chapter 4, the following chapter presents the implementation details which was implemented for this thesis. Chapter 5.1 introduces the exposed API from Binary Ninja and chapter 5.2 describes the implementation of the previously introduced components of the framework. Furthermore, the plugin system is described in detail in chapter 5.3 while finally the chapter 5.4 gives a preview of the further development.

5.1 Binary Ninja Interface

Binary Ninja exposes a very rich API. This is owed to the design of Binary Ninja where the whole Graphical Interface is based on the exposed interface. When comparing this API with IDA Pro the difference is noticeable since IDA Pro was designed at first without an exposed API and thus the API is still quite limited.

Introducing the Binary Ninja API again the previous defined is_sorted function is used which was disassembled in the figure 10.

The following parts display exemplary the use of the Binary Ninja API in an interactive Python shell. The first commands extract the fourth instruction in the lifted Medium-Level Intermediate Language:

```
1 >>> current_function.medium_level_il[3]
2 <il: rsi = zx.q(arg2.esi - 1)>
```

Listing 19: API Example I

The following code extracts the variable to which the instruction writes to:

```
1 >>> current_function.medium_level_il[3].dest
2 <var uint64_t rsi>
```

Listing 20: API Example II

The following code returns the variables, where the current instruction reads from, in a list:

```
1 >>> current_function.medium_level_il[3].vars_read
2 [<var int32_t arg2>]
```

Listing 21: API Example III

The Binary Ninja API has a tremendous functionality to work on the disassembled binary. For further reference, it is recommended to read the official API documentation [18].

5.2 Components

This chapter describes the different implementation details of the components which were introduced in chapter 2.1.

5.2.1 Resource Loader

The Resource Loader module was developed for the thesis which handles either command-line arguments or parameters from the GUI. Further, it does perform a necessary check whether the given file does exist or not. The following code is responsible for this code (Comments are stripped out for visibility):

```
1  def is_valid_file(parser, arg):
2    if not os.path.exists(arg):
3        parser.error("The file %s does not exist!" % arg)
4    else:
5        return arg
6
7  [...]
8
9  def main():
10     parser = argparse.ArgumentParser(description='VAF command-line tool:
   Searches Automagically for Bugs')
11     parser.add_argument('--deep',
12                         dest='deep', action='store_true',
13                         help='Uses Deep Search mode. This might take
   longer but it will also get a grasp of compiler optimizations')
14     [...]
15     parser.add_argument('target', metavar='target-path', nargs='+',
16                         help='Binary to be analysed',
17                         type=lambda x: is_valid_file(parser, x))
18
19     [...]
20     parser.add_argument("--cov", default=None,
21                         dest="coverage", help="Provide a coverage file
   for better filtering")
22
23     parser.add_argument("--cov_folder", default=None,
24                         dest="cov_folder", help="Provide a folder with
   coverage files for better Prioritization")
25
26     [...]
27
28     input_file = args.target
29     for filename in input_file:
```

```
30            print("Analyzing {0}".format(filename))
31            bv = binaryninja.BinaryViewType.get_view_of_file(filename)
32            print("arch: {0} | platform: {1}".format(bv.arch, bv.platform))
33            bv.update_analysis_and_wait()
34            [...]
35
36    if __name__ == "__main__":
37        main()
```

Listing 22: Resource Loader Module

When starting to describe the above-presented source code, It starts with the function "is_valid_file" and is responsible for checking whether the given file is present on the current file system. Further, the "main" function is displayed which is called when the framework is starting. The "main" function adds an argument parser which takes parameters for the deep analysis ("–deep" described in chapter 5.3.2.2), the coverage files either as a single file or in a folder with the parameters "–cov" or "–cov_folder" and finally the target binary to analyze.

5.2.2 Plugin Loader

The Plugin Loader module is adjusted from the "deen" project [51] to perform the loading of the externally written plugins. It is designed to be simple and without much overhead. The Plugin Loader searches in the plugin directory for available files. If they are in the correct format, the loader is willing to load the plugin into the framework. The following code excerpts demonstrates how the Plugin Loader module works:

```
1   [...]
2   def main():
3       [...]
4       plugins = PluginLoader(argparser=parser)
5       [...]
6       for name, _ in plugins.available_plugins:
7           plugin = plugins.get_plugin_instance(name)
8           plugin.vulns = []
9           plugin.run(bv, args.deep)
10          [...]
11          del plugin
```

Listing 23: Plugin Loader Module I

The code presented above calls the PluginLoader class which handles loading the plugin. Further, the loop iterates over every available plugin and gets the instance for it. Next,

the loop clears previously found vulnerabilities and then calls the entry function (run) for every available plugin with the required arguments.

5.2.3 Core

The core functionality is written to support plugin developers in performing tasks such as slicing or loop detection. Since this is not part of the Binary Ninja framework, it was required to develop this part of the software first to handle different cases of the vulnerability analysis. The following sections introduce these core parts.

SSA Slicing. Single Static Assignment slicing is a technique used for tainting and following variables through the control flow. This is needed when the underlying analysis is based on sources and sinks. Hence, when identifying a possible sink or source, it is needed to either backtrack or follow this variable forward through the control flow. Further, it is also required to track the call flow forwards and backward since variables can be passed via parameters to a function.

This slicing technique can be achieved using the Binary Ninja API when using SSA variables. Since by design a Single Static Assignment can only be assigned once but used multiple times there exists a 1 to N relationship which the Binary Ninja API exposes through the SSA variable. The following excerpt displays the function prototype:

```
1 def get_ssa_var_uses(self, ssa_var):
2   [...]
3   return result
```

Listing 24: get_ssa_var_uses[20]

As the name suggests, this interface will return every use of the SSA variable to the caller. Using the shown function enables to slice through the program when visiting every point where the function uses this variable. When trying to slice backward, the following exposed function can be used:

```
1 def get_ssa_var_definition(self, ssa_var):
2   [...]
3   return result
```

Listing 25: get_ssa_var_definition[19]

When performing this recursively or iteratively, it is possible to traverse through the program and find the control flow from sources to sinks and reverse.

The following describes the implementation of the forward slice functionality including tracing on variables:

```
def do_forward_slice_with_variable(instruction, function):
  # if no variables written, return the empty set.
  if not instruction.ssa_form.vars_written:
      return set()
```

Listing 26: Forward Slicing I

The very first part of the function describes whether the given instruction does perform writes on a variable. This SSA variable is then later used as the initial point for slicing:

```
instruction_queue = {}

for var in instruction.ssa_form.vars_written:
  if var.var.name:
    for use in function.ssa_form.get_ssa_var_uses(var):
      instruction_queue.update({use: var})

visited_instructions = [(instruction.ssa_form.instr_index, None)]
```

Listing 27: Forward Slicing II

The second part of the function defines at first an instruction queue. This container is used to track which instructions still need to be visited for a full analysis. Further, the following loop is then responsible for filling the queue with the very first SSA variable uses.

The following part will iterate through the instruction queue and saves every visited function and the corresponding SSA variable in a list which is later returned to the caller:

```
while instruction_queue:

  visit_index = instruction_queue.popitem()

  if visit_index is None or visit_index[0] in visited_instructions:
    continue

  instruction_to_visit = function[visit_index[0]]

  if instruction_to_visit is None:
    continue
```

```
12
13      for var in instruction_to_visit.ssa_form.vars_written:
14        if var.var.name:
15          for use in function.ssa_form.get_ssa_var_uses(var):
16            instruction_queue.update({use: var})
17
18      visited_instructions.append(visit_index)
19
20    return visited_instructions
```

Listing 28: Forward Slicing III

5.2.4 Post Processing and Prioritization

The following describes how the post-processing is performed and how precisely the prioritization is implemented.

Post Processing. The Post Processing is designed to help in identifying the severity of vulnerabilities by coloring the output. The module considers the following cases depending on the severity of the vulnerability:

Severity	Color
80-100	Red
60-79	Magenta
40-59	Yellow
20-39	Blue
0-19	White

Prioritization. The prioritization is based first on the supplied code coverage files to highlight potential vulnerabilites that were already visited by supplied input. After the sorting from the code coverage file is done it sorts based on severity.

5.2.5 Vulnerability Reporting

The Vulnerability Reporting module prints out the discovered vulnerabilities in an easy to grasp way. The output can either be displayed on the command-line or within the Binary Ninja GUI.

The following output displays the colored text sorted by potential vulnerabilities:

```
Vulnerability:
goodG2B 0x112b  memcpy(var_78, var_e8, 99);
                Potential Overflow!
                        dst var_78 = 8
                        n <const 0x63> = 99

Description:
The amount of Copied bytes is bigger than the destination Buffer
---
Vulnerability:
MLIL CWE195_Signed_to_Unsigned_Conversion_Error__connect_socket_memcpy_01_bad 0x108f
                Potential bad sign conversion
                        Variable rdx_2 is signed but will be implicitly converted by memcpy to size_t

Description:
It appears that signed variable is converted by an implicit conversion with a function call.
---
Vulnerability:
MLIL goodG2B 0x112b
                Potential bad sign conversion
                        Variable rcx_1 is signed but will be implicitly converted by memcpy to size_t

Description:
It appears that signed variable is converted by an implicit conversion with a function call.
---
```

Figure 21: Vulnerability Reporting

The above output is the result of the framework running multiple plugins. The very first vulnerability was discovered with the Buffer Overflow Plugin described in chapter 5.3.2.2. The following two vulnerabilities were discovered by the Type Conversion plugin which is described in chapter 5.3.2.3.

Every single plugin can configure the format, in which the output is displayed to the user. However, the framework performs the coloring and sorting.

5.3 Plugin System

Since the framework is designed to make it very easy for the community to develop their analysis based on the given Framework a plugin system was required. The overall structure for the plugin system which is used by the framework originates from the "deen"[51] project and is slightly modified to fit the framework.

The following skeleton class shows the overall structure of how a plugin needs to be built. Every plugin is required to inherit this parent class. Thus, it is briefly explained in the following:

```
1 class Plugin(object):
2   error = None
3
4   name = ''
5
6   display_name = ''
```

```
7
8   aliases = []
9
10  vulns = []
11
12  _traces = []
13
14  _binaryView = None
```

Listing 29: Plugin Class I

The above first part defines variables which are required for every plugin to work. The error variable indicates whether an error occurred while loading or while performing the analysis. Further, the name/display_name and aliases variable can be used to name a plugin. The list "vulns" is used to save every discovered vulnerability during the lifetime of a plugin. The "vulns" list is later passed to the post-processing module. After the plugin is done with its processing, the plugin passes the "_traces" variable to the Prioritization Module. Finally, the "_binaryView" variable represents a current state from the analyzed Binary Ninja state. Plugins can use this variable internally to perform further analysis if needed.

The following displays some core functions of the plugin skeleton (getter/setter/comments and some other functions are stripped for visibility):

```
1   def __init__(self, bv):
2       self._binaryView = bv
3
4   def __del__(self):
5       for vuln in self.vulns:
6           if len(self._traces) > 0:
7               bb = get_basic_block_from_instr(self.bv, vuln.instr.address)
8               vuln.cmd_print_finding(self._traces, bb)
9           else:
10              vuln.cmd_print_finding()
11
12  [...]
13
14  def append_vuln(self, v):
15      if not [e_vuln for e_vuln in self.vulns if not e_vuln != v]:
16          self.vulns.append(v)
17
18  def run(self, bv):
19      raise NotImplementedError
```

```
20    [...]
```

Listing 30: Plugin Class II

The constructor "__init__" is called implicitly by the Plugin Loader Module to initialize the "_binaryView" variable. Further, to pass the discovered vulnerabilities to the Post Processing and Prioritization module the deconstructor "__del__" is overwritten. Another notable aspect is the code coverage part in the deconstructor where it performs the analysis of the given code coverage file and compares it to the discovered vulnerabilities.

The append_vuln function should be used by the plugin itself to append a discovered vulnerability. The function does perform very basic duplicate checking to prevent vulnerabilities to be added multiple times.

Finally, the run function is called by the plugin loader after it successfully loaded the plugin. Thus, this is where the analysis algorithm needs to be either implemented or called from. The skeleton forces the plugins to overwrite this function using the raise of a "NotImplementedError" error.

Above, it was shown how a plugin can be implemented within the framework. The following excerpt displays an exemplary plugin. This might be used for further author´s to start their work with the framework:

```
1   from . import Plugin
2   from src.avd.core.sliceEngine.loopDetection import loop_analysis
3
4   __all__ = ['ExamplePlugin']
5
6
7   class PluginExample(Plugin):
8       name = "ExamplePlugin"
9       display_name = "Example Plugin"
10      cmd_name = "ePlugin"
11      cmd_help = "Displays the basic usage of a Plugin"
12
13      def __init__(self, bv=None):
14        super(PluginExample, self).__init__(bv)
15        self.bv = bv
16
17      def set_bv(self, bv):
18        self.bv = bv
19
20      def run(self, bv=None, deep=None, traces=None):
```

```
21          super(PluginExample, self).__init__(bv)
22          self.perform_analysis()
23          return
24
25      def perform_analysis(self):
26          for func in self.bv.functions:
27              func_mlil = func.medium_level_il
28              for bb in func_mlil:
29                  if loop_analysis(bb):
30                      for instr in bb:
31                          print(instr)
```

Listing 31: Example Plugin

5.3.1 Pre-Processing Plugins

Due to the design of the framework, it is possible to update the current Binary View. Thus, this enables the use of pre-processing plugins. These plugins can be designed to resolve function names from debugging functions and update the Binary View for other plugins. This makes the framework flexible, and certain plugins can prepare the environment before the vulnerability plugins are running. Another compelling case is malware these programs are often heavily obfuscated, packed, and encrypted. Thus, the pre-processing plugins can take over at first and deobfuscate /unpack or decrypt the binary for the other plugins if possible. This process can, of course, be chained. However, since these deobfuscation algorithms are again not decidable, it is required of some manual work by the user.

5.3.2 Vulnerability Plugins

In the previous section 5.3 the overall Plugin structure was introduced. The following section describes the already developed plugins to detect a different kind of vulnerabilities. Chapter 5.3.2.1 describes the approach to find large stack frames inside functions while chapter 5.3.2.2 models the large buffer overflow plugin developed for this thesis. Finally, chapter 5.3.2.3 displays an algorithm to discover implicit type confusions and chapter 5.3.2.4 describes the DFS algorithm to find uninitialized data problems ending with chapter 5.3.2.5 where the plugins is from an external source and migrated to the framework.

5.3.2.1 Large Stack Frames The most simple developed plugin is scanning the code for large stack frames. Even though this does not indicate a vulnerability directly, problems usually appear in the vicinity of such large buffers. Hence, this plugin is designed to

be more of an informational plugin. Furthermore, the following flow graph displays the analysis process of the Large Stack Frame plugin:

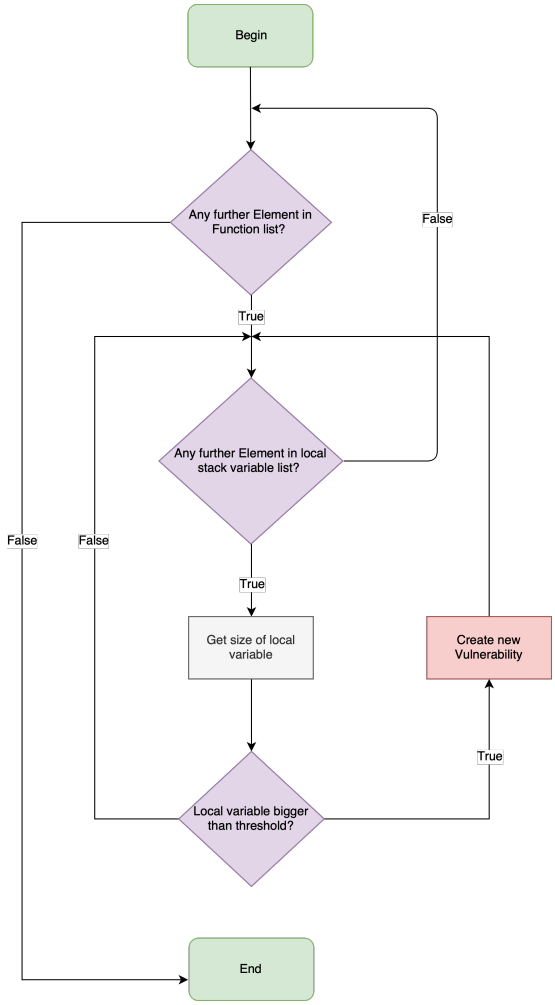

Figure 22: Large Stack Frame Data Flow Graph

The above displayed flow graph is implemented by the following source code excerpt:

```
1  _threshold = 150
2
3  def run(self, bv=None, deep=None, traces=None):
4    super(PluginLargeStackFrame, self).__init__(bv)
5    self._find_large_stack_frames()
6    return
7
8  @staticmethod
9  def _calc_size(var, func):
10   if SSAVariable == type(var):
11       var = var.var
12   if len(func.stack_layout) - 1 == func.stack_layout.index(var):
```

```
13          return abs(var.storage)
14      else:
15          return abs(var.storage) - abs(func.stack_layout[func.stack_layout.
        index(var) + 1].storage)
16
17  def _find_large_stack_frames(self):
18      for func in self.bv.functions:
19          for var in func.stack_layout:
20              size = self._calc_size(var, func)
21              if size >= self._threshold:
22                  [...]
23                  self.vulns.append(vuln)
```

Listing 32: Large Stack Frame Plugin

As described in section 5.3 the Plugin Loader is calling at first the run function inside the Plugin. The run function initializes the _binaryView variable with a possible updated analysis and then calls the internal analysis function "_find_large_stack_frames". This function iterates over every discovered function from Binary Ninja and then iterates over every single local stack variable. The size of the variable is calculated with the internal "_calc_size" function. If this size is bigger than the defined threshold variable "_threshold", a vulnerability notification is generated.

5.3.2.2 Buffer Overflows

This chapter introduces the Buffer Overflow plugin which was developed for this thesis. It is designed to catch various potential overflow scenarios.

Source > Destination. One indicator of a potential overflow can by default be that the source buffer is bigger than the destination. To check for such an occurrence, the plugin scans the binary for potential sinks.

The following flow graph displays the basic idea and the algorithm behind it:

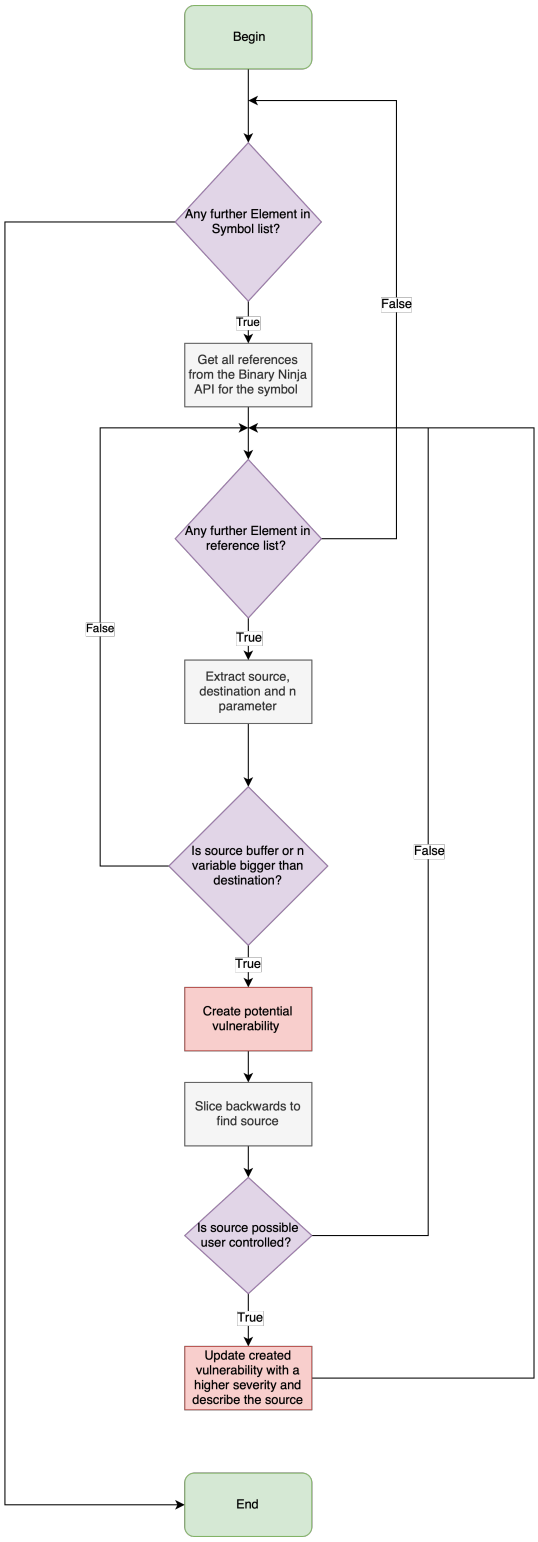

Figure 23: Buffer Overflow Flow Graph

If the plugin finds references of known sinks, it will start to analyze the parameters of it. To determine whether the source buffer or the size parameter is bigger than the destination buffer, the plugin performs a lot of different operations. For example when

dealing with format string functions such as "snprintf" where the function parses a format string. Thus, the plugin requires to do the very same to calculate the overall maximum length of the source buffers including every character in the format string.

The following displays a list for some of the current implemented problematic functions with the corresponding interesting parameters (This list is not complete for all problematic functions but can be extended if required.):

Function	Destination	Source	n
memmove	0	1	2
memcpy	0	1	2
strncpy	0	1	2
strcpy	0	1	
strcat	0	1	
strncat	0	1	2
sprintf	0	2	
snprintf	0	3	1
vsprintf	0	2	
fgets	0		1
gets	0		
scanf	unlimited		

Since the functions have different parameters, it is needed for the plugin to understand at which exact position the source buffer, the destination buffer, and the possible length is placed. Thus, the plugin handles the different cases.

Compiler Optimization and Loops. The previously described approach can handle most of the cases. However, two particular cases cannot be handled with this approach.

Case one is a self-implemented memcpy like function where the copy operation is performed by a loop and delimited for example by a semicolon or similar characters.

The second case is related to compiler optimizations. If the compiler can calculate the final size of the source buffer accurately and the size is below a certain threshold it is possible that the compiler is not resolving, for example, it to the programmed "memcpy" call and is rather implementing a memcpy like a loop. This is faster since it is not required to have a function call overhead and call the corresponding "libc" function.

The following source code displays the example of the above-described compiler optimization:

```
1  #include <stdio.h>
2
3  int vuln(){
4      char form[2048];
5      char bad[1024];
6      fgets(form, 2047, stdin);
7      memcpy(bad, form, sizeof(form));
8      return 1;
9  }
10
11 int main(int argc, char const *argv[])
12 {
13     char buf[21];
14     for(;;)
15     {
16         fgets(buf, 20, stdin);
17         if(strcmp("vuln\n", buf) == 0){
18             vuln();
19             continue;
20         }
21     }
22     return 0;
23 }
```

Listing 33: Compiler Optimization

The function vuln contains a straight buffer overflow where fgets (line 6) reads up to 2047 bytes from the standard input and the typical pattern of using the source buffer size for the memcpy call.

Compiling the above-displayed source code with optimizations enabled (default). The compiler can emit an optimized out memcpy. The following graphic from Binary Ninja displays the disassembled binary:

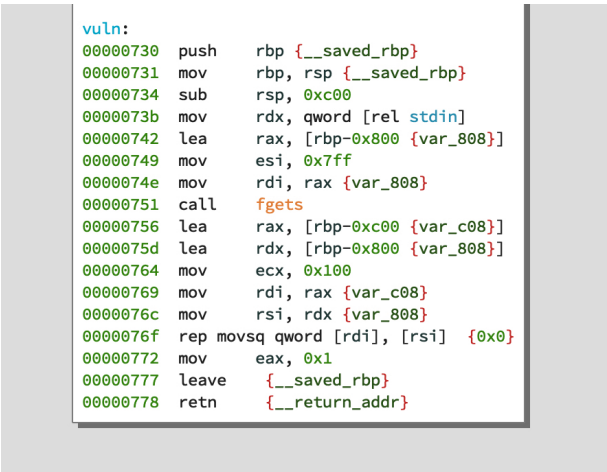

Figure 24: Optimized memcpy

Analyzing the graph in figure 25 it is possible to the see the fgets function call but according to the source-code directly after the memcpy would follow. At this point, the compiler could calculate at compile-time that a loop (the instruction "rep movsq qword [dst] [src]" copies until the counter register is at zero) would be faster than the call to the memcpy. Since this function, of course, is prone to a buffer overflow it is necessary to find such loops with this pattern of coping byte per byte and calculate the overall loop length.

Graph Parsing. When lifting the above-displayed assembly to the Medium-Level Intermediate language using the Binary Ninja API. The result can be displayed in a graph. Thus, it is possible to use some graph theory to determine such byte copying patterns. The following graph displays the lifting from assembly to the Medium-Level Intermediate language:

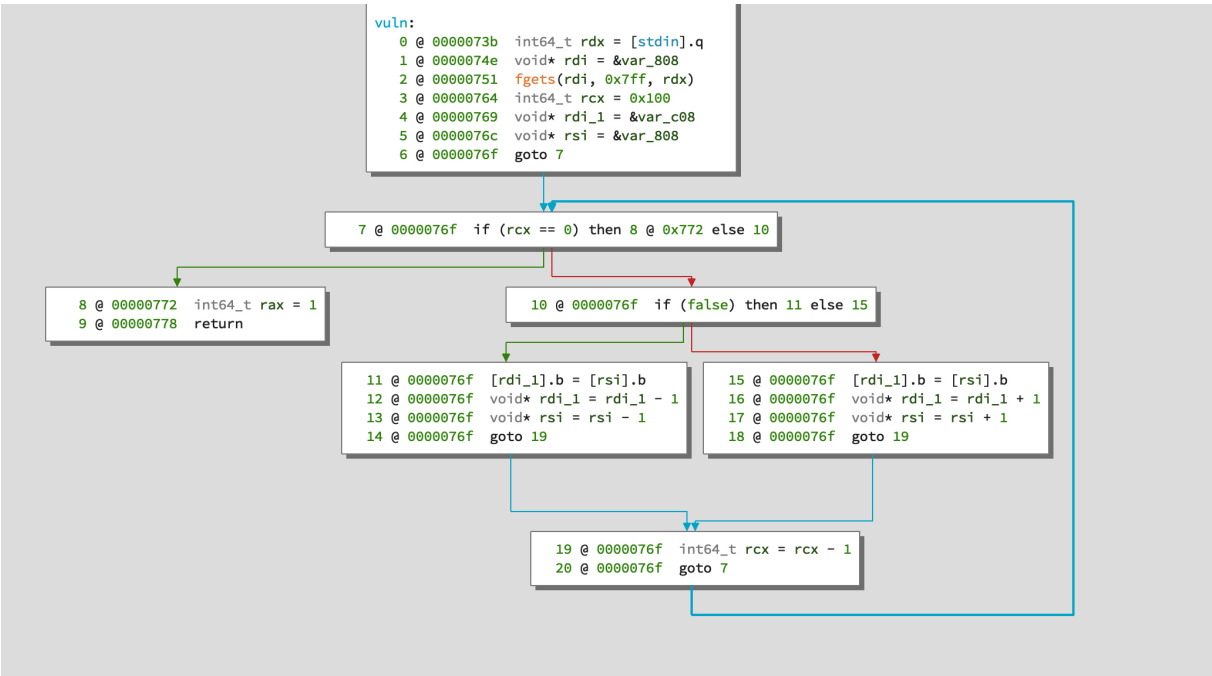

Figure 25: Optimized memcpy

To search for similar patterns in general the following algorithm called "deep search" was developed for the framework:

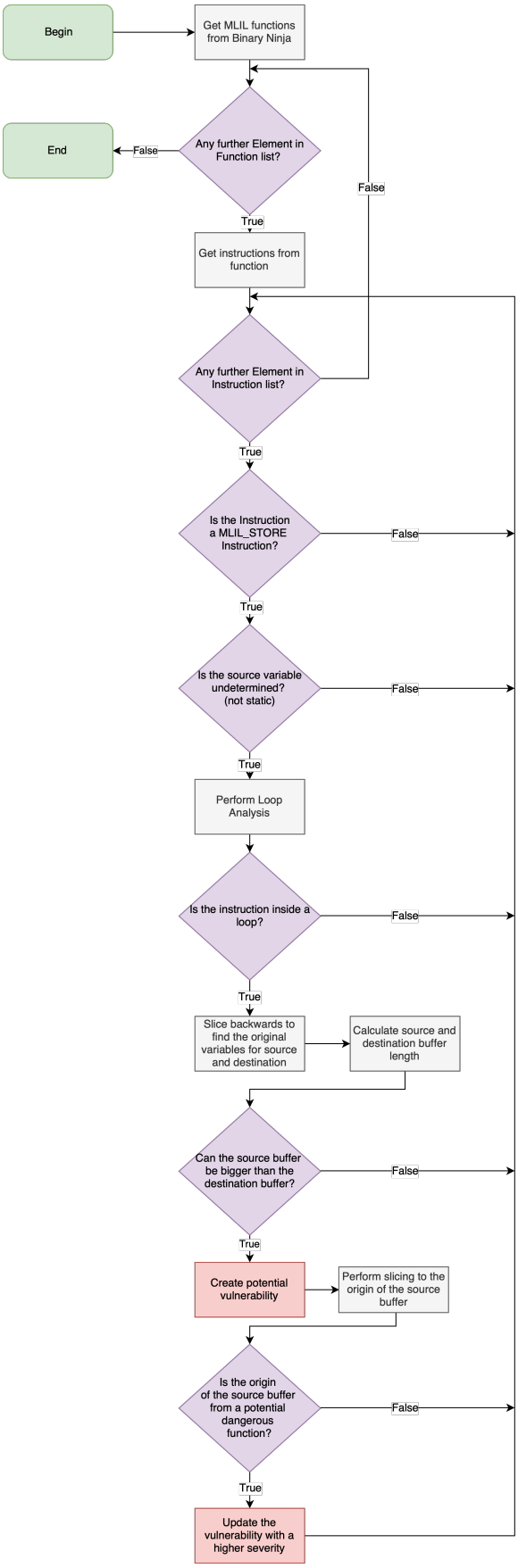

Figure 26: Deep Search Algorithm

The displayed flow graph in figure 26 is self-explaining. However, a few algorithms are abbreviated such as the loop analysis algorithm. Hence, the following part explains this in more detail.

Detecting whether the copy action is performed in a loop is a crucial part of finding such patterns.

First, the function to detect a loop has to create a graph from the function. Thus, every basic block has to represent a vertex, and the outgoing edges from the basic blocks are also added to the graph.

Second, to detect loops on the created graph, we perform a depth-first search (described in chapter 2.6) and walk on the successors of the vertexes. If we hit a vertex which is already in the walked path, it indicates a loop. To erase duplicates and merging loops it is necessary to find the common dominator for all the inner loops.

Finally, the result of the algorithm are unique loops inside the analyzed function. Thus, it is possible to determine whether this instruction was placed inside a loop.

It needs to be noted that this approach can take a long time on large programs and lead to quite some false positives. Thus, the deep search algorithm is only enabled when passing the "–deep" flag to the framework.

5.3.2.3 Type Conversion

This chapter introduces the Type Conversion plugin. This plugin is designed to identify possible implicit type confusions which were introduced in chapter 3.4.3. While Binary Ninja only served the Low-Level Intermediate Language and did not inherit such a signed analysis an independent signed analysis script was developed by Sophia D'Antoine [25]. However, since the introduction of the Medium-Level Intermediate Language, Binary Ninja offers their very own signed analysis, and the following plugin was developed to make use of this feature.

To verify the successful detection of such a problematic type conversion the following source code was selected to test the plugin on.

```
1  void Signed_to_Unsigned_Conversion_Error()
2  {
3      int data;
4      int recvResult;
5      [...] // setting up the Socket
6      recvResult = recv(connectSocket, inputBuffer, CHAR_ARRAY_SIZE - 1, 0)
       ;
7      if (recvResult == SOCKET_ERROR || recvResult == 0)
```

```
8         {
9             break;
10        }
11        inputBuffer[recvResult] = '\0';
12        // Convert to int
13        data = atoi(inputBuffer);
14        [...]
15        char source[100];
16        char dest[100] = "";
17        memset(source, 'A', 100-1);
18        source[100-1] = '\0';
19        if (data < 100)
20        {
21            // POTENTIAL FLAW
22            memcpy(dest, source, data);
23            dest[data] = '\0';
24        }
25    }
```

Listing 34: Type Conversion Problem

As described in the Type Confusion chapter 3.4.3, when the "data" variable land in the "memcpy" function sink it is converted implicitly to an unsigned integer since the function expects input of type "size_t". Binary Ninja disassembles and lifts this basic block like the following figure displays:

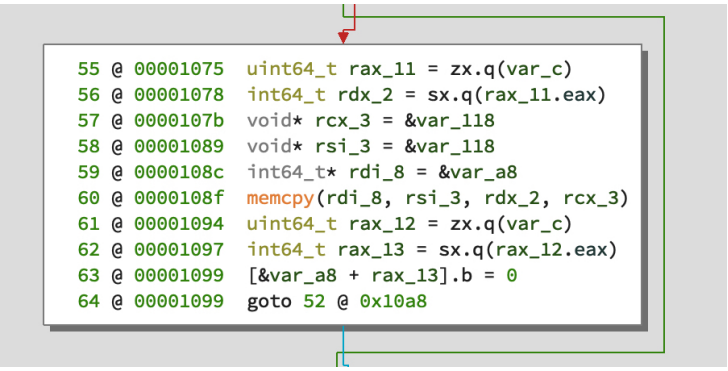

Figure 27: Binary Ninja Type Conversion

The figure 27 displays that Binary Ninja already is trying to assign the correct type to the variables. For example, at index 56 the variable "rdx_2" is assigned the "int64_t" type. This variable is later used in the "memcpy" function for the length parameter and implicitly converted to an unsigned integer. The following function in the plugin can analyze the binary for precisely this pattern.

```
unsinged_sinks = {"malloc":0, "memcpy":2}

def _function_sign_analysis_start(self, func):
  for blocks in func.medium_level_il:
    for instr in blocks:
      if instr.operation == MediumLevelILOperation.MLIL_CALL:
        try:
          call_name = self.bv.get_function_at(instr.dest.constant).name
        except AttributeError:
          # Bypass GOT references...
          continue

        if call_name in self.unsinged_sinks.keys():
          try:
            if instr.vars_read[self.unsinged_sinks.get(call_name)].type.
    signed:
              [...] # Creating a new discovered vulnerability
```

Listing 35: Type Conversion Problem

The variable "unsigned_sinks" represents the currently implemented sinks. This can, however, be very fast extended with just adding more sinks and specifying on which parameter the implicit conversions happen. Further, the function iterates over every basic block and instruction from the passed function parameter. While iterating the function checks whether the operation performed is a function call (MLIL_CALL) and checks if the function name has an occurrence in the specified sinks variable. If this is the case, it checks the implicit conversion parameter if it is a signed variable.

It should be noted that this approach does not discover every single type conversion error, but further development can increase the bug coverage of this plugin.

5.3.2.4 Uninitialized Data This section introduces the plugin to discover uninitialized data. The problems with uninitialized variables and the exploitability were introduced in chapter 3.4.4. The current approach scans for uninitialized local stack variables and thus must be extended in the future to cover such uninitialized variables which are placed on the heap.

The following describes the approach by the plugin. The plugin iterates over every discovered function from Binary Ninja and creates a graph of the function similar to the deep analysis in chapter 5.3.2.2, but it returns a list of every possible basic block path using the depth-first algorithm. The following source code displays how the used depth-first

algorithm (described in chapter 2.6) is implemented inside the plugin to walk all possible paths in a recursive approach:

```
def _find_all_paths(self, graph, start_vertex, end_vertex, path=[]):
  path = path + [start_vertex]
  if start_vertex == end_vertex:
    return [path]
  if not graph.vertices.has_key(start_vertex):
    return []
  paths = []
  for node in graph.get_vertex_from_index(start_vertex).
   get_successor_indices():
    if node not in path:
      newpaths = self._find_all_paths(graph, node, end_vertex, path)
      for newpath in newpaths:
        paths.append(newpath)
  return paths
```

Listing 36: Depth-First Algorithm

While iterating over the functions, the plugin extracts the local stack variables from the current functions; the Binary Ninja API exposes a prototype to check where these variables are used inside the analyzed function. The plugin loops over the over every instruction where the variable is used and slices backward on every basic block from the discovered unique paths. If the plugin discovers that a single path uses a variable in a function call where the variable was not written to, the plugin creates a new potential vulnerability.

Improvement. This approach does contain some limitations. The first limitations is that due to the graph DFS search the control flow is not correctly taken into account. For example if a variable is initialized inside a loop which branches from the initial declaration the initialization process is inside another branch of the graph. Thus, the algorithm can find a path where the variable is not initialized properly and this results in a false positive. The following source code illustrates exactly this behavior:

```
  static void goodG2B()
  {
      int h,j;
      char * data;
      for(h = 0; h < 1; h++)
      {
          data = "string";
```

```
8           }
9           for(j = 0; j < 1; j++)
10          {
11              printLine(data);
12          }
13      }
```

Listing 37: DFS Limitation

The above source code excerpt displays such a false positive. It is quite evident that the "data" variable is initialized in the loop directly above. However, since we analyze this function on a lifted intermediate language the following figure displays this function in the lifted state:

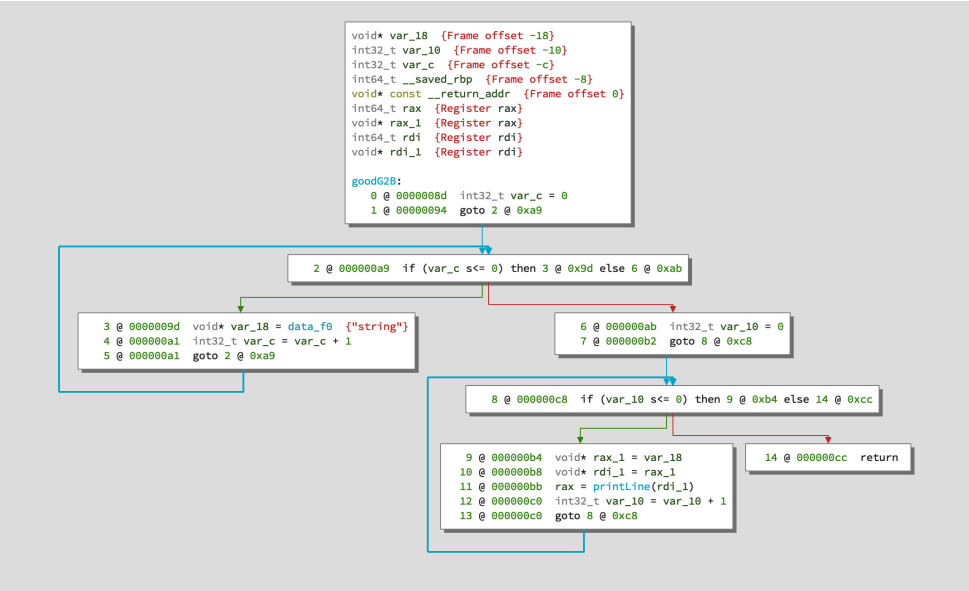

Figure 28: DFS Limitation Lifted IL

When the above shown Python source code displayed in listing 36 runs over this graph with the DFS search, it returns a path where the variable used in "printline" is not initialized. Hence, a false positive since when analyzed correctly it is inevitable for the control flow to bypass the initialization loop of the data variable.

To reduce the occurrence of such false positives a future improvement of the plugin is to eliminate paths which cannot be unique due to the control flow by using concolic execution which was described in chapter 2.7. This enables the plugin to check whether it is possible to walk a path without branching into the loop.

Another limitation is that some structures and variables can be initialized from other threads. This scenario is unlikely to be solved with this algorithm and is instead tackled

in the upcoming race-condition plugin, where such races are analyzed.

5.3.2.5 Heartbleed Plugin To illustrate the fast development and adoptions of available public scripts to discover security vulnerabilities the Heartbleed discovery algorithm [22] from Josh Watson was used to demonstrate the effectiveness of the framework.

The following quote describes the famous Heartbleed bug [42]:

> The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet.[42]

The vulnerability detection script described earlier tries to find precisely this occurrence where untrusted input is used as the size parameter for the "memcpy" function. The script makes use of Z3 [50] to determine whether it is untrusted or not.

The following figure displays the vulnerable "memcpy" call at index 35 in the function "tls1_process_heartbeat":

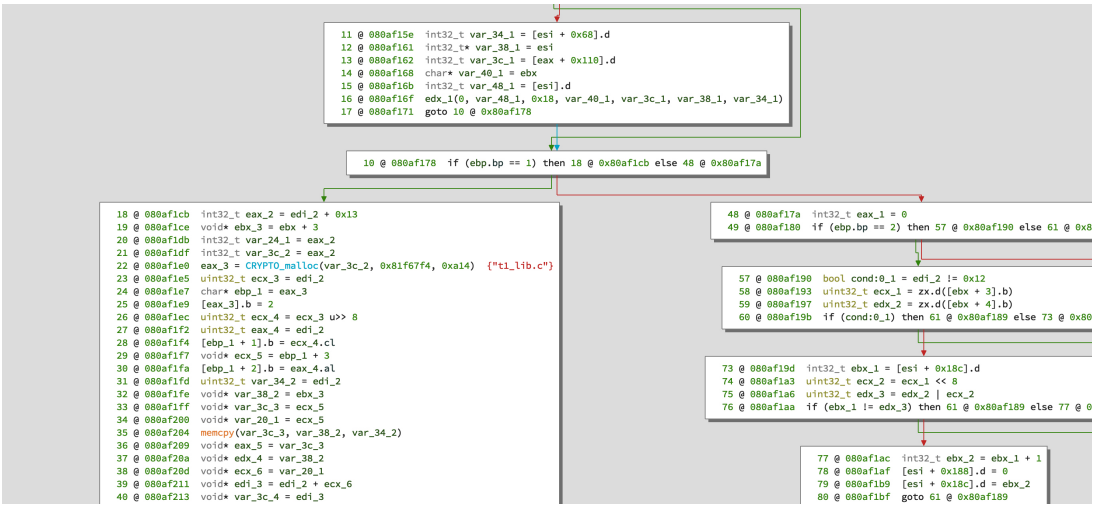

Figure 29: Heartbleed Vulnerability

After running the plugin to discover the vulnerability, the following output is presented to the auditor:

```
---
Vulnerability:
tls1_process_heartbeat 0x80af204
                the memcpy function uses a size parameter that potentially comes from an untrusted source
Description:
Potential Untrusted Source.
---
Vulnerability:
ssl3_get_client_key_exchange 0x80998ab
                the memcpy function uses a size parameter that potentially comes from an untrusted source
Description:
Potential Untrusted Source.
---
Vulnerability:
ssl3_get_key_exchange 0x809d5ac
                the memcpy function uses a size parameter that potentially comes from an untrusted source
Description:
Potential Untrusted Source.
---
```

Figure 30: Heartbleed Plugin Output

When inspecting the address of the output it matches directly to the Heartbleed vulnerability in the "tls1_process_heartbeat" function, and thus it displays that with the right plugin such severe vulnerabilities can be discovered automatically.

5.3.3 Testing

It is neccessary to test whether the plugins for the framework are discovering vulnerabilities or are improving when updates are submitted to the release builds. Thus, several open source test frameworks for vulnerability discovery are part of the proposed automatic vulnerability discovery framework.

The following frameworks are part of the testing environment.

Juliet. The National Institute of Standards and Technology (NIST) released a testing framework for static analyzers. The following quote is directly from their website and describes the framework.

> The Juliet test suite is a systematic set of thousands of small test programs in C/C++ and Java exhibiting over 100 classes of errors, such as buffer overflow, OS injection, hardcoded password, absolute path traversal, NULL pointer dereference, uncaught exception, deadlock, and missing release of resource. These test programs should be helpful in determining capabilities of software assurance tools, particularly static analyzers.[24]

This framework is the most developed one containing the very most test cases.

DARPA Challenge Binaries on Linux, OS X, and Windows. The Cyber Grand Challenge which was held by DARPA [26] in 2016 released a lot of different programs which modeled real-world vulnerabilities in their operating system called DECREE [37].

However, since mapping DECREE binaries are not precisely what is needed in the real world, the Trail of Bits[21] team started to transform the vulnerabilities from the DECREE platform to the UNIX operating system.

The following quote describes why these examples are necessary for testing.

> These programs were specifically designed with vulnerabilities that represent a wide variety of software flaws. They are more than simple test cases, they approximate real software with enough complexity to stress both manual and automated vulnerability discovery.[23]

Thus, these examples are perfect for testing. However, some of the vulnerabilities are very hard to model statically and thus needs concolic or dynamic execution.

Custom. The custom framework is designed to first model test cases that were made up and second real-world vulnerabilities that were found in assessments. The number of custom test cases should increase over time.

5.3.4 Unit Tests

Since the proposed framework needs to be tested frequently if a change on the core system does break functionality on vulnerability discovery from some plugins. The framework contains of a unit testing part. Every single previous discovered vulnerability is placed into these unit tests to ensure the continous quality.

5.4 Road Map

This section describes the upcoming features that are still to be developed for the framework. The bulletpoints are sorted by priority.

angr. It is foreseen to implement anger more deeply into the core system to enable a better use of concolic execution for the plugins.

Extending the Core. To ease the development of plugins it is crucial to improve the core functionality of the framework. Thus, functions that are part of plugins are slowly imported to the core while cleaning up the code.

Documentation. Since currently there is close to no documentation besides this thesis it is necessary to include a very detailed public available documentation.

Slice Engine and C++. The slice engine needs to expose a better interface to work with for plugins. Hence, it will recieve a rework. This rework is also designed to handle C++ dynamic calls better.

GUI. Since the Binary Ninja devs announced a upcoming change to how the GUI system is working this also needs to be adjusted for the framework. Currently, the best way to get an extensive GUI is to hook the Qt GUI.

Pre-Processing Plugins. Currently there is a lack of pre-processing plugins as described in chapter 5.3.1. Hence, some decryption, deobfuscation, and unpacking plugins needs to be developed for the framework.

6 Conclusion

Discovering vulnerabilities is and will be an essential task for auditors and vulnerability researcher. In many cases, this valuable work is limited by time constraints and thus easy to spot vulnerabilities are overlooked. This thesis described and evaluated several approaches to finding vulnerabilities and picked a promising approach using the evaluated intermediate language.

This thesis introduced several low-level concepts of how compilers work and some algorithms and analyses they currently apply on the input, such as intermediate languages or the SSA form. Further, this thesis introduces an approach to discover vulnerability patterns including a detailed description of how such vulnerabilities can be spotted in the source code and the corresponding binary format. Finally, this thesis describes a process to semi-automate the discovery of vulnerabilities, using different plugins that were initially developed for the framework. The presented Buffer Overflow, Type Conversion, Large Stack Frame, and Uninitialized Variable Plugin contain already implemented approaches to find their associated vulnerabilities based on the presented test frameworks. The proof of finding such real-world vulnerabilities was presented with the Heartbleed plugin. These plugins are however only an initial proof of concept and still needs more development to model more security relevant vulnerabilities.

The effectiveness of the presented approach will only be shown when the framework is accepted by the information security community and more plugins are developed for the framework.

Note: The framework is developed on the author´s GitHub [36]. The framework is under heavy development and thus it might differ at the time of reading from the described code in this thesis.

6.1 Future Work

One of the most promising areas for further research is the adaption of our implementation to support more vulnerability patterns. Currently only some patterns for buffer overflows, type conversions and uninitialized variables is supported. Due to the rising significance of IoT devices these issues will hopefully be fixed in the future allowing the analysis of a massive amount of vulnerability patterns. Analysing such IoT devices in general seems to be a promising approach for the overall security of these devices. The downside is the exhausting process of disclosing the discovered vulnerabilites to the vendors.

Another current problem with the framework is that every plugin is performing the inital operation such as parsing all functions. This initial process is costly for the overall performance and it might be possible to multithread this process to reduce the amount of time the framework needs for running all plugins.

7 References

[1] URL: https://www.pine64.org/?page_id=3707.

[2] URL: https://developer.arm.com/docs/ddi0210/latest/introduction/architecture.

[3] URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0344i/Beiiegaf.html.

[4] URL: https://github.com/google/syzkaller.

[5] URL: https://gitlab.com/akihe/radamsa.

[6] URL: https://github.com/googleprojectzero/domato.

[7] URL: http://lcamtuf.coredump.cx/afl/.

[8] URL: https://github.com/google/honggfuzz.

[9] URL: https://blog.chromium.org/2012/04/fuzzing-for-security.html.

[10] URL: https://unicorn.360.com/en/index/.

[11] Moritz Lipp; Michael Schwarz; Daniel Gruss; Thomas Prescher; ... "Meltdown: Reading Kernel Memory from User Space". In: (2017).

[12] Rapid 7. *metasploit*. URL: https://www.metasploit.com/.

[13] Dennis Andriesse. *Practical Binary Analysis*. No Starch Press, 2018.

[14] *angr FOSDEM*. URL: https://fosdem.org/2017/schedule/event/valgrind_angr/attachments/slides/1797/export/events/attachments/valgrind_angr/slides/1797/slides.pdf.

[15] *appscan*. URL: https://www-03.ibm.com/software/products/en/appscan-source.

[16] *Binary Analysis Platform*. URL: https://github.com/BinaryAnalysisPlatform/bap.

[17] *Binary Ninja Intermediate Language*. URL: https://docs.binary.ninja/dev/bnil-llil/.

[18] *Binary Ninja Official API Documentation*. URL: https://api.binary.ninja/.

[19] BinaryNinja. *get_ssa_var_definition*. URL: https://api.binary.ninja/_modules/binaryninja/mediumlevelil.html#MediumLevelILFunction.get_ssa_var_definition.

[20] BinaryNinja. *get_ssa_var_uses*. URL: https://api.binary.ninja/_modules/binaryninja/mediumlevelil.html#MediumLevelILFunction.get_ssa_var_uses.

[21] Trail of Bits. URL: https://www.trailofbits.com/.

[22] Trail of Bits. *Vulnerability Modeling with Binary Ninja*. URL: https://blog.trailofbits.com/2018/04/04/vulnerability-modeling-with-binary-ninja/.

[23] Trail of Bits. *Your tool works better than mine? Prove it.* URL: https://blog.trailofbits.com/2016/08/01/your-tool-works-better-than-mine-prove-it/.

[24] Paul E. Black. *Juliet 1.3 Test Suite.* URL: https://www.nist.gov/publications/juliet-13-test-suite-changes-12.

[25] Sophia D'Antoine. *Be a Binary Rockstar.* URL: https://www.sophia.re/Binary-Rockstar/index.html.

[26] *DARPA.* URL: https://www.darpa.mil/.

[27] *drcov.* URL: http://dynamorio.org/docs/page_drcov.html.

[28] *DynamoRIO.* URL: https://www.dynamorio.org/.

[29] Forallsecure. *Mayhem.* URL: https://forallsecure.com/.

[30] *fortify.* URL: https://software.microfocus.com/en-us/software/sca.

[31] Hex-Rays. *Interactive Disassembler.* URL: https://www.hex-rays.com/index.shtml.

[32] Zero Day Initiative. *VMware Exploitation through uninitialized buffers.* URL: https://www.thezdi.com/blog/2018/3/1/vmware-exploitation-through-uninitialized-buffers.

[33] Intel. URL: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[34] Mateusz Jurczyk. "Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking". In: (2018). URL: https://j00ru.vexillium.org/papers/2018/bochspwn_reloaded.pdf.

[35] Mateusz j00ru Jurczyk. *Exploiting a Windows 10 PagedPool off-by-one overflow.* URL: https://j00ru.vexillium.org/2018/07/exploiting-a-windows-10-pagedpool-off-by-one/.

[36] Birk Kauer. *Zeno Framework.* URL: https://github.com/Traxes/zeno.

[37] legitbs. *What is DECREE.* URL: https://blog.legitbs.net/2016/05/what-is-decree.html.

[38] *Lighthouse Code Coverage Explorer for IDA Pro and Binary Ninja.* URL: https://github.com/gaasedelen/lighthouse.

[39] *Lighthouse drcov parser.* URL: https://github.com/gaasedelen/lighthouse/blob/master/plugin/lighthouse/parsers/drcov.py.

[40] *LLVM.* URL: https://llvm.org/.

[41] B. Chess; G. McGraw. "Static analysis for security". In: *IEEE Security and Privacy* (2004).

[42] Riku; Antti; Matti; Neel Mehta. *The Heartbleed Bug.* URL: http://heartbleed.com/.

[43] *OCaml*. URL: https://ocaml.org/.

[44] P. Oehlert. "Violating assumptions with fuzzing". In: *IEEE Security and Privacy* (2005).

[45] LLVM project. URL: https://llvm.org/docs/LangRef.html.

[46] LLVM project. *Clang: a C language family frontend for LLVM*. URL: https://clang.llvm.org/.

[47] LLVM project. *Scalar Replacement of Aggregates*. URL: https://llvm.org/docs/Passes.html#sroa-scalar-replacement-of-aggregates.

[48] *Pwn2Own*. URL: https://www.zerodayinitiative.com/blog/2019/1/14/pwn2own-vancouver-2019-tesla-vmware-microsoft-and-more.

[49] Ole André V. Ravnås. *Frida*. URL: https://www.frida.re/.

[50] Microsoft Research. *Z3Prover*. URL: https://github.com/Z3Prover/z3.

[51] Niklaus Schiess. *deen*. URL: https://github.com/takeshixx/deen.

[52] Mark Dowd; John McDonald; Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison Wesley Professional, 2006.

[53] David Brumley; Ivan Jager; Thanassis Avgerinos; Edward J. Schwartz. "BAP: A Binary Analysis Platform". In: (2011). URL: https://users.ece.cmu.edu/~aavgerin/papers/bap-cav-11.pdf.

[54] Yan Shoshitaishvili et al. *angr Framework*. 2016.

[55] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: UC Santa Barbara, 2016. URL: http://cs.ucsb.edu/~chris/research/doc/ndss16_driller.pdf.

[56] S. Rugaber; K. Stirewalt. "Model-driven reverse engineering". In: *IEEE Software* (2004).

[57] Ray Toal. *Intermediate Representations*. URL: http://cs.lmu.edu/~ray/notes/ir/.

[58] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society, Volume s2-42, Issue 1* (1937).

[59] Vector35. *Binary Ninja*. URL: https://binary.ninja/.

Appendix

Recommended Books

Book	ISBN
Practical Binary Analysis	978-1-59327-912-7
The Art of Software Security Assessment	B004XVIWU2

STATUTORY DECLARATION

I declare that I have authored this Thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Date: .....

(signature)