Information Security

Functionality Analysis of Binaries

Detecting Cryptography through LLVM IR Visualization

Diploma Thesis

For attainment of the academic degree of

Diplom-Ingenieur/in

submitted by

Patrick F. KOCHBERGER

is151511

in the

Masters Course Information Security at St. Pölten University of Applied Sciences

The interior of this work has been composed in LaTeX.

Supervision

Advisor: Dipl.-Ing. Dr. Sebastian Schrittwieser, Bakk.

Assistance: Damjan Buhov, MSc

St. Pölten, February 12, 2018

(Signature author) (Signature advisor)

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.
- diese Arbeit mit der vom Begutachter/von der Begutachterin beurteilten Arbeit übereinstimmt.

Der Studierende/Absolvent räumt der FH St. Pölten das Recht ein, die Diplomarbeit für Lehre- und Forschungstätigkeiten zu verwenden und damit zu werben (z.B. bei der Projektevernissage, in Publikationen, auf der Homepage), wobei der Absolvent als Urheber zu nennen ist. Jegliche kommerzielle Verwertung/Nutzung bedarf einer weiteren Vereinbarung zwischen dem Studierenden/Absolventen und der FH St. Pölten.

Ort, Datum *Unterschrift*

Kurzfassung

Die Untersuchung von unbekannten Programmen oder Schadsoftware macht es oft notwendig kryptographische oder andere Funktionalität aufzuspüren. Funktionalität als ein bestimmtes, mögliches Verhalten kann verwendet werden, um Software zu klassifizieren oder die Benutzerin/den Benutzer auf etwaige Folgen oder vorhandene nicht erwünschte Bestandteile hinzuweisen. Diese Analyse von Binärdateien kann ein unübersichtlicher und schwieriger Prozess sein. Um diesen zu vereinfachen und angenehmer zu gestalten ist es notwendig, neben Automatisierung, einem Analysten/einer Analystin einen Überblick zu geben. Die Darstellung als Bild kann nicht nur dabei helfen, sondern bietet auch die Möglichkeit visuelle Muster zu erkennen.

In dieser Arbeit liegt der Fokus auf statischer Analyse von Binärdateien sowie der Suche nach Funktionalität in ausführbaren Dateien. Der derzeitige Stand der Technik bezüglich Funktionalitätsanalyse und die Charakteristiken von kryptographischen Operationen wurden untersucht und ein neue Methodik zur Unterstützung der Analyse wurde vorgeschlagen. Dabei wird ein Programm durch die graphische Darstellung von LLVM IR Code für einen Analysten oder eine Analystin angezeigt.

Hierdurch können für bestimmte kryptographisch Muster für den Menschen sichtbar gemacht werden. Es ist mit der derzeitigen Visualisierung möglich diverse rechenintensive Funktionalität zu erkennen.

Abstract

The analysis of unknown or malicious software makes it often necessary to detect or identify several types of functionality, in particular cryptographic. In this thesis, the focus lies on static binary analysis and how functionality, represented by cryptography, can be detected in executables. The characteristics of cryptography and the current state-of-the-art for finding cryptography have been researched. A new approach for helping an analyst detect functionality through the visualization of LLVM IR is proposed. The visual aid, that is an output of this process, helps to render patterns visible and assist the analyst. The proof-of-concept implementation shows that the visual detection of certain cryptographic routines is easily possible, even without the necessity of a skilled expert. Analysis also showed that with the currently implemented type of visualization, AES functionality is very hard to detect, while other crypto algorithms can clearly and almost instantly be spotted. Similarly, the calculated ratio between arithmetic, bitwise and other operations provides an easier method for arriving at the same conclusion, but can easily lead to false positive findings.

Citation

This work can be cited using the following bibliography entries: (Listing 1,Listing 2).

```
@mastersthesis{kochberger2018,
    title = {Functionality Analysis of Binaries - Detecting Cryptography
       through LLVM IR Visualization},
    author = {Patrick, Kochberger},
    school = {University of Applied Sciences St. P\"{o}lten},
    type = {Diploma Thesis},
    year = {2018}
}
```

Listing 1: BibTeX

```
%0 Thesis
%T Functionality Analysis of Binaries - Detecting Cryptography through LLVM IR
     Visualization
%9 Diploma Thesis
%A Patrick KOCHBERGER
%D 2018
%I University of Applied Sciences St. P\"{o}lten
```

Listing 2: EndNote

Contents

1. Introduction

[Signature-based] Anti-Virus “is dead“.

— Brian Dye, Symantec’s senior vice president, Wall Street Journal 2014.05.04[1]

Technology, in particular Information Technology (IT) and software have made everyone’s life easier. The incorporation into business processes and even the day-to-day individual usage leads to a near-complete dependency on the correct functioning and availability of IT devices and systems.

Because of their importance, IT systems and infrastructures are an interesting and often profitable target for various types of cyber-attacks. Corporate and private networks and devices are **continuously threatened** by ever changing and evolving attacks. Every day new malicious software is being generated or adapted and reused.

For a long time, **signature based detection** has been used to counter this flood of malware and assaults. One problem with this strategy is that only known, already-seen Malware that has likely at some point even caused damage, can be turned into signatures. It fails to detect new or unknown malicious code with the same behavioral pattern. In some cases, changing even one byte in a binary may result in a totally different signature. Therefore, **behavioral analysis** is becoming more and more important in identifying Malware. In comparison, it is rather difficult to alter the behavior and changing the behavior can cause the desired (bad) functionality to be lost.

To give an analyst, end user or anti-malware program the ability to identify unwanted or harmful parts shipped inside an executable, the **machine code** has to be searched for certain **functionality**. An example of unwanted functionality would be encryption or searching for files in the files system or connecting to servers in the Internet. However, it is not limited to that.

Another challenge that still lies ahead is that once some kind of functionality has been found, it can mostly not be said if it is **good or bad**. What is good in one binary may not be desired behavior in another scenario. For example, encryption can be used to protect important files from unauthorized access. On the other hand, ransomware may use cryptography to deny the user access to his or her data and demand ransom money.

[1]`https://www.wsj.com/news/article_email/SB10001424052702303417104579542140235850578-1MyQjAxMTA0MDAwNTEwNDUyWj` - Last accessed: 2017.12.22

Why is the analysis of cryptographic operations in binaries important? Detecting **cryptography** and its context, e.g. which algorithm or what keys are used in binaries, helps to simplify the analysis process and aids the analyst to quickly identify important parts of a piece of software.

In addition, attention should be drawn to cryptographic functions, because of the associated mistakes [1, p.1] that can slip into a program when using or implementing cryptography: Insecure algorithms or implementations may have been used, or the implementation may not be according to specification. In addition, algorithms may leak information about secret data or the choice and generation of parameters (key,IV, padding) may be faulty or yield additional information about an application.

The current threat landscape poses a further **motivation**, especially **Ransomware**, but also other types of Malware heavily depend on cryptography.

The Cybercrime tactics and techniques reports [2] [3] by Malwarebytes show that in the first months of the year 2017, ransomware was the most used type of malware (see Figure 1.1).

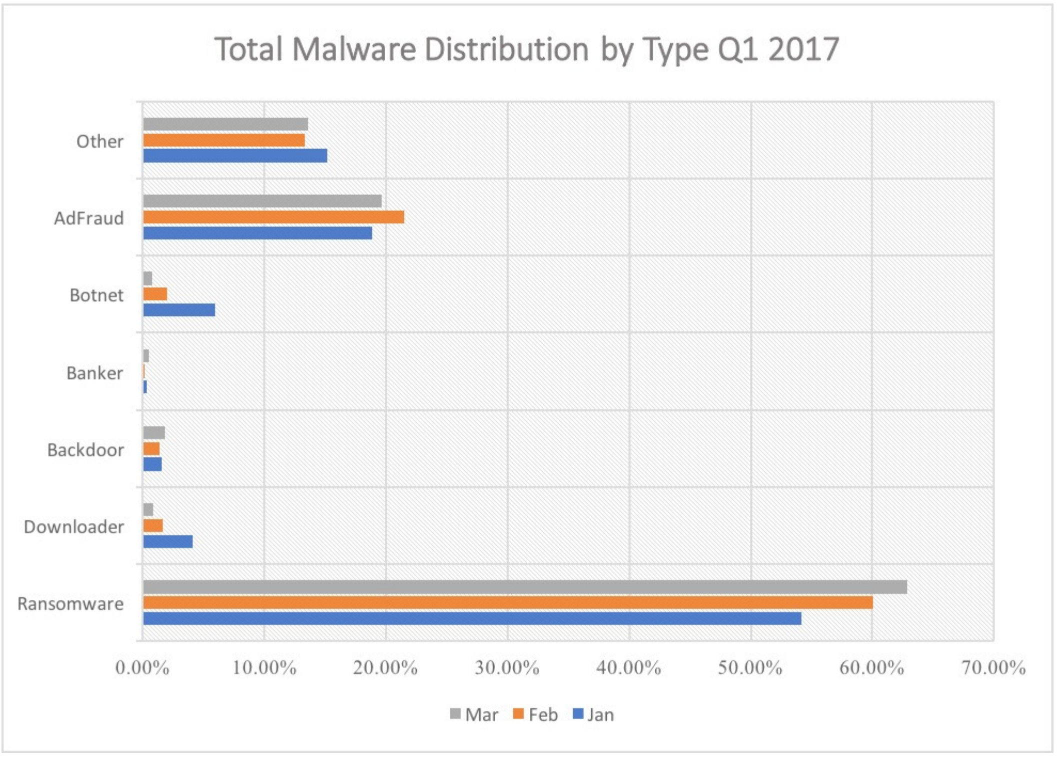


Figure 1.1.: Total Malware Distribution by Type Q1 2017 [2, p.2, Figure 1]

In Symantec's Internet Security Threat Report [4] as well as Malwarebyte's reports [2], Cerber and Locky are currently listed as the biggest ransomware threats.

1.1. Motivation

From theoretical computer science and computability theory, especially the work focusing on e.g. the halting problem [5] [6] by Alan Turing and Rice's theorem [7, p.312-314], the limitations of program analysis are well-known. [8, p.1]

The undecidability of the halting problem implies that there is no general way to decide the halting status for all programs. [9] On the other side, testing and proving any given program (a subset of all programs) can be possible. This means that several problems in program analysis can be linked to the halting problem and can only be reliably decided if the halting problem is solvable.

Furthermore, this leads to the realization that testing is only an approximation. A certain property or bug can only be detected by one approach and may not be found by another one. Alternatively, not finding it, does not prove the program does not have this property or bug. Therefore, no amount of exhaustive testing can give complete assurance of correctness or the absence of bugs.

For the analysis of executables, the undecidability, known through the halting problem and Rice's theorem, means that tools used can not give the correct answer for all programs. Sometimes they just might fail or run forever. [10]

The here mentioned limitations influence[2] the area of computer analysis, but do not prevent the work as a whole. They show the theoretical restrictions and that some inaccuracy has to be accepted.

1.2. Thesis Outline

The preceding sections emphasize the need for detecting functionality and features in executables, but also show the current theoretical boundaries.

Hence, this work focuses on the following question:

- What methods can be used to help an analyst identify cryptography in binaries?

This document is organized in several parts. The first part, chapter 1, introduces the topic, problems, challenges and motivation. After that, chapter 2 describes some prerequisites, basic and fundamental knowledge, terms and concepts. In chapter 3 the related work is listed. The proposed process and conducted evaluation are commented on in chapter 4 and chapter 5. At the end, the conclusions, future work and final thoughts are summarized in chapter 6. Finally, the setup, machines and programs used in this work are cataloged in Appendix B, while the source code of the samples, examples and programs can be found in Appendix A.

[2] `https://www.quora.com/What-are-the-implications-of-Turings-halting-problem-proof-and-how-has-it-affected-the-field-of-computer-science-today` - Last accessed: 2018.01.04

2. Program Analysis

Truth can only be found in one place: the code.
— Robert Cecil Martin, Author of Clean Code: A Handbook of Agile Software Craftsmanship, ISBN 9780132350884.

This chapter describes important terms and concepts regarding program analysis.

2.1. Introduction

Program analysis is the field of assessing the behavior of a given software. This not only helps to understand programs, but to also optimize them or find bugs and flaws. [11, p.11] A compiler can, for example, use program analysis during optimization to avoid redundant computations. [12]

The scope of program analysis can be divided into four categories. These are grouped into the measures taken during the analysis (**static** or **dynamic**) and the representation of the program that is analyzed (**source** or **binary** code). This classification splits program analysis into static source analysis, dynamic source analysis, static binary analysis and dynamic binary analysis. [13, p.1-3]

	static	**dynamic**
source	static source analysis	dynamic source analysis
binary	static binary analysis	dynamic binary analysis

Table 2.1.: Program analysis - Categories [13, p.3, table 1.1]

Source code analysis is done using the source code of a program. This category includes the examination of Intermediate Representations (IRs) directly derived from the source. An example for tools conducting this kind of investigation are compilers [14]. [13, p.2]

Binary code analysis, on the other hand, explores executable machine code, object code, byte code, or IRs that are lifted from them. [13, p.2]

From this point forward, this work only focuses on the area of binary analysis. Which means that, if not specifically noticed, whenever static or dynamic analysis is mentioned, static or dynamic binary analysis is meant.

Note, that source and binary is merely one possibility of categorization. It is entirely possible to segment program analysis into other aspects. It could, for example [15], be divided into manual and automatic, where e.g. manual static would reflect a code review and manual dynamic would be a security or penetration test.

2.2. Binary Analysis

Binary analysis uses **executable machine code** to examine a program. This includes object code and executable intermediate representations (byte code for virtual machines). [13, p.2]

Applications, utilizing binary analysis, range from malware detection, finding vulnerable components, reverse engineering and forensics to code optimization, measuring performance and debugging. [16, p.S11] [17, p.24]

There are several benefits of binary analysis.

The most important is that **no access to the source code** is required, which therefore also means that **no cooperation** from the programmer or author of the software is needed. There are two large sets of programs where only the binary code is available: Malware and Commercial-Of-The-Shelf (COTS) programs. [18, p.25] Legacy systems are another reason why the source code of the application may not be available.

Working on a binary level makes analysis techniques applicable to a wider range of samples. They can likewise be used for programs where the source is available, closed source programs and COTS applications. [18, p.3]

Also, the binary level has the highest **correctness** and **fidelity**, because the code at this level is what is actually executed. [18, p.25], [13, p.2] As shown and explained in previous work [19]–[21], the machine code may not always do exactly the same as the source code. Compilers may generate flawed code, either because of errors in the compiler software or undefined behavior [22], [23].

To counter undefined behavior or unstable code tools like STACK [24], a static checker for unstable code, or the Integer Overflow Checker (IOC) [25], which searches for undefined integer behavior, have been developed. However, bugs, introduced during the compilation process, can not be ruled out completely.

Furthermore, the machine code is **independent of the programming languages and the compilers** that have been used. [18, p.25], [13, p.2] On the other hand, this means that binary code is very specific and platform dependent. The style of the code is reliant upon the architecture and Operating System (OS). [13, p.2] To counter this disadvantage and make it easier for the analyst and analysis frameworks, Intermediate Representations and Intermediate Languages are used.

2.3. Static vs Dynamic Analysis

The following paragraphs describe and compare static and dynamic analysis. The main distinctions are summarized in Table 2.2.

	static	**dynamic**
coverage/paths	many or even all	one path (certain execution)
safe	safer, code is only looked at	less safe, because sample or parts of it are executed
runnable	does not have to be operable	has to be executable
speed/efficiency	usually slower	usually faster
soundness	sound	unsound
preciseness	less (more approximate)	more (actually executed)
used for applications that require	correct inputs	precise inputs
false positives	more	less

Table 2.2.: Comarison static vs dynamic analysis

Static analysis examines the program without running it and takes all possible situations that could arise into account. To do so, a model of the state of the application is built to see how it responds to the state. Since there are usually a huge number of states or user inputs it is only feasible and practical to keep track of so many. Hence, abstract models of the states are used to hide information, but make modifications and keeping track of the information easier. [13, p.2] [26, p.1] This results in a decrease in preciseness and therefore an increase in approximation. [26, p.1]
Typical fields of application are compiler optimization [14] [26, p.1] or semantic aware malware detection [27]. It has been shown that, although powerful, malware detectors utilizing static analysis can be fooled by obfuscation like scrambling control flow or hiding data. [28]

Dynamic analysis conducts an investigation of a program by **executing** it. This gives the opportunity to make statements about the actual behavior using runtime information. [29, p.1]

Dynamic analysis operates based on the executed instructions and the actual values. For this reason, no uncertainty can arise. It is clear which addresses are accessed and what data is used, which makes it easier if packing, encryption or other similar means of obfuscation are present. [18, p.25]

Dynamic analysis has no need to use abstractions or approximations and as a result is **precise**, meaning, that it is clear which path has been taken in the binary. [26, p.1]

This leads also to one disadvantage, the results are **not a general description** of the behavior of a program. Future runs may exhibit totally different parts of the same binary. Therefore, the main challenge in dynamic investigations is to find good, representative test cases. [26, p.2] One typical example of exploiting this drawback is malware. During analysis, the malicious operations are not performed, however afterwards they may be carried out. One possibility would be to simply not perform certain functionality the first few times it is executed, or not perform it when certain clues, pointing to an analysis environment, are detected.

This means that there are two types of properties: features unearthed because of a specific test case or test suite and real properties or characteristics of the program. [26, p.2]

The scope of the program that has been covered is called **coverage**. It can be measured by e.g. comparing all paths or instructions to those that have been executed. [18, p.26]

There are various techniques that fall into the class of dynamic approaches.

One such method is dynamic taint analysis, which runs a program and watches the flow of defined taint sources. [29, p.1]

Another is dynamic forward symbolic execution, which generates a logical formula that describes a certain path through the executable. [29, p.1]

Static and dynamic analyses take, in some regards, opposing sides, but are definitely not mutual exclusive. They can either supplement each other, or be combined into a hybrid analysis. [26, p.2] The combination allows for one to provide information to the other, that would otherwise not be available. [26, p.2]

2.4. Program Proving and Program Testing

To verify if a program is working correctly, there are several methods available. Either specific test cases are generated and **program testing** is applied or **program proving** is used to verify that all executions satisfy the specification. [30, p.385]

2.5. Compilation and Decompilation

Programming languages are used, by programmers, for writing computation instructions. They are employed to help people better understand code and also to embody the connection between human and machine. Software that translates source code into an executable form is called compiler. During the process of compilation, the source language is transformed into the target language. [14, p.1ff]

The term decompilation is used for the opposite process, in the reverse direction, meaning the translation from machine to source code. [11, p.12]

Similarly, disassemblers are the opposite of assemblers. The act of disassembling is the translation of binary code into assembly code. [31, p.45] Simply stated, they read bytes from the binary machine code and translate them by looking them up in a table. [32, p.11]

Arbitrary examples for disassemblers are:

- Binary Ninja [33]
- IDA Pro [34]
- objdump [1]
- Radare2 [35]

However, several challenges complicate this procedure. This mostly has to do with the loss of information during the compilation and assembly process. Not only the textual identifiers, such as names of variables, labels, functions or macros, but also the separation of code and data is lost. [32, p.15-16]

To help further understand the process of disassembling, the "big" picture of compilation and decompilation is visualized in Figure 2.1. There the whole process from C-code over binary code to again C-code can be overlooked. The boxes represent the single steps or functionalities that are carried out.

The left part shows the translation from source code to machine code. The source code is preprocessed, compiled, assembled and is linked with library or other assembled code. The result is an executable binary.

The opposite side is occupied by the phases of the analysis process. Here, the machine code is disassembled and can also be decompiled afterwards.

So far, the compilation process directly transforms the high level languages into the assembly code. Usually in between those two operations, optimization is carried out. Operations performed during the optimization phase can often be independent from the source coding language. Furthermore, the source code can be written in different programming languages and there could be several target architectures. This is where Binary Analysis Frameworks, Intermediate Representations (IRs) and Intermediate Languages (ILs) come

[1] `https://linux.die.net/man/1/objdump` - Last accessed: 2018.01.04

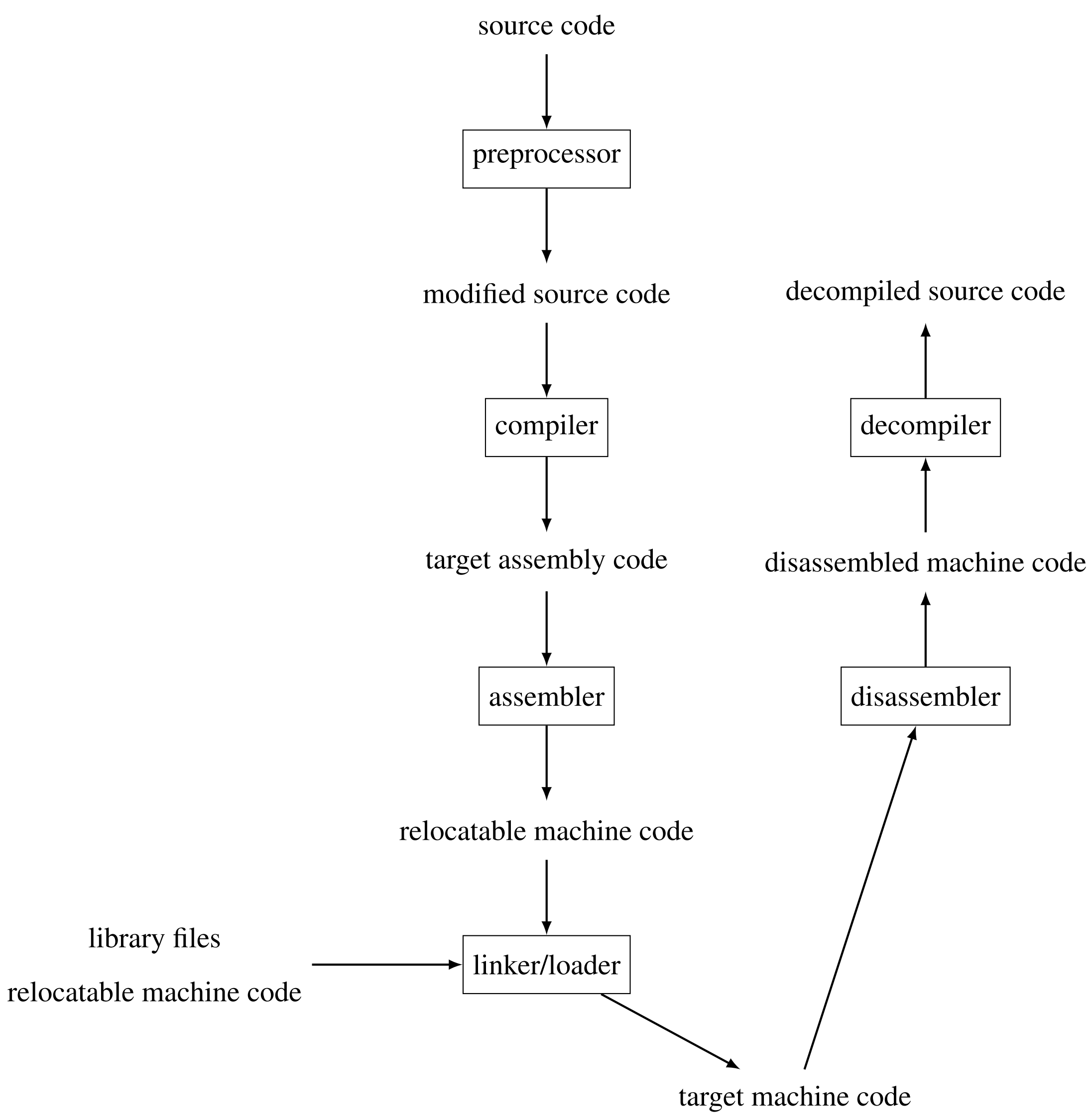


Figure 2.1.: Compilation, Decompilation Process. Adapted from [14, Figure 1.5] [32, p.3-19, Figure 1]

into play.

2.6. Binary Analysis Frameworks

Binary analysis frameworks are a collection of instruments for the examination of executables. They can be used as a foundation for binary analysis and help, compliant with the Don't Repeat Yourself (DRY) principle [36, p.27], to not repeat fundamental, but essential parts of every analysis project or tool. Features they offer vary, and usually include binary disassembling, debugging, function detection, graph generation and extraction.

The following list is only a small randomly hand-picked selection of the available frameworks:

- angr [37]
- BAP [38]
- Binary Ninja [33]
- Capstone [2]
- Hopper [39]
- IDA Pro [34]
- Radare2 [35]
- Retargetable Decompiler (RetDec) [11], [40]

In the end, for this work, **RetDec** has been chosen. This has many reasons, starting with it being free and open source. Furthermore, it is based upon LLVM.

Each framework has its own focus and specialty. To give an impression of the typical capabilities of binary analysis frameworks, the Radare2 Framework is shortly outlined in subsection 2.6.1.

2.6.1. Radare2 Framework

The radare2 framework is a collection of command line based tools. They can either be used independently or combined. The features of radare2 comprise, but are not limited to assembling, disassembling, debugging, searching and patching. [35, p.9]

Radare2 (r2) comes with its own Intermediate Language (IL), Evaluable Strings Intermediate Language (ESIL).

The presently included parts of the framework are listed in Figure 2.3.

However, the biggest advantage of radare2 is the r2pipe [3] API. It allows scripts and programs to control and send commands to radare2 and get the result as a string or JavaScript Object Notation (JSON). R2pipe is among others available for python, NodeJS, Swift and C.

Radare2 or r2 is the core or main application and can be described as a hexadecimal editor and debugger.

Similar to the tool readelf, the rabin2 tool extracts and displays information about executables.

The assembler and disassembler component of the framework is called rasm2 and together with a compiler for tiny binaries, ragg2, they make it possible to generate simple binary code.

For the comparison and integrity of binaries radiff2 and the block based hashing application rahash2 can be utilized.

[2] `http://www.capstone-engine.org/` - Last accessed: 2018.01.04

[3] `https://github.com/radare/radare2-r2pipe` - Last accessed: 2018.01.04

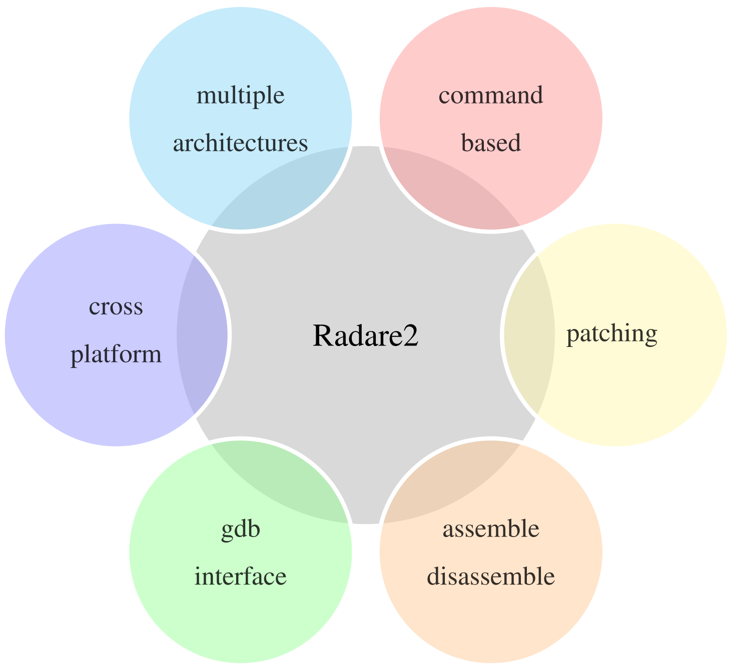


Figure 2.2.: Features of radare2

If the analysis requires different environments, rarun2 can be used. It can be directed by simple scripts and can for example change the arguments, permissions or directories for the analysis.
Another handy tool rax2 can evaluate mathematical expressions and convert different representations, like hexadecimal to integer or American Standard Code for Information Interchange (ASCII) strings to hexadecimal representation.

2.7. Intermediate Representations and Intermediate Languages

Intermediate Representations (IRs) and Intermediate Languages (ILs) represent the operations that will be done on the target machine, but hide the specific machine details. [41, p.148]
IRs are utilized by compilers [14], [41], to simplify the design and binary analysis or reverse engineering tools [42], to allow cross platform analysis. [43, p.241]
The term is also used for the translated code of high level languages, like Java or C#, that are not compiled to machine code. [44, p.1]
Compilers use Intermediate Representations (IRs) to enhance portability and modularity. The front end translates into the IR and the back end takes it from there. Consequently, the front end does not have to deal with machine specific technicalities and the number of different architectures. [41, p.148][45, Section 11.1] This advantage is illustrated by Figure 2.4, where the lines between the languages and architectures

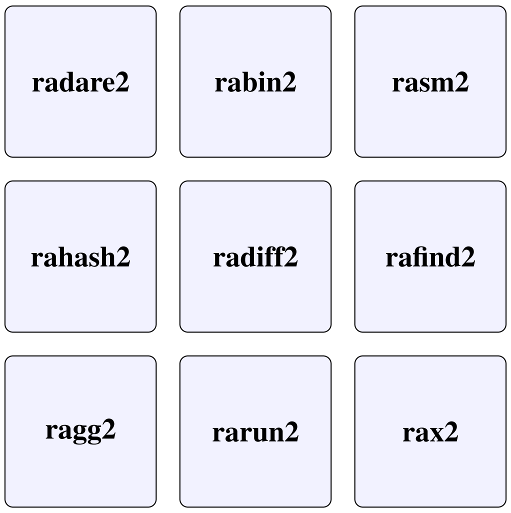


Figure 2.3.: Overview of radare2

represent the necessity for translations.

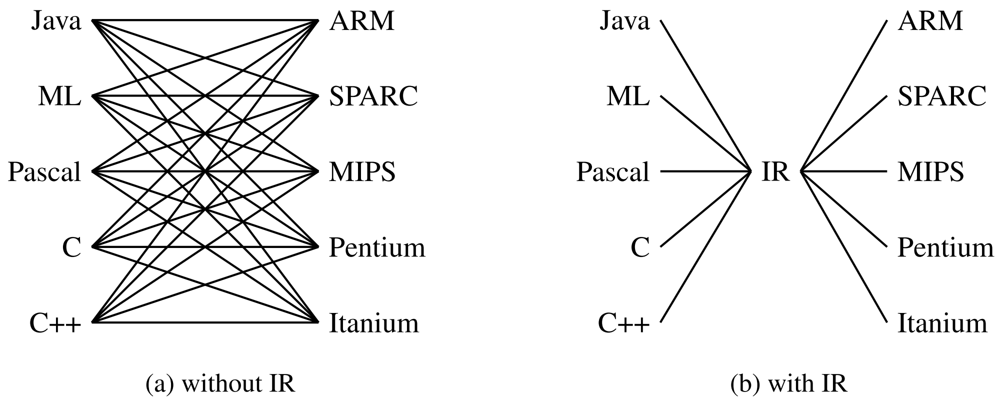


Figure 2.4.: Compiler design for several languages and target systems. Adapted from [45, Figure 11.2] [41, Figure 7.1]

Similarly, in the field of malware detection, binary analysis and reverse engineering, Intermediate Representations make solutions portable and simplify the process by abstracting the program code. [44, p.1]
In program analysis, the IRs are used to normalize the code for the following steps, they therefore have to deal with the differences of diverse architectures. The analyst and tools based on IRs do not have to worry about the underlying diversity e.g. different names of registers or the instruction set.
The act of translating binary executable code into high-level intermediate representations is called binary lifting. Kim *et al.* introduce an approach for systematic testing of binary lifters. [46]
Quite a few IRs have been developed, some specifically for malware and program analysis. The following

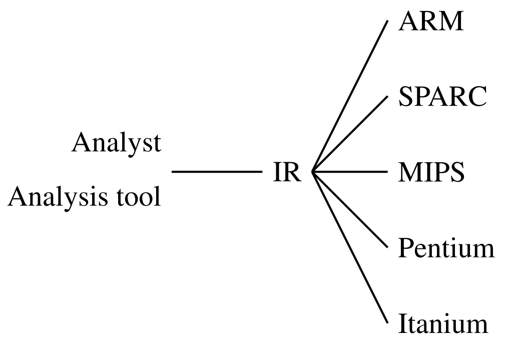


Figure 2.5.: Analysis utilizing IRs

arbitrary, non-exhaustive list represents only a subset of the available IRs:

- **ESIL** [35, section 1.8.2]

  The radare2 framework uses ESIL, a language based on evaluable strings. [35, section 1.8.2]

- **LLVM IR** [47]

  IR used by the LLVM framework. See subsection 4.2.3.

- **Malware Analysis Intermediate Language (MAIL)** [44] [48]

  MAIL is a simple and small, but extensible language, that was developed to simplify the analysis of Malware capable of dynamic obfuscation and metamorphism. Furthermore, MAIL statements can also be assigned patterns to allow annotating Control Flow Graphs (CFGs).

- **Reverse Engineering Intermediate Language (REIL)** [49]

  REIL is used by the binary analysis IDE BinNavi[4]. It is a platform-independent IL capable of absorbing disassembled assembly code and designed with static code analysis and reverse engineering in mind.

- **Vine IL**

  One of the two ILs used by BitBlaze [50] (Vine IL - low level; VEX IL - high level).

- **VEX**

  The VEX IR has been developed alongside the tool Valgrind [51, p.92-96], but has since also been used by Shoshitaishvili *et al.* in their tool Firmalice. They created pyvex [5], python bindings for the libVEX and open sourced them. The VEX IL is also used by the BitBlaze framework [50].

- **WIRE**

  The WIRE IR has been developed with static analysis in mind. It contains higher level constructs like functions (calls and arguments), dynamic memory allocation. It is formally defined using BNF. [53]

[4]`https://github.com/google/binnavi` - Last accessed: 2018.01.04

[5]`http://angr.io/api-doc/pyvex.html` - Last accessed: 2018.01.04

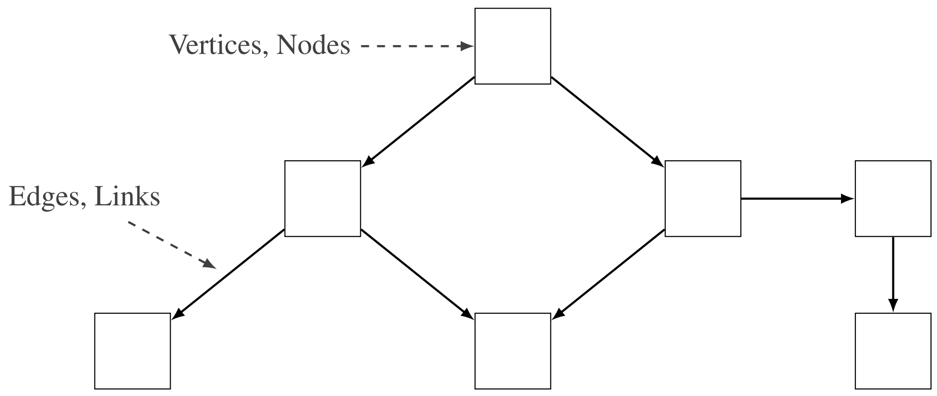


Figure 2.6.: Visualization and components of a graph

That cross platform analysis and the usage of IRs for that is feasible, has been shown by Pewny *et al.* They developed a proof-of-concept tool that does semantic cross-architecture matching of binary code. Their approach uses IDA Pro to extract the CFG and disassembly, translate it into VEX IR and use pyvex and the z3 theorem prover for the normalization of the expressions. They then use random input to get the semantic I/O behavior of a basic block and combine and compare those using MinHash. This generates signatures that can be used for matching. [42]

2.8. Graphs

Graphs consist of a pair of sets $G = (V, E)$. The elements of the first set, V, are **vertices**, the ones from the second, E, are called **edges**. The edges are unordered pairs of 2 elements from V connecting these vertices (v_i, v_j). The order of a graph is equal to the number of vertices and the size of a graph equals to the number of edges. [54, p.4]

Graphs can be visualized (as in Figure 2.6) by depicting the nodes as some sort of two-dimensional geometric shape (e.g.: rectangles) and the edges as lines (e.g.: arrows) connecting the vertices.

Directed graphs or short digraphs, consist of vertices and directed edges. The can be denoted by $G = (V, E)$ where $V = \{v_1, v_2, v_3, ..., v_n\}$ and $E = \{(v_i, v_j), (v_k, v_l), (v_m, v_n), .../v \in V\}$. The directed edges are ordered pairs e.g. (v_i, v_j), where the order implies that the edge goes from the first element v_i to the second vertex v_j. [55, p.2] [54, p.11]

In a directed graph, a **path** or subpath is a series of nodes that are connected with each other. In this sequence of nodes $n_1, n_2, n_3, ...n_k$, every node n_i for $i = 1, 2, 3, k-1$ is connected with its successor n_{i+1}. [56, p.59f]

A directed graph is called acyclic if it has no circuits. [54, p.15] This means that in Directed Acyclic Graphs (DAGs), for a certain vertex v_i there is no possible path looping back to it.

A **subgraph** G' of a directed graph is again a directed graph, being formed of a subset of the vertices and edges. It can be denoted by $G' = (V', E')$, where $V' \subset V$, $E' \subset E$ applies and $G \cap G' = G'$, $G \cup G' = G$ holds true. [55, p.2]
Various versions of graphs are used in program analysis. They can be grouped into the categories by the information they provide. This results in the types of graphs depicting control, data and hybrid dependencies or flows.

2.8.1. Control Flow Graph (CFG)

The idea regarding CFGs is to use a directed graph to express the **control flow**. [55, p.1] A CFG is a directed graph that links basic blocks of the program with transitions. [52, p.7] The nodes consist of a linear sequence of instructions, and edges are pairs of nodes, where the second is a successor of the first. [55, p.2]
A **basic block** can be connected to many other basic blocks and even itself, but only has one entry point and one exit point, meaning that it is not possible to have branching in the middle of the block. [55, p.2]

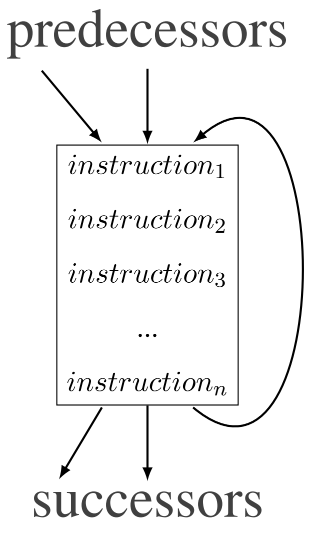


Figure 2.7.: Basic block

A CFG, generated through static analysis, usually models all possible executions of a program. Consequently, one path through the graph equals one execution.
Similarly, the term Control Dependency Graphs (CDGs) is used for graphs, visualizing, which statements r determine if a certain statement s is executed. [52, p.7]
CFGs have various areas of application, e.g. control flow integrity, compiler optimization or binary analysis. Control-Flow Integrity (CFI) is a technique for hardening software. CFI and the enforcement thereof aims to enhance security in programs. This is done by constraining the control flow, which means that changes

of the control flow are restricted to a set of locations. These locations come from a predetermined CFG. Therefore, any deviation form the path of the known targets, that are usually gained by CFG, is considered a violation of the security policy. [57] [58] [59] [60]
Approaches for the implementation of CFI on a binary level include PathArmor [61], bin-CFI [62] and CCFIR [63].
The possible usage of the CFG for binary analysis and malware detection is shown with the aid of the Enriched Control Flow Graph Miner (ECFGM) [64].
ECFGM uses an enriched version of the CFG. The additional information comprises of statistical information about the assembly instructions and API calls. Before the generation of the ECFGM, the input PE-file is disassembled and normalized. The nodes of the graph are then used for machine learning. [64]

2.8.2. Examples

To further deepen the understanding of CFGs, some basic patterns are discussed and illustrated. To generate the following examples, Listing A.1 has been used.
If-statements and branches are ordinarily represented by CMP and JMP operations in the assembly language. In CFGs, an “if“ is represented by two arrows at the end of a block, leading to two other basic blocks. These two blocks embody the then (true) or the else (false) path.

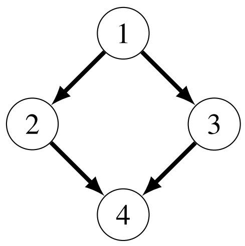
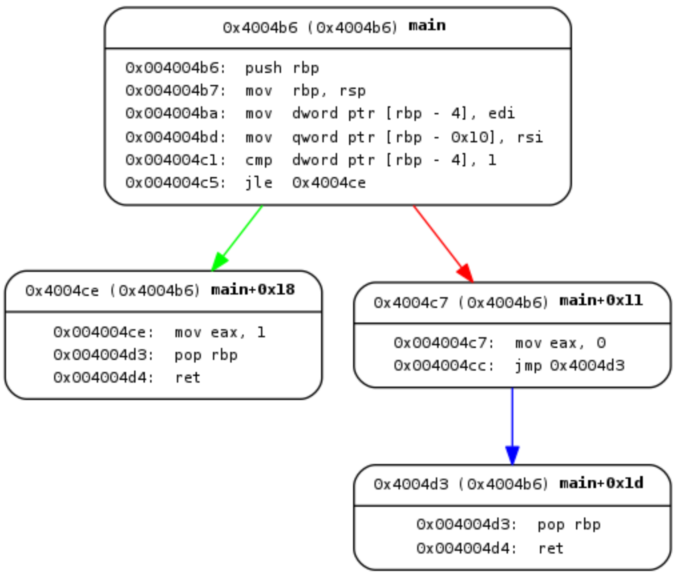


Figure 2.8.: CFG Theory - if

In Figure 2.8 the execution after the “if“ is resumed at basic block 4.
Figure 2.9 shows a CFG of a simple program only containing an if-statement. The assembly instructions are listed inside the basic blocks. The graph represents the code from Listing A.5.
Loops pose another pattern and can be spotted by searching for circles in the graph.
Figure 2.10 shows how a do-while-loop will be represented in a CFG. The node with the number 2 is where the while-decision is checked. If it results in a true, the loop begins from its beginning, in this case node number 1. Otherwise, the execution is continued after the loop, which here is node number 3.
To demonstrate this Listing A.4 was compiled and the generation of the CFG resulted in Figure 2.11.



```
0x4004b6 (0x4004b6) main
0x004004b6:  push rbp
0x004004b7:  mov  rbp, rsp
0x004004ba:  mov  dword ptr [rbp - 4], edi
0x004004bd:  mov  qword ptr [rbp - 0x10], rsi
0x004004c1:  cmp  dword ptr [rbp - 4], 1
0x004004c5:  jle  0x4004ce
```

```
0x4004ce (0x4004b6) main+0x18
0x004004ce:  mov eax, 1
0x004004d3:  pop rbp
0x004004d4:  ret
```

```
0x4004c7 (0x4004b6) main+0x11
0x004004c7:  mov eax, 0
0x004004cc:  jmp 0x4004d3
```

```
0x4004d3 (0x4004b6) main+0x1d
0x004004d3:  pop rbp
0x004004d4:  ret
```

Figure 2.9.: CFG Angr Example - if

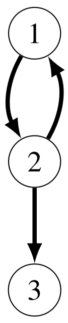


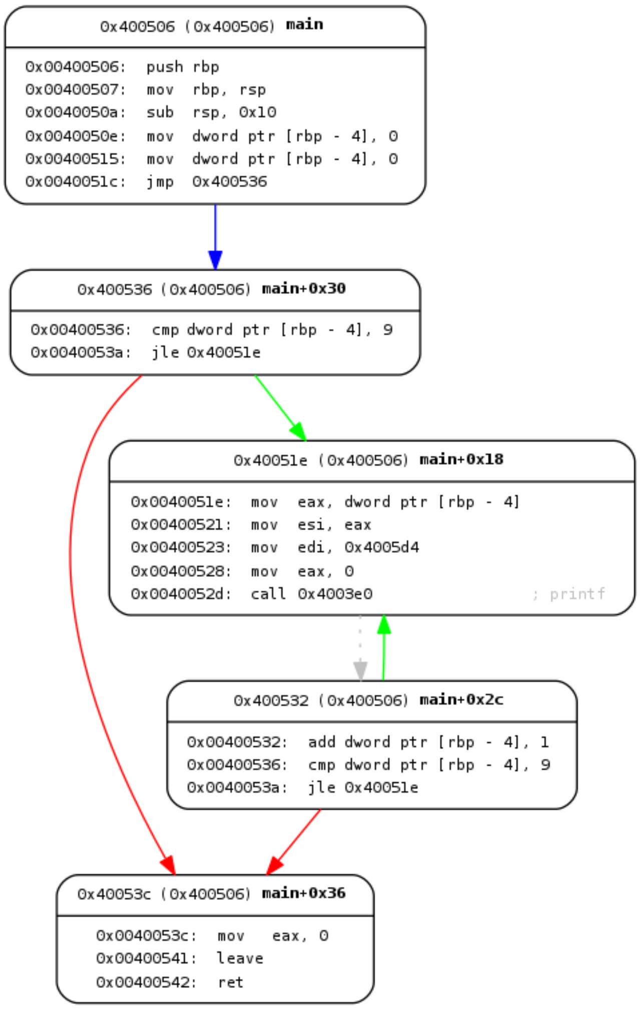
Figure 2.10.: CFG Theory - loop

2.8.3. Call Graph (CG)

The Call Graph (CG) or call multigraph [56] is an hierarchical **abstraction of the control flow**. It can be seen as the control flow on a **function level** and shows what procedure calls which other functions. CGs depict the calling relationships between subroutines. The nodes impersonate functions and the edges portray the caller-callee affiliation.

The CG is perfect for gaining a general view of a program, but also contains valuable intelligence for an analyst.

Faruki *et al.* even employed the Call Graph (CG) for Malware detection. They propose the following steps: They check for packer signatures and, when necessary, unpack the executable with the hardware virtualization unpacker ETHER. IDA-Pro is used for disassembling. Afterwards, unwanted and unneeded elements are removed before generating the CFG, from which the CG is built. From there, they extract the



```
0x400506 (0x400506) main
0x00400506:  push rbp
0x00400507:  mov  rbp, rsp
0x0040050a:  sub  rsp, 0x10
0x0040050e:  mov  dword ptr [rbp - 4], 0
0x00400515:  mov  dword ptr [rbp - 4], 0
0x0040051c:  jmp  0x400536
```

```
0x400536 (0x400506) main+0x30
0x00400536:  cmp dword ptr [rbp - 4], 9
0x0040053a:  jle 0x40051e
```

```
0x40051e (0x400506) main+0x18
0x0040051e:  mov  eax, dword ptr [rbp - 4]
0x00400521:  mov  esi, eax
0x00400523:  mov  edi, 0x4005d4
0x00400528:  mov  eax, 0
0x0040052d:  call 0x4003e0                  ; printf
```

```
0x400532 (0x400506) main+0x2c
0x00400532:  add dword ptr [rbp - 4], 1
0x00400536:  cmp dword ptr [rbp - 4], 9
0x0040053a:  jle 0x40051e
```

```
0x40053c (0x400506) main+0x36
0x0040053c:  mov  eax, 0
0x00400541:  leave
0x00400542:  ret
```

Figure 2.11.: CFG Angr Example - do-while-loop

Application Programming Interface (API) calls and use the API callgrams as feature vectors. These feature vectors are the input for data-mining with WEKA[6] [66], which classifies the executable, separating them into malicious and benign. [65, p.3]

2.8.4. Data Flow Graph (DFG)

Data Flow Graphs (DFGs) and Data Dependency Graphs (DDGs) or Value Dependence Graphs (VDGs) [67] [68] [69, p.55] [70, p.263] display **dependencies of operations**. [1]

Data Flow Graphs (DFGs) are DAG. The nodes in this graph represent input variables or arithmetic and logic operations. The edges link operands with the operations. [1, p.3-4]

The inputs are an unordered set, meaning that they are all equal and can either be an operation or an input

[6] `http://www.cs.waikato.ac.nz/ml/weka/index.html` - Last accessed: 2018.01.04

variable. Tags (special labels) can be used to distinguish, in cases where the order is relevant. [1, p.3-4]

Input variables can be constant (fixed, known value) or non-constant (register or memory locations). [1, p.3-4]

The DFG $G = (V, E)$ is generated by iterating over the instructions $i \in P$ of the program P. [1, p.3-4]

In high level languages, local and global variables, parameters or constants plus the corresponding operations make up the nodes. On a binary level, registers, flags or memory addresses are the equivalents. [71, p.5]

2.8.5. Examples

A function, representing the quadratic formula (Equation 2.2) would have the DFG depicted in Figure 2.12.

$$ax^2 + bx + c = 0 \tag{2.1}$$

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.2}$$

The quadratic formula is a way of calculating the solution of quadratic equations (Equation 2.1) in elementary algebra. The unknown is x and a, b, c are constants, where $a \neq 0$.

For the subtraction (Equation 2.3) and division (Equation 2.4), where the order of operands is not commutable, the edges have labels to specify the assortment.

$$\begin{array}{r} minuend \\ -subrahend \\ \hline difference \end{array} \tag{2.3}$$

$$quotient = \frac{dividend}{divisor} \tag{2.4}$$

Another simple example for understanding the DFG is the Pythagorean theorem (Equation 2.6). The graph in Figure 2.13a again represents the DFG.

$$c^2 = a^2 + b^2 \tag{2.5}$$

$$c = \sqrt{a^2 + b^2} \tag{2.6}$$

In the DFGs so far, the edges had to store the intermediate results.

Another model introduces intermediate nodes to store the results. The figures 2.13a and 2.13b make a direct comparison between approaches storing intermediate results and those that do not.

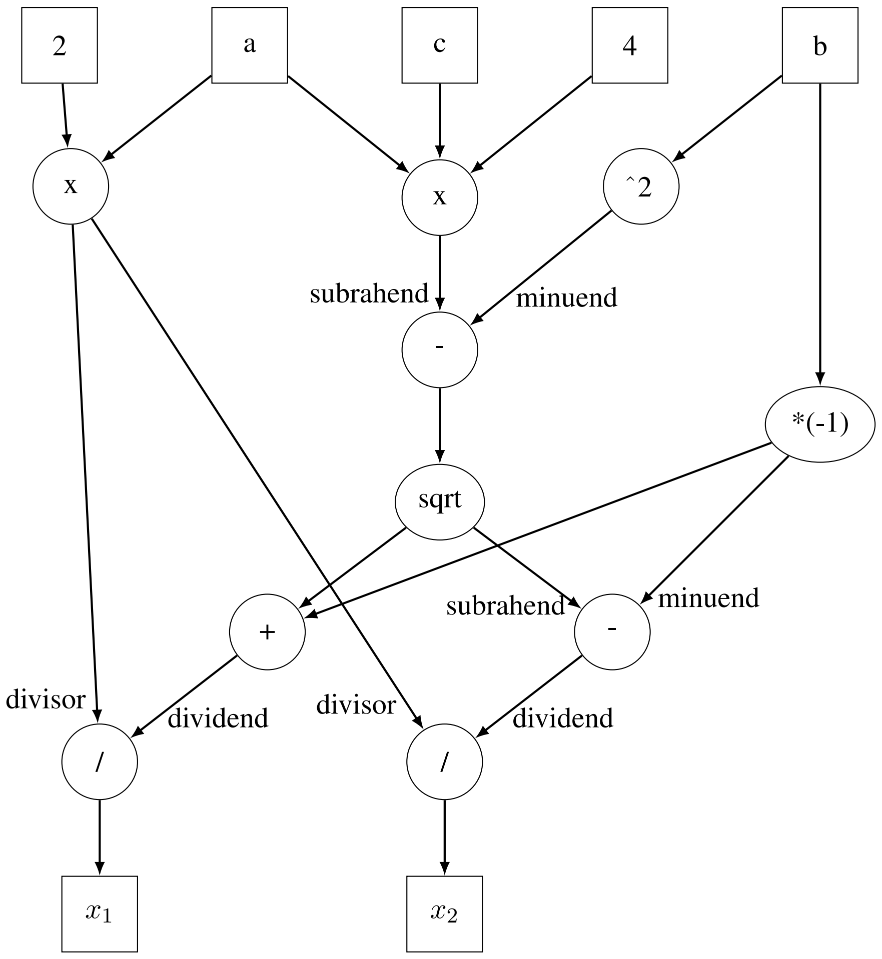
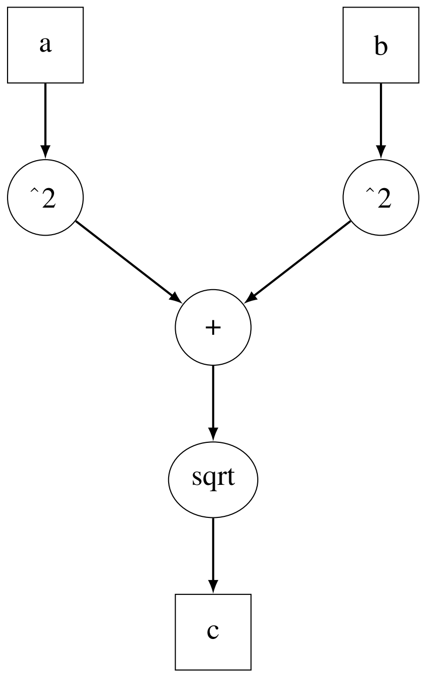
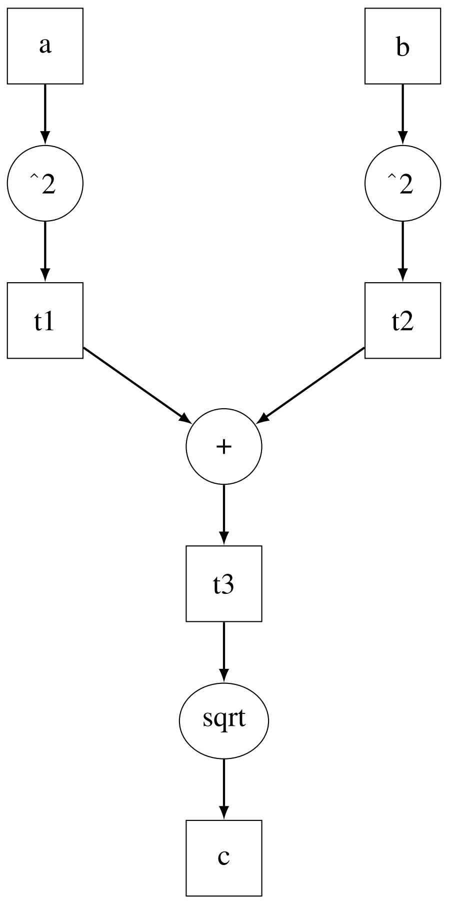


Figure 2.12.: DFG of the quadratic formula



(a) DFG of the Pythagorean theorem without storing intermediate results



(b) DFG of the Pythagorean theorem using intermediate result storage

Figure 2.13.: DFG of the Pythagorean theorem

2.8.6. Hybrid Graphs

The term hybrid graph is used in this document for graphs that combine control and data graphs.

One such specimen is the Program Dependence Graph (PDG), which shows both the control and data dependencies. [72, p.320]

Nodes act as statements, predicate expressions or operators and operands. [72, p.322]

Data dependency is the effect that two statements, s_1,s_2 would mean something different if reversed. [72, p.322]

The example in Listing 2.1 illustrates data dependency. It calculates d in line 2 using a, a itself is calculated in line 1. By interchanging the two lines, d would not have the same value as before.

```
1 a = b + c;
2 d = a + d;
```

Listing 2.1: Data dependency example

Control dependency is the relation between a statement s_1 and a variable or predicate, influencing the execution of said statement. [72, p.322]

For instance, in Listing 2.2 the execution of line 2 hinges on the outcome of the condition in line 1.

```
1 if(a){
2     b = c + d;
3 }
```

Listing 2.2: Control dependency example

The PDG can be generated by combining the CDG and the Data Dependency Graph (DDG). [52, p.7] [71, p.4]

Improvements, such as the System Dependence Graph (SDG), being a PDG but with inter procedural support and other variations like the Hybrid Information and Control Flow Graph (HI-CFG) [73], have been developed.

The tool BinGold [16] uses a hybrid graph called Semantic Flow Graph (SFG) to extract the semantics of 2016
binary code and calculate similarity.

Their process consists of disassembling, normalizing and extracting the normalized instructions, DFG and SFG. The instructions are used for exact matching, while the DFG allows the calculation of the graph edit distance, and the similarity measurement is computed from the SFG. Those features are then used to get the semantic similarity of binaries. [16, S17]

2.9. Slicing

Program slicing is the act of dissecting and **decomposing** a program by taking the data and control flow into account. The program is reduced and minimized until only a **desired behavior** or functionality is left. The result is called a slice and represents a subset of the original application. [74, p.352] The point of interest or **slicing criterion** is usually a pair consisting of program point and a set of variables. Slicing comes in the two variants: static, making no presumptions about the input and dynamic, starting with a specific test case. [69, p.1]

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 
 4 int main(int argc, char *argv[])
 5 {
 6     if (argc != 2) {
 7         printf("Usage: %s n\n",
               argv[0]);
 8         exit(1);
 9     }
10 
11     int n = atoi(argv[1]);
12     int i=1;
13     int sum=0;
14     int prod=1;
15 
16     for(i=1;i<=n;i++){
17         sum += i;
18         prod *=i;
19     }
20     printf("SUM: %d\n", sum);
21     printf("PROD: %d\n", prod);
22     return 0;
23 }
```

Listing 2.3: C program calculating sum and product of values from 1 to n

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 
 4 int main(int argc, char *argv[])
 5 {
 6     if (argc != 2) {
 7         printf("Usage: %s n\n",
               argv[0]);
 8         exit(1);
 9     }
10 
11     int n = atoi(argv[1]);
12     int i=1;
13 
14     int prod=1;
15 
16     for(i=1;i<=n;i++){
17 
18         prod *=i;
19     }
20 
21     printf("PROD: %d\n", prod);
22     return 0;
23 }
```

Listing 2.4: Sliced version reduced to the product functionality

Listing 2.3 and Listing 2.4 [69, idea from p.2, figure 1] demonstrate the principle of slicing on a source code level. The first listing shows a simple C code calculating the sum and product of the numbers 1 to n, where n is entered by the user. The second shows the sliced version of the same program. The criterion therefore was to reach line 21 and the variable prod.

Slicing is used for debugging purposes, parallelization or preprocessing a program for further analysis or when reusing certain elements of software.

2.10. Function vs Functionality

In this work, a function is defined as a group of operations that have been clustered, e.g. by the programmer. In contrast, functionality is some specific kind of feature of an application that can stretch over multiple programming functions or only be a small part of a function. This vice versa means that a function can provide a single functionality or combine multiple.
Example: E.g.: the functionality AES-encrypt is made up of the functions add_roundkey, sub_byte, shift_rows and mix_columns.
Another example would be the write-to-disk functionality. In the C programming language it can either be constructed by using fopen(...) and fprintf(...) or by using syscall(...) directly.
Functionality can, but does not have to, be provided by a function. This becomes more vivid by looking at it from the point of the compiler and decompiler. For example, when compiling from C to the 32-Bit Intel Architecture (IA-32), the CALL and RET instructions are used for accessing and exiting functions. A compiler may also decide that a particular function is being inlined. If the resulting code is decompiled, the function can be lost, while the functionality is still there.
In Listing 2.5 the programmer wrote two functions add(...) and sub(...) for adding and subtracting two double variables. During the compilation process, the compiler can then decide if the add function will be inlined. If the compiled program is again decompiled, the decompiled c code can look similar to Listing 2.6.

```c
double add(double a, double b) {
    return a + b;
}
double sub(double a, double b) {
    return add(a, -b);
}
```

Listing 2.5: C source code for addition and subtraction

```c
double sub(double a, double b) {
    return a + -b;
}
```

Listing 2.6: Inlined C source code for addition and subtraction

2.10.1. Functionality Finding

Identifying functionality can be archived in various ways. As shown in Figure 4.6 it is possible to search for specific characteristics of certain functionality at various levels.
When looking for vulnerability or security relevant functionality, some functionality is more interesting than others and is a more promising candidate for finding vulnerabilities that can be exploited. Authentication,

memory corruption, user input validation, encryption, file system or internet access, only to name some, are often part of malware and vulnerable software.

To explain the process of detecting functionality and corresponding vulnerabilities, a few examples have been listed. The tool Firmalice [52] is looking for authentication bypasses and backdoors. Driller [75], searches for memory corruption vulnerabilities.

If the goal is not to find out if a binary contains certain, already previously known behavior, but to generally list all kind of known reused implementations and provide an overview, function identification can be applied.

2.10.2. Function Identification

The identification of functions is an important part of binary analysis. It also helps to categorize software and to quickly find out what kind of functionality it may contain. The challenge here lies in the loss of information during the compilation phase. The names and affiliations of functions are lost during that step. There are several approaches and propositions, like signature based or machine learning solutions.

They support the analysis process by identifying reused, well-known patterns of widely used functions and libraries, or publicly available open source software, as e.g. [76].

A tool named **unstrip** has been developed and extended [77] to fingerprint functions from the GNU C library.

IDA Pro, using its **F**ast **L**ibrary **I**dentification and **R**ecognition **T**echnology (FLIRT), relies on patterns for library function detection. First, certain signatures are used on the entry point to identify the compiler. Afterwards, the whole binary is scanned using compiler specific signatures. [34, p.211ff]

Other approaches utilize machine learning for the identification process. E.g. ByteWeight [78] or [79], where Shin *et al.* use neural networks.

In 2017, the tool Nucleus [80] was developed. It uses the CFG [80, p.] and searches for functions. The tool has no training or maintenance phase. It does not generate signature databases and therefore is independent of the compiler. [80, p.2,12]

2.11. Symbolic Execution

Symbolic execution is the act of using representations (symbols) as inputs when running a program. This method is a mixture of program proving and testing. [30, p.385]

The idea comes from software testing. [81, p.1]

Instead of concrete, symbolic input values are used. The execution of the program is carried out by a **symbolic execution engine**. For each path, a state (ι, σ, π) has to be obtained. It is made up of the next statement to be evaluated ι, a **path constraint** π being a **first order boolean formula** and the **symbolic memory store** σ. [81, p.1-2] The formula is a set of constraints that must be satisfied to reach this path. [82, p.209] It is made up of an expression using α_i and is a result of the paths taken. [81, p.2] The store must hold the knowledge of which program variable is mapped to what symbolic value. [81, p.1] It links the variables using expressions, concrete and symbolic values α_i. [81, p.2]

The next statement or instruction ι, when evaluated affects what happens next. There are three changes [81, p.2-3] that can occur:

- Unconditional jumps like goto in the C programming language change the next statement to be evaluated ι.
- Assignments alter the variables in the symbolic store.
- Conditions or branches (in C: if) affect the path. This means that the execution has to be forked and the path conditions have to be augmented. Let an example be, if e then $instruction_{true}$ else $instruction_{false}$. The two branches would then have the updated constraints: $\pi_t = \pi \wedge e_e$ and $\pi_f = \pi \wedge \neg e_e$, where e_e is the evaluated formula of e. Accordingly the next statement will be $\iota_t = instruction_{true}$ and $\iota_f = instruction_{false}$.

Model checkers, specifically Satisfiability Modulo Theories (SMT) solvers determine if paths are within reach, meaning if some concrete values can be assigned to reach the path. [81, p.2]

The results can be represented in an **execution tree**. [83, p.1]

While in theory, every possible control flow path is explored, real life samples have shown that there are several obstructions (some are listed in subsection 2.11.1) hindering analysis. [81, p.5]

2.11.1. Challenges and Limitations

There are several challenges, limitations and areas of improvement, which are described in the following paragraphs.

Constraint solving: The process of evaluating constraints is one of the key bottlenecks, slowing down the whole analysis and playing a dominant role in the assessment of runtime. To counter that symbolic execution engines try to apply optimizations. One optimization being to eliminate non relevant constraints or using the similarity of paths for incremental solving. [83, p.5] [81, p.4]

Environment interaction: The program may interact with the environment by opening files, communicating with the network or using environment variables. This may lead to immediate side effects that could influence the execution later on. For example, creating a file that is used later on and has therefore been taken into account. [81, p.4]

Loops: The critical task here is picking the right number of iterations, if the loop conditions can not be assessed. This can, for example, happen when loops depend on environment or input parameters. [81, p.4]

State or Path explosion: Some constructs (e.g. in higher languages loops and recursive functions) might lead to a lot of forks and drive the number of execution states up. To overcome the obstacle of limited resources, practical tools work with heuristics to lead the path or prioritize states. [81, p.4]

2.12. Concolic Execution

To overcome some limitations of symbolic execution, it can be combined with concrete execution. This approach is named concolic execution and is a portmanteau of **conc**rete and symb**olic**. [81, p.5]

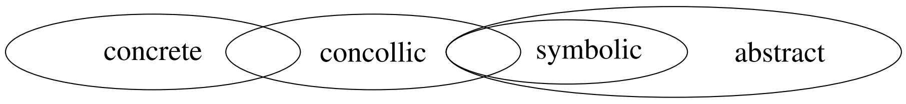


Figure 2.14.: Comcolic execution model

Concolic execution is an **under-approximate form** of program analysis, because it trades losses of correctness and completeness with an increase in performance and speed. [81, p.6]

Additionally to the symbolic information, another concrete store σ_c is administered. [81, p.5]

3. Related Work

This chapter lists the related work. At the beginning, selective work done in the area of Functionality Finding and visualization in binary analysis is cited. After that, the various signature-based static and dynamic methods for detecting cryptography are discussed in the sections 3.3 and 3.4. The section at the end describes the work done with the Angr Framework.

3.1. Functionality Finding

Over the years, several different methods of searching for functionality have emerged.
One is to take a reference binary, where the functionality is known and compare it, or in a dynamical context its execution trace [84] [85] with the sample.
One such example for static functionality finding by comparison to references has been suggested by Gao *et al.* Their tool, **BinHunt** [86], uses graph isomorphism techniques, symbolic execution and theorem proving 2008 on the control flow to detect semantic differences, meaning changes of behavior. The huge challenge here is to mask out the syntactic differences occurring because of recompilation, which e.g. entails basic block reordering or the usage of different registers.
Reanimator [87] dynamically executes binaries and observes their behavior. If malicious activities are 2010 detected, they can be extracted and put into a model. The hereby obtained templates can later be used in a statical analysis. If in another binary this certain functionality is dormant, inactive or sleeping during the dynamic analysis, it can still be detected by utilizing the model.
HumIDIFy [88] is a semi-automated tool for tracking down hidden functionality in embedded device 2017 firmware. The first step of their approach is to extract the different binaries from the firmware image. The previously trained system classifies each of the executables by its functionality and labels them with a category and a degree of certainty. The preliminary examination is thereby shortened, because the binaries are already put into rough classes identifying the type of service they should provide.
Often besides well known, default or standard libraries software relies on other Open Source Software (OSS) for common features. Careless and hasty implementation of OSS, maybe with even little knowledge

2017 of its origin, can not only cause legal but also security issues. **OSSPolice** [76] is a fully automated tool for analyzing mobile apps to identify OSS license violations and the usage of known vulnerable versions of OSS. Currently, the tool is capable of scanning Android applications.

Another possibility to detect functionality is through not looking for specific operations, but rather for the characteristics they provoke in other areas. These attacks, based on non-functional characteristics, are called side-channel attacks.

2013 **CacheAudit**[1] [89] [90] examines the interaction of a binary with the cache. It does this statically and requires a cache configuration in addition to an executable.

2016 In [91], Bos *et al.* proposed their technique, which identifies cryptographic functionality by analyzing the memory access. They built a plugin for widely known dynamic binary instrumentation frameworks, allowing them to create execution traces containing additional information about the memory accesses.

The last method, mentioned here for completeness but related work regarding cryptography is described later in section 3.3, is used to hunt down distinct constants used by a specific functionality. E.g. if it is desired to detect operations where the circumference of a circle is calculated, a tool could look out for the usage of the constant π which is usually an essential part of the calculation $C = d\pi = 2r\pi$.

3.2. Visualization of Functions, Instructions and other information

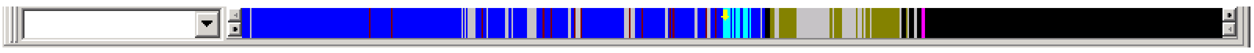
Binary analysis environments and frameworks, likes some Integrated Development Environments (IDEs) for software development, try to aid their users by visualizing the current position in the code. This is achieved by using CGs, CFGs or other images.

CFGs not only help to gain perspective during manual analysis, but allow during e.g. debugging to see upcoming branches and operations. As depicted in Figure 3.2, this representation is widely used in several binary analysis frameworks.

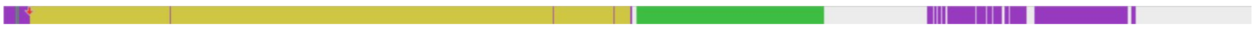
Additionally, to help navigate and indicate the position inside a binary, IDA Pro and the Hopper disassembler employ bars that colorize the different types of bytes in the assembly code. In Hopper, this navigation bar (see Figure 3.1) distinguished between code (blue), procedures (yellow), ASCII strings (green), data (purple) and undefined parts (grey).

Furthermore, the disassembler application Hopper [39] has additional graphic views. In version 4, according to the tutorial [93] on their website, these views can be found in the inspector, which is located at the farthest right part of the hopper window.

[1]`http://software.imdea.org/projects/cacheaudit/` - Last accessed: 2018.01.07



(a) IDA Pro [92]



(b) Hopper [93]

Figure 3.1.: Binary Analysis Frameworks - Navigation examples

These views depict the entropy using several techniques. This visualization of binaries is a powerful tool to identify different areas inside of a block of data. A lot of work has been done in the area of entropy visualization and several tools have shown their effectiveness.

The tool **binvis.io** [94] can interactively depict the entropy of the bytes inside a file. This can be used to immediately spot packed, encrypted or obfuscated parts of malware [2] and cryptographic materials [3], such as certificates.

Similarly, the goal of **pixd** [95] is to enable a user to quickly and easily identify the content and, to some extent, the file type of any file. This is achieved by representing each octet of data as a colorized square.

Biteye and the improved version **Vix** [96] are also colorized hexadecimal dump generators, much alike the other programs described here.

The **PortEx** [97] Java library and the based upon tool PortExAnalyzer can analyze Portable Executable (PE) binaries. One of the features it includes, besides scanning the headers and calculating hash values, is to visualize the local entropy.

..cantor.dust.. [98] [99] is a tool for interactive visual reverse engineering. It not only implements the features laid out by Cortesi, but also adds statistical analysis and Naive Bayes classification.

binglide [100] is a visual reverse engineering tool and is capable of byte, entropy and 2,3-gram visualization. It has been abandoned in favor of contributing to VELES.

Veles[4] [101] emerged out of all of the previously listed pioneering projects. It is an open source tool for helping human analysts detect patterns through statistical visualization. Veles provides an extensible design and a client server architecture. It is regularly updated and tested in Capture The Flag (CTF) competitions.

In the area of debugging and dynamical visualization, the project **Senseye** [102] uses resembling manners of visual display. It can analyze, monitor and visualize large streams of data, live memory or static files. The tool implements various visualization techniques for the entropy including histograms and 3D views. Moreover, there are views that allow the translation of the data to ASCII and even to disassembly.

[2]`https://corte.si/posts/visualisation/malware/index.html` - Last accessed: 2018.01.09

[3]`https://corte.si/posts/visualisation/entropy/index.html` - Last accessed: 2018.01.09

[4]`https://github.com/codilime/veles` - Last accessed: 2018.01.09

At an academic level [103] and [104] must be mentioned.

In [104], Conti *et al.* performed a study about different binary fragments and the byteplot representations of these fragments. In addition, they also found that visual aids help differentiate cohesive regions of data. For data in applications, they chose three categories: machine code, data structures and packed data. Furthermore, Conti *et al.* show that data can not only be organized into these different groups, but for some fragments it may even be possible to guess what functionality or representations are used. Packers or null-terminated strings leave a certain footprint that can be visualized.

The adoption and usefulness of the visual data analysis in reverse engineering has been shown in [103]. They use the visualization of bytes to find headers, footers and embedded regions. By depicting bytes, Conti *et al.* show that even though the structure itself is unknown, it is easy to distinguish fixed and variable length records.

3.3. Signature Based Cryptography Detection

This section lists the static, on data constants, signature based approaches to find cryptography in binaries.

Approaches that fall into this category use signatures, which arise from constants, structures or essential function calls, such as magic constants or S-boxes. [105, p.3]

Several publicly available tools use this techniques to discover if cryptography is present. Most of the tools have not been formally publicized, which makes paying tribute to the original source hard. However, credit was given whenever possible. More details about the tools can be found in [105] and [106].

- bfcrypt [5] [6]
- DRACA [7] [105]
- Findcrypt IDA Pro plugin [8] [105]
- Hash & Crypto Detector [9] [105]
- KANAL PEiD plugin [10] [105]

[5] `http://fwhacking.blogspot.co.at/2011/03/bfcrypt-crypto-scanner.html` - Last accessed: 2018.01.04

[6] `https://github.com/fwhacking/bfcrypt` - Last accessed: 2018.01.04

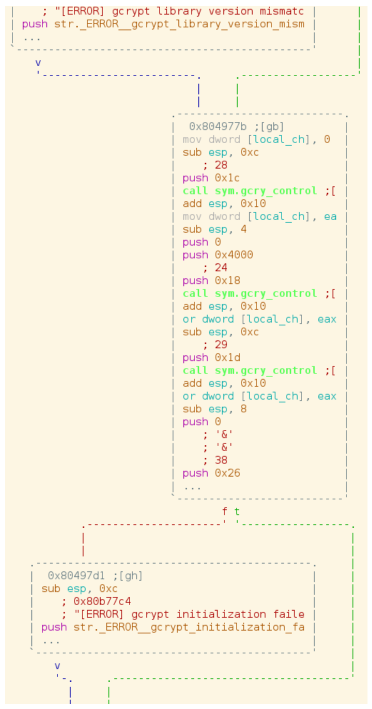
[7] `http://www.literatecode.com/draca` - Last accessed: 2018.01.04

[8] `http://www.hexblog.com/?p=27` - Last accessed: 2018.01.04

[9] `http://www.woodmann.com/collaborative/tools/index.php/Hash_%26_Crypto_Detector` - Last accessed: 2018.01.04

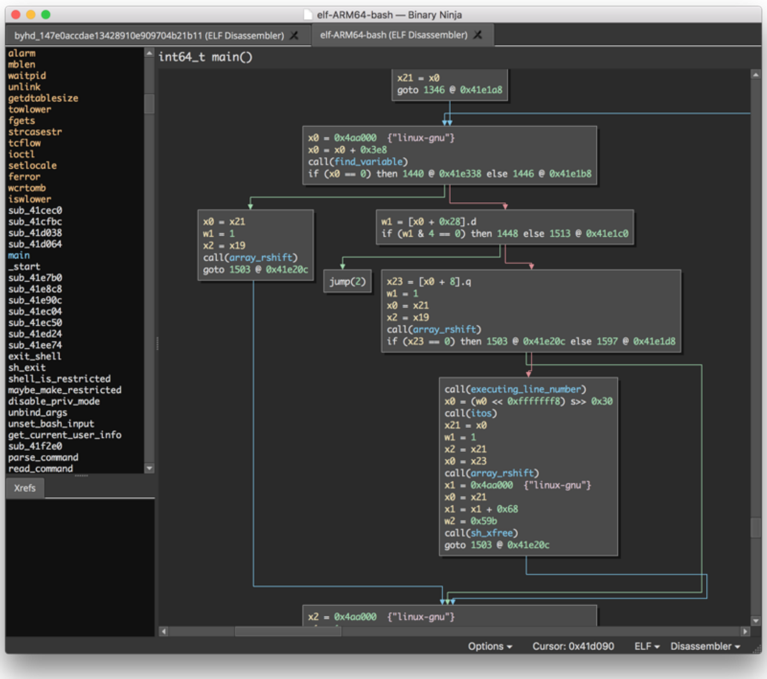
[10] `http://www.dcs.fmph.uniba.sk/zri/6.prednaska/tools/PEiD/plugins/kanal.htm` - Last accessed: 2018.01.04

(a) radare2



```
; "[ERROR] gcrypt library version mismatc
push str._ERROR__gcrypt_library_version_mism
...

0x804977b ;[gb]
mov dword [local_ch], 0
sub esp, 0xc
    ; 28
push 0x1c
call sym.gcry_control ;[
add esp, 0x10
mov dword [local_ch], ea
sub esp, 4
push 0
push 0x4000
    ; 24
push 0x18
call sym.gcry_control ;[
add esp, 0x10
or dword [local_ch], eax
sub esp, 0xc
    ; 29
push 0x1d
call sym.gcry_control ;[
add esp, 0x10
or dword [local_ch], eax
sub esp, 8
push 0
    ; '&'
    ; '&'
    ; 38
push 0x26
...

0x80497d1 ;[gh]
sub esp, 0xc
    ; 0x80b77c4
    ; "[ERROR] gcrypt initialization faile
push str._ERROR__gcrypt_initialization_fa
...
```

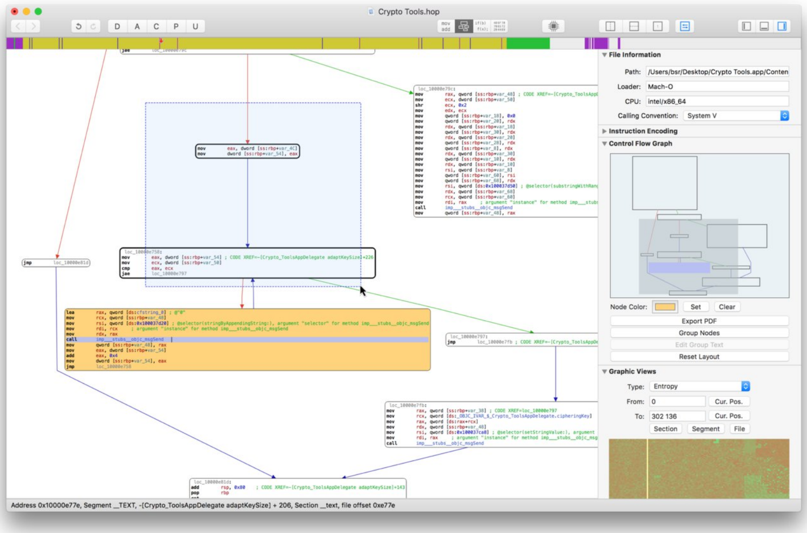
(b) Binary Ninja - Picture from [33]



elf-ARM64-bash — Binary Ninja
byhd_147e0accdae13428910e909704b21b11 (ELF Disassembler)
elf-ARM64-bash (ELF Disassembler)
Xrefs
Options
Cursor: 0x41d090
ELF
Disassembler

```
alarm
mblen
waitpid
unlink
getdtablesize
towlower
fgets
strcasestr
tcflow
ioctl
setlocale
ferror
wcrtomb
iswlower
sub_41cec0
sub_41cfbc
sub_41d038
sub_41d064
main
_start
sub_41e7b0
sub_41e8c8
sub_41e90c
sub_41ec04
sub_41ec50
sub_41ed24
sub_41ee74
exit_shell
sh_exit
shell_is_restricted
maybe_make_restricted
disable_priv_mode
unbind_args
unset_bash_input
get_current_user_info
sub_41f2e0
parse_command
read_command
```

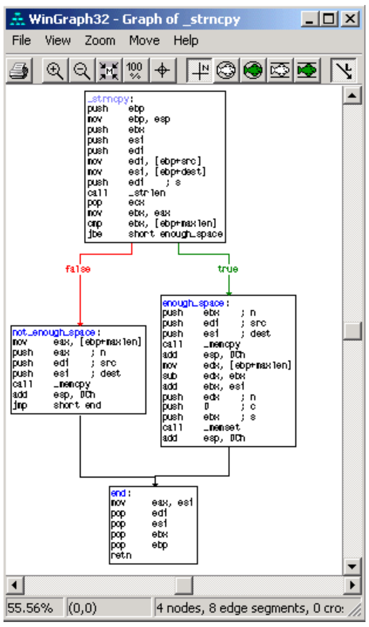
```
int64_t main()

x21 = x0
goto 1346 @ 0x41e1a8

x0 = 0x4aa000  {"linux-gnu"}
x0 = x0 + 0x3e8
call(find_variable)
if (x0 == 0) then 1440 @ 0x41e338 else 1446 @ 0x41e1b8

x0 = x21
w1 = 1
x2 = x19
call(array_rshift)
goto 1503 @ 0x41e20c

w1 = [x0 + 0x28].d
if (w1 & 4 == 0) then 1448 else 1513 @ 0x41e1c0

jump(2)

x23 = [x0 + 8].q
w1 = 1
x0 = x21
x2 = x19
call(array_rshift)
if (x23 == 0) then 1503 @ 0x41e20c else 1597 @ 0x41e1d8

call(executing_line_number)
x0 = (w0 << 0xfffffff8) s>> 0x30
call(itos)
x21 = x0
w1 = 1
x2 = x21
x0 = x23
call(array_rshift)
x1 = 0x4aa000  {"linux-gnu"}
x0 = x21
x1 = x1 + 0x68
w2 = 0x59b
call(sh_xfree)
goto 1503 @ 0x41e20c

x2 = 0x4aa000  {"linux-gnu"}
```



```
Path: /Users/bsr/Desktop/Crypto Tools.app/Conten
Loader: Mach-O
CPU: intel/x86_64
Calling Convention: System V
Type: Entropy
From: 0
To: 302 136
```

```
Address 0x10000e77e, Segment __TEXT, -[Crypto_ToolsAppDelegate adaptKeySize] + 206, Section __text, file offset 0xe77e
```

(c) Hopper - Picture from [39]



WinGraph32 - Graph of _strncpy
File
View
Zoom
Move
Help
false
true
55.56%
(0,0)
4 nodes, 8 edge segments, 0 cro:

```
_strncpy:
push    ebp
mov     ebp, esp
push    ebx
push    esi
push    edi
mov     edi, [ebp+src]
mov     esi, [ebp+dest]
push    edi      ; s
call    _strlen
pop     ecx
mov     ebx, eax
cmp     ebx, [ebp+maxlen]
jbe     short enough_space

not_enough_space:
mov     eax, [ebp+maxlen]
push    eax      ; n
push    edi      ; src
push    esi      ; dest
call    _memcpy
add     esp, 0Ch
jmp     short end

enough_space:
push    ebx      ; n
push    edi      ; src
push    esi      ; dest
call    _memcpy
add     esp, 0Ch
mov     edx, [ebp+maxlen]
sub     edx, ebx
add     ebx, esi
push    edx      ; n
push    0        ; c
push    ebx      ; s
call    _memset
add     esp, 0Ch

end:
mov     eax, esi
pop     edi
pop     esi
pop     ebx
pop     ebp
retn
```

(d) IDA Pro [92]

Figure 3.2.: Binary Analysis Frameworks - CFG examples

- Kerckhoffs [11] [105]
- Keygener Assistant [12]
- Signsrch [13] [14] [15]
- SnD Crypto Scanner OllyDbg [16] [105]
- x3chun Crypto Searcher [17] [105]

Gröbert *et al.* evaluated six of them. Their tests showed that the tools are not able to detect all cryptographic implementations and most tools have a lot of false positives. [105, p.3]

Another evaluation in [106, p.176] compared the results of existing static tools with their dynamic approach called Aligot (see section 3.4). The samples that have been used for the comparison contained TEA, RC4, AES and MD5 cryptographic functions.

2015 In 2015, a signature-based fast search for cryptographic constants was proposed in [107]. Their solution incorporates several improvements on naive search algorithms and is 5-18 times faster than other state-of-the-art solutions. Additionally, the authors integrated their approach into the AVG Retargetable Decompiler and used the results to improve the decompiled code. [107, p.5]

The signature-based approaches fail if the cryptographic constants are computed on-the-fly, are obfuscated or the encryption is done without the expected values. [107, p.5,6] If this happens, other means of identifying cryptography have to be employed.

3.4. Static and Dynamic Cryptography Detection

2008 In his work, [108] **Lutz** presented a generic tool for automatic decryption of network communication. The detection is based on the presence of loops, decrease of information entropy and heavy usage of integer arithmetics in decryption functions. [108, p.21,42] They use dynamic analysis and put the binary instrumentation framework Valgrind to work. Their assumption is that at some point, the encrypted input coming from a file or the network traffic will be decrypted. The tool developed by Lutz consist of three steps. First, dynamic information is gathered by executing the sample. The data gathered is supplied to an offline, static analysis. With the output of the previous steps, at last, the sample is executed again and the decrypted

[11] `https://github.com/felixgr/kerckhoffs` - Last accessed: 2018.01.04

[12] `http://www.woodmann.com/collaborative/tools/index.php/Keygener_Assistant` - Last accessed: 2018.01.04

[13] `https://github.com/nihilus/IDA_Signsrch` - Last accessed: 2018.01.04

[14] `http://aluigi.altervista.org/mytoolz.htm` - Last accessed: 2018.01.04

[15] `http://www.macromonkey.com/ida-signsrch-released/` - Last accessed: 2018.01.04

[16] `https://tuts4you.com/e107_plugins/download/download.php?view.1923` - Last accessed: 2018.01.04

[17] `https://tuts4you.com/e107_plugins/download/download.php?view.460` - Last accessed: 2018.01.04

message will be extracted. [108]

ReFormat is an automatic reverse engineering application focusing on encrypted messages in protocols. It requires an encrypted message and the application that decrypts it. This information is used to find the buffers that hold the decrypted message during runtime. The functionality for decryption is spotted by using the discovery, that encryption amounts to a high percentage of bitwise and arithmetic instructions compared to other processing functionality. [109, p.4,15] ReFormat is based on the Valgrind framework. [109, p.9-10] 2009

The idea of ReFormat is affirmed by [110], where the authors extracted encryption and decryption routines of a malware sample.

For the tool **Dispatcher** the authors, Caballero *et al.*, enhanced the ReFormat technique to allow multiple boundaries between encrypted and plain text data while also finding the buffer holding the unencrypted/unencoded data before the encryption. Furthermore, the detection process has been simplified and improved. [111, p.7-8] 2009

BCR is a system for automatically extracting the assembly code associated with a function. Therefore, dynamic analysis, hybrid disassembly and function extraction is used. [112, p.5,17] They tested their system by extracting decryption functions from malware, using the idea from [109] and [111], that cryptographic code has a high fraction of arithmetic and bitwise operations. [112, p.13] 2009

The authors of [113] detect the usage of cryptography by analyzing the **Input/Output (I/O) behavior** using the interfaces of the OS. Their solution collects memory, buffers and context information (filenames, network endpoints) by monitoring the according API functions (e.g.: send, WriteFile, connect, CreateFile). The location of the cryptographic actions is detected by searching for the origin (allocation) of the buffer. [113, p.2,4,5] 2009

TaintScope [114] is a checksum aware fuzzing tool, able to identify checksums and use them for automatic vulnerability detection. Similarly, **BitFuzz** [115] is also capable of detecting checksums, but discerns itself by additionally being applicable on decompression and decryption, and also using extensively symbolic execution. [115, p.11] BitFuzz uses the BitBlaze platform as foundation. [115, p.7] Both are reliant on the fact, that checksum functions highly mixes the input bytes, meaning one output byte is highly dependent on many input bytes. 2010 2010

In [116], an approach was published that uses program tracing and extracts data pattern. These patterns are the I/O behavior of a group of instruction and help with detecting and understanding cryptographic parts of binaries. [116] 2011

In [117] and [105] **Gröbert *et al.*** published improved heuristics based on generic characteristics and signatures of specific implementations. They use the instrumentation framework Pin to generate a trace and ana- 2011 lyze it applying several methods. [105, p.2] The techniques are categorized into signature-based and generic. Generic being ones that need no specific knowledge about the specific cryptographic algorithms. [105, p.9] Their tactic is based on a combination of different heuristics. The sequence of instructions that are executed is combined into a chain heuristic. This can be extended by using a combination of mnemonics and constants. These pairs include certain constant instructions, that have been found in various implementations. [105, p.10] Their final feature is called verifier heuristic, which is the relationship between input and output of permutation boxes. [105, p.10] [117, p.50f] They also implemented an evaluation of the basic blocks using the number of bitwise arithmetic instructions [117, p.44f] and loop detection [117, p.46f]. The authors claim that relying on entropy as a feature is, especially for algorithms using Cipher Block Chaining (CBC) modes, problematic, because the input of one stage is the output of its predecessor XOR the plaintext. [117, p.14]

2012 The **Crypto Intelligence System (CIS)** [118] aggregates several cryptographic detection heuristics. The authors integrated and refined the idea of a high-percentage of arithmetic and bitwise operations, calculating the entropy, monitoring for crypto APIs and analyzing the taint behavior. The detection methods have been evaluated regarding the dependency of the detection statistics (detection rate, false positives) and type of cryptographic algorithm (symmetric, asymmetric and hash).

2012 The tool set **Aligot** [18] works under the hypothesis that cryptographic functions can be identified by using the input and output parameters. Let the decryption function be $F_x(K, C) = M$, it is unlikely that another cryptographic function is identified using the I/O pair $((K, C), M)$. [106, p.170] Aligot collects the execution trace using the instrumentation framework Pin and does loop detection. Then, the data flow between the loops is deduced. The extracted flow is compared with a reference set afterwards. [106, p.170-171]

2014 The binary analysis framework **CipherXRay** relies on the avalanche effect to identify cryptographic operations. It intercepts and instruments the program and collects run-time data, focusing on taint propagation, address and number of bytes involved. From this information, an avalanche effect pattern is derived and

2015 used to find the location, size and boundary of the input, output and/or key buffer. [119, p.3] [120] The difference to TaintScope [114] or BitFuzz [115], though they have the similar idea of a high I/O dependence, is, that CipherXRay [119] can separate multiple nested or chained rounds or steps. [119, p.12]

2015 In 2015, Lestringant *et al.* presented a static identification method using **DFGs isomorphism**. Their course of action consists of building the DFG, normalizing it with rewrite-rules and searching for a subgraph (signature) in the DFG. The authors suggest, due to performance reasons, to only use their approach on code snippets opposed to a whole program. [1, p.3] They prefer the DFG over the CFG to be able to also detect

[18]https://code.google.com/p/aligot/ - Last accessed: 2018.01.05

implementations which avoid conditions and do loop unrolling. [1, p.3]

The prototype **CryptoHunt**[19] [121] uses bit precise symbolic execution and compares reference implementations with the sample. First, the binary code is executed and a trace is recorded. Then, loops are detected and the I/O relations are expressed through bit precise symbolic execution. That transforms the parameters, used as an input by the loop, to boolean variables, which are the only atomic data type. The output of the loops are expressed as boolean formulas. These are compared using guided fuzzing and a theorem prover. [121, p.4] 2017

3.5. Anti Ransomware

PayBreak [20] [122] is a cryptographic key vault implementing a key escrow mechanism. The tool can be used to automatically store keys and allows, in the case of an infection with ransomware, to restore the data. 2017

3.6. Angr Framework

Attempting to accumulate static and dynamic techniques and building an open binary analysis framework, Shoshitaishvili *et al.*, the authors of [37], developed **angr**. This framework has been used by the authors and is the foundation for several other projects and tools. 2016

In [52] the binary analysis framework **Firmalice** based on angr was presented. This framework can be used to analyze the firmware of embedded devices. [52, p.1] Their tool can be used to search for authentication bypass vulnerabilities, such as hardcoded credentials or hidden authentication interfaces in embedded software. [52, p.3] They load the firmware into their engine and use static analysis and symbolic execution. The result is checked against a security policy. [52, p.3] To overcome the limitations of symbolic execution regarding speed and performance, Firmalice searches for the points leading to privileged code and creates a backward slice. [52, p.6-7] 2015

In 2016, a paper about **Driller**, an automatic vulnerability finder using fuzzing and selective concolic execution, was published. [75, p.1] 2016

To handle path and memory explosions, the tool **WatSym** was developed. [123, p.5] It is built upon S2E for selective symbolic execution and angr in order to extract the CFG from the binary. [123, p.43] 2016

In [124], angr has been used as part of their approach to detect **BOOMERANG** vulnerabilities. BOOMERANG is a type of confused deputy vulnerability in Trusted Execution Environments (TEEs). TEEs by design have 2017

[19]`https://github.com/s3team/CryptoHunt` - Last accessed: 2018.01.04

[20]`https://github.com/BUseclab/paybreak` - Last accessed: 2018.01.04

unrestricted access to the memory, while access to the TEE from untrusted environments is prohibited. The vulnerability described here coaxes the TEE into using its elevated privileges to modify memory of the untrusted OS or other applications. [124, p.1,9]

2017 The tool **Ramblr** performs binary reassembling and therefore allows the ability to apply patches on a binary level. It depends on static analysis and angr's capability to extract the CFG. If Ramblr can not guarantee the correctness of the result, it aborts with error messages. [125, p.1,6]

2017 **ShellSwap** is a tool to replace shellcode. The replaced code still exploits the vulnerability, but performs different activities than the originally implemented ones. The main challenges in achieving this are: differentiating between the shellcode and the exploit, replacing the shellcode using non-trivial data transformations, and finally minding the dependence of the exploit and the rest of the program. [126, p.1] It uses symbolic tracing, shellcode layout remediation and path kneading. [126, p.1]

4. Analysis approach

As a part of this work, an approach for statically detecting functionality, focusing on cryptography, has been developed. It uses, combines and improves several currently available approaches, ideas and tools.
This chapter describes the said process, the implementation and frameworks it is based upon.
Let it be mentioned here, that the detection itself has to be done manually. The tool solely provides information for an analyst.

4.1. Cryptography

Cryptography can be utilized if someone (A=Alice) wants to send a message to somebody else (B=Bob) without allowing any third party or eavesdropper (E=Eve) to read it. [127, p.86-87] The original message m or **plaintext** is **enciphered** or **encrypted**, producing a coded message or **ciphertext**. The process in the opposite direction is called **deciphering** or **decryption**. [128, p.32]
Kerckhoffs' principle or **Kerckhoffs' desideratum** [130] is a list of requirements for cryptographic systems. The most interesting being, that even if the details about a specific scheme are public knowledge, the security should still be given because of the chosen key. [129, p.14] This means that although (in Figure 4.1) Eve knows the cryptographic primitives used by Bob and Alice, but not the key they use, the communication should be secure.

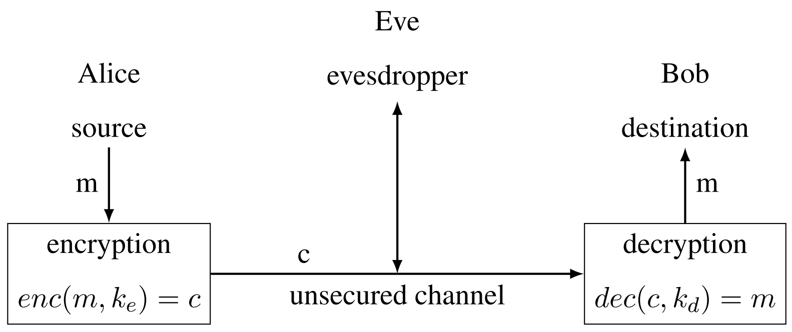


Figure 4.1.: Scheme of a communication using cryptography [129, p.13, idea from figure 1.6]

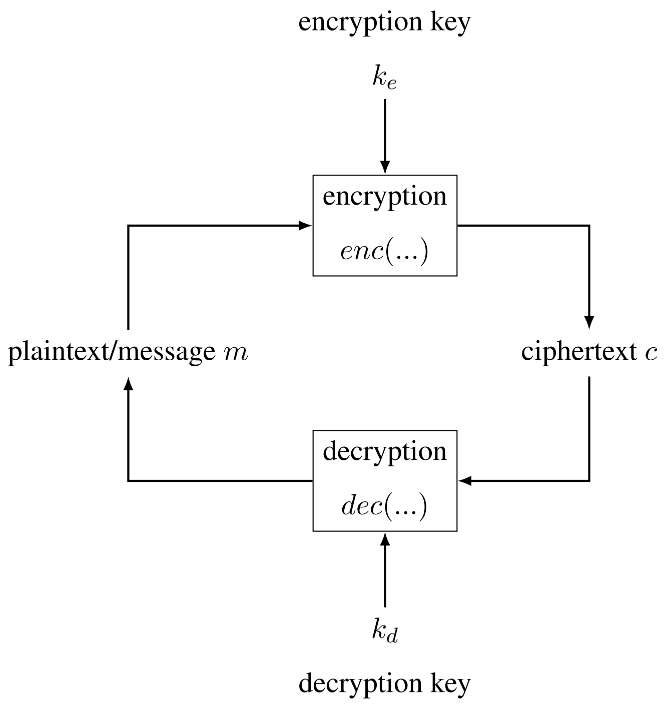


Figure 4.2.: Data flow in a cryptographic system

Symmetric encryption uses one single key for encryption as well as decryption. Symmetric ciphers are, for example, Data Encryption Standard (DES) and Advanced Encryption Standard (AES). [128, p.32]

Asymmetric cryptosystems use different keys, one key for encryption and the second matching or paired for decryption. [128, p.267]

The **avalanche effect** is a property of ciphers, being that a small change in the input invokes a big change in the output. [131, p.207f]

4.1.1. AES (Advanced Encryption Standard)

The Rijndael algorithm, specified as Advanced Encryption Standard (AES), uses 8-bit arrays as input and output. [132, p.31]

The different steps and their composition are depicted in Figure 4.3.

The AES [133] works on bytes ordered in a 4x4 square containing the data and padding at the end. The input and output matrices are in column major order (see Figure 4.4). Internally, the **State**, a two dimensional array, is used.

The Sub Bytes operation is an independent nonlinear byte substitution using a substitution table (S-box).

The Shift Rows transformation cyclically shifts the bytes in a row according to the index (starting at 0).

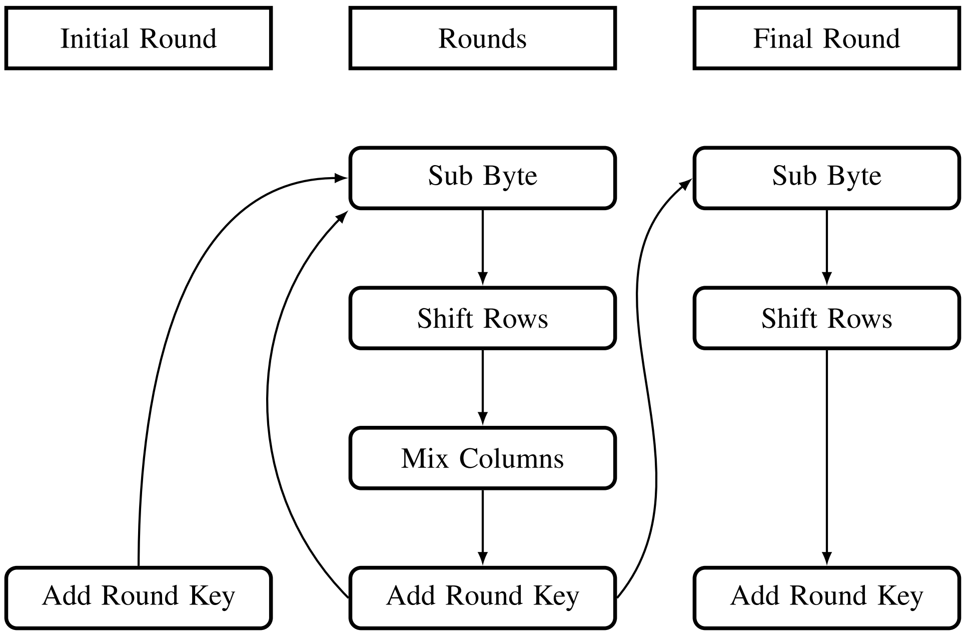


Figure 4.3.: AES - Overview functions

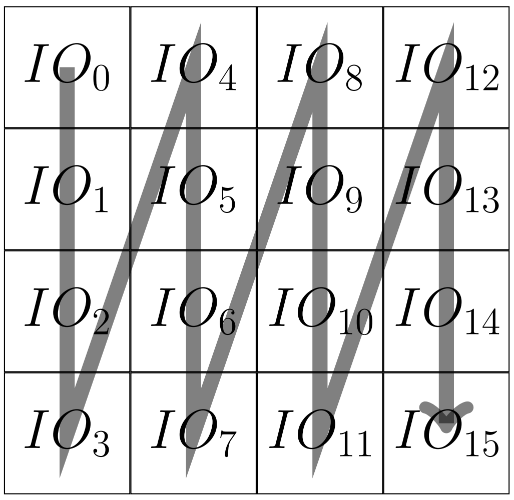


Figure 4.4.: Column-major order

The Mix Columns step works column by column and mixes up the bits.

The Add Round Key part of the process simply calculates the bitwise Exclusive OR (XOR) of the current State with the round key.

The whole AES encryption process can be broken down into three different types of rounds. The initial round consists only of the Add Round Key procedure. Hence, it only XORs the plain text and the key. After that, several regular rounds with all the previously explained subfunctions are conducted. The final round misses the Mix Column step.

The key length and the number of rounds are are specified in the NIST FIPS 197 Standard [133, p.25-26], and can either be 10, 12 or 14 (see Table 4.1).

Keysize [Bit]	**Block Number** $N_b = blocklength/32$	**Key Number** $N_k = keylength/32$	**Number of Rounds**
128	4	4	10
192	4	6	12
256	4	8	14

Table 4.1.: AES - Keysize and Rounds

4.1.2. Characteristics and Detection of Cryptography

Cryptographic functions and the resulting code have, compared to other functionality, a few to some extent unique characteristics [105, p.9] (see Figure 4.5): Usually the usage of bitwise and arithmetic instructions is higher in cryptographic functions than compared to code of other functionality. Secondly, most cryptographic algorithms make use of loops, therefore some parts are executed multiple times. Another trait is that some crypto algorithms have is their special I/O behavior. If one compares the input and output, lots of mixing [114] [115] and changes (avalanche effect) [119] can be observed.

As done in Figure 4.6, the approaches and characteristics for detecting functionality, e.g. cryptographic primitives, can be grouped by the level of abstraction or specificness used.

The more specific the level, the easier it is to correctly recognize functionality, but it also increases the chance to miss functionality that does not feature on certain traits.

The levels range from general to algorithm to implementation and an example using cryptography can be found in Figure 4.6.

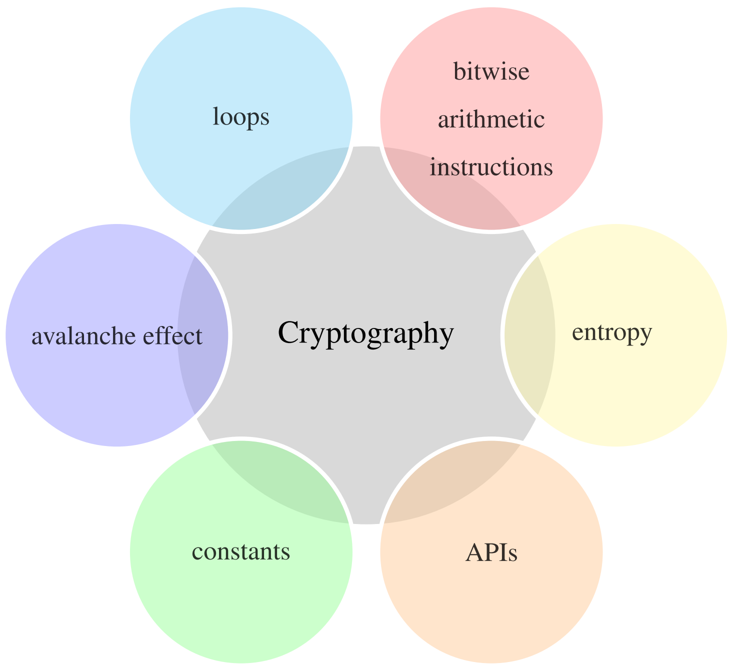
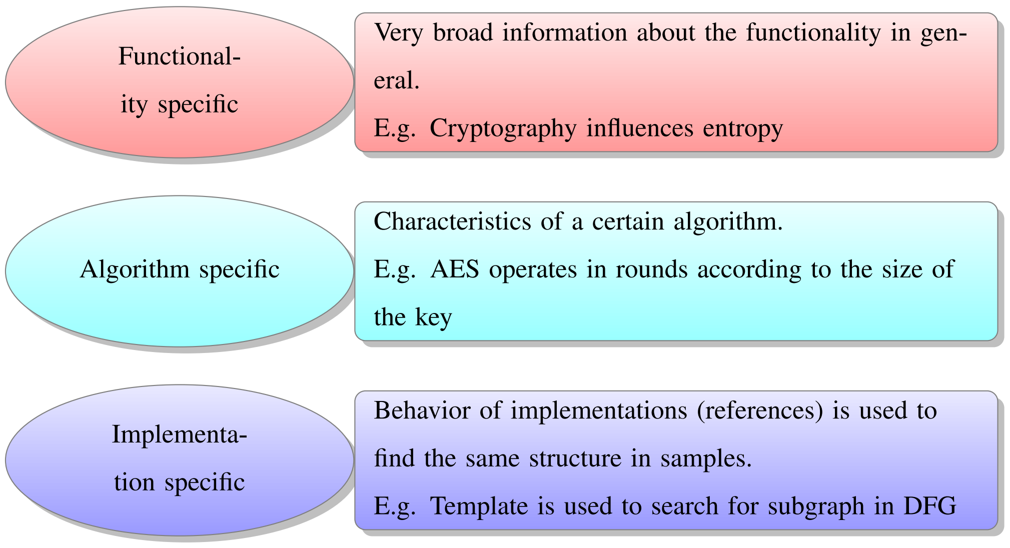


Figure 4.5.: Characteristics of cryptography



Very broad information about the functionality in general.
E.g. Cryptography influences entropy

Characteristics of a certain algorithm.
E.g. AES operates in rounds according to the size of the key

Behavior of implementations (references) is used to find the same structure in samples.
E.g. Template is used to search for subgraph in DFG

Figure 4.6.: Levels of functionality detection

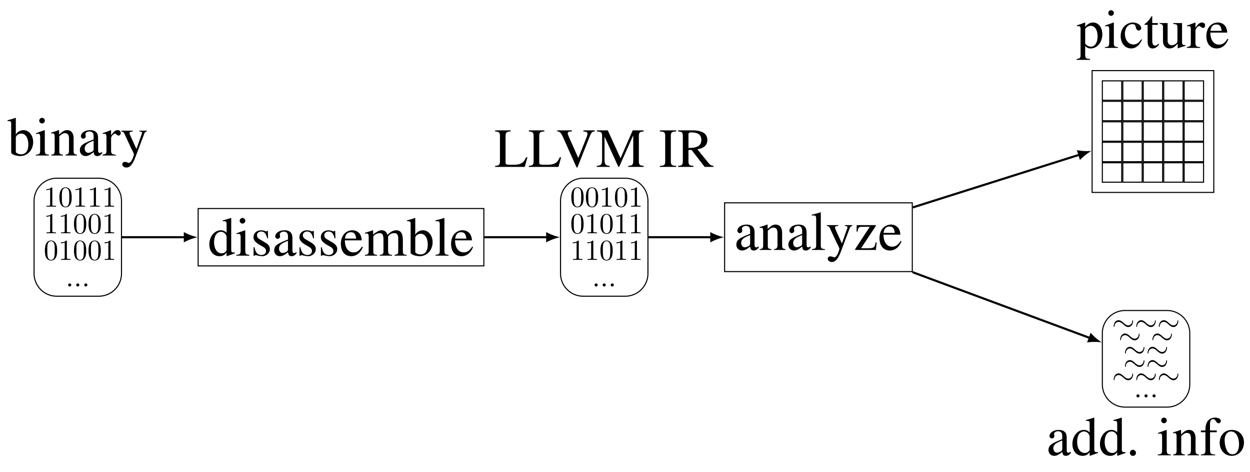


Figure 4.7.: Overview of the analysis approach

4.2. Overview of the analysis approach

Figure 4.7 depicts the approach from the binary until the result, which helps the analyst find functionality in binaries.

The input is any binary that can be analyzed by the disassembler. The output is a picture that illustrates the LLVM IR of the executable and additional information.

The process starts with disassembling the binary and gaining the LLVM IR bitcode. This code is used as an input for the detection of loops and the creation of a picture.

The LLVM IR is parsed and transformed into a picture. Each command is represented in the picture as a pixel or box. These pixel have different colors. Related commands get color codes that are close to each other.

Therefore, a high-percentage of a certain type of similar commands, like bitwise and arithmetic operations, is revealed by a cluster of resembling shades of colors.

The hope here is that visual patterns are, for the human eye, easier to spot.

Since a sheer picture can not easily give feedback to the spectator, a HTML file, mimicking a figure, is created instead. This makes it possible to also show the LLVM IR code when hovering over the boxes and display the command that is visualized.

4.2.1. Disassembly

A great number of disassemblers (see section 2.5) and Binary Analysis Frameworks, capable of disassembling, are available. They differ not only in their list of features or types of output, but also their reliability and purpose of application.

For this Proof of Concept (PoC), the RetDec framework is used for disassembling. It analyzes the binary and then outputs the disassembled LLVM IR in bitcode and textual form.

To facilitate the installation, execution and migration onto an analyst's system of choice, a Docker image, with the help of the Dockerfile in Listing 4.1, has been created.

```
FROM ubuntu:17.04
MAINTAINER none


ARG VERSION
#ENV VERSION=v0.13.x
#ENV GRADLE_VERSION 2.3

COPY entrypoint.sh /

# Install packages
RUN apt-get update && \
    apt-get upgrade --assume-yes && \
    apt-get install --assume-yes ca-certificates && \
    apt-get install --assume-yes wget && \
    apt-get install --assume-yes openssl && \
    apt-get install --assume-yes gcc && \
    apt-get install --assume-yes g++ && \
    apt-get install --assume-yes make && \
    apt-get install --assume-yes cmake && \
    apt-get install --assume-yes git && \
    apt-get install --assume-yes python3 && \
    apt-get install --assume-yes perl && \
    apt-get install --assume-yes bash && \
    apt-get install --assume-yes coreutils && \
    apt-get install --assume-yes bc && \
    apt-get install --assume-yes doxygen && \
    apt-get install --assume-yes graphviz && \
    apt-get install --assume-yes upx && \
    apt-get install --assume-yes flex && \
    apt-get install --assume-yes bison && \
    apt-get install --assume-yes zlib1g-dev && \
    apt-get install --assume-yes autoconf && \
    apt-get install --assume-yes automake && \
    apt-get install --assume-yes pkg-config && \
    apt-get install --assume-yes m4 && \
    apt-get install --assume-yes libtool && \
    apt-get install --assume-yes clang && \

# Update the CA-Certificates
    update-ca-certificates && \

# Download and install retdec tool from Github
    cd / && \
    mkdir retdec && \
    cd /retdec/ && \
    git clone --recursive https://github.com/avast-tl/retdec.git && \
    mkdir build && \
    mkdir install && \
    cd build && \
    cmake ../retdec/ -DCMAKE_INSTALL_PREFIX=/retdec/install/ && \
    make && \
    make install && \


# Clean up
    rm -rf /var/cache/apk/*

VOLUME /outside
ENTRYPOINT ["/entrypoint.sh"]
```

Listing 4.1: RetDec - Dockerfile

Moreover, containers also have the advantage of keeping the system clean and allow the execution of programs on multiple systems, without them being developed platform-independently.

4.2.2. RetDec

Retargetable Decompiler (RetDec)[1] [11, p.12] [40] is a retargetable machine code decompiler framework based on LLVM. It uses the LLVM IR as IR. Furthermore, it is not designed as a monolithic application, but instead as a chain of libraries that can be put together. The advantage of the modular design is the possibility to just use a single tool from the framework.

RetDec is structured like a typical compiler. The main big blocks are preprocessing, the core and the backend.

The preprocessing module takes the input binary file and looks for compiler and packer signatures. If necessary, the executable is unpacked.

The core basically decodes the binary into LLVM IR and optimizes it.

The backend lifts the LLVM IR up to a higher representation and does further optimization. Afterwards, it can generate the CFG, CG or translate it to High-Level Language (HLL) code, like C code.

RetDec uses a modified clone of LLVM version 3.9.1 [134].

4.2.3. LLVM IR

LLVM [135] [45] is a compiler framework. Its infrastructure is not monolithic, but rather a set of libraries or collection of compiler technologies. [45] A LLVM-based compiler implements the classical three phase

[1] `https://github.com/avast-tl/retdec` - Last accessed: 2018.01.04

compiler modules: frontend, optimizer and backend. The LLVM framework has its own Intermediate Representation (IR) that is used in the optimizer module.

The LLVM IR is a type-safe assembly language that looks like a RISC instruction set and uses the Static Single Assignment (SSA) form [136]. [47] [45]

To output the LLVM IR of C code, clang with the -emit-llvm option can be used.

```
clang -S -emit-llvm basic_operations.c
```

The code in Listing 4.3 was generated this way, with Listing 4.2 as input.

```c
double multiply(double a, double b) {
    return a*b;
}
unsigned shiftleft(unsigned b, unsigned i) {
    return b<<i;
}
```

Listing 4.2: C example code for some basic operations

```llvm
; ModuleID = 'basic_operations.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define double @multiply(double %a, double %b) #0 {
  %1 = alloca double, align 8
  %2 = alloca double, align 8
  store double %a, double* %1, align 8
  store double %b, double* %2, align 8
  %3 = load double* %1, align 8
  %4 = load double* %2, align 8
  %5 = fmul double %3, %4
  ret double %5
}

; Function Attrs: nounwind uwtable
define i32 @shiftleft(i32 %b, i32 %i) #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %b, i32* %1, align 4
  store i32 %i, i32* %2, align 4
  %3 = load i32* %1, align 4
  %4 = load i32* %2, align 4
  %5 = shl i32 %3, %4
  ret i32 %5
}

attributes #0 = { nounwind uwtable "less-precise-fpmad"="false" "no-frame-
    pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="
    false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "unsafe
    -fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

```

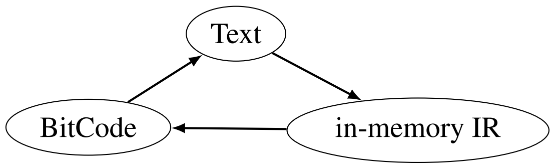


Figure 4.8.: LLVM IRs forms

```
!0 = metadata !{metadata !"Debian clang version 3.5.0-10 (tags/RELEASE_350/
    final) (based on LLVM 3.5.0)"}
```

Listing 4.3: LLVM code of Listing 4.2

LLVM IR can occur in three different shapes or representations (see Figure 4.8) that can be converted into each other.

There is the human readable text form, which is the LLVM assembly language. Then, there is the LLVM In-Memory compiler IR and the LLVM bitcode, which is a binary form that can be used for machine-based processing. [137, p.7] [47]

The tools *llvm-as* and *llvm-dis* are the LLVM assembler and disassembler and allow the transformation of the bitcode file (.bc file extension) into the human readable LLVM assembly language (.ll file extension).

Another tool llvm-bcanalyzer can print statistics about LLVM bitcode and can therefore help with understanding the structure.

LLVM bitcode files [138] (.bc files) can be identified by the first two magic numbers 'BC' or 0x42 and 0x43. The next two bytes are reserved for an application specific magic number. They are therefore only significant for application-specific applications.

The LLVM bitcode file format is a stream of bits with several simple, XML-like structures. It consist of **primitives**, **blocks**, **data records** and **abbreviations**.

The bits are read from the least- to the most-significant bit of each byte.

The stream is a sequence of unsigned integer values with either fixed or variable width. **Fixed width integer values** are simply written to the file as they are, a 2-bit wide integer value would express 1_{10} as 01_2. **Variable width integer (VBR) values** are split into chunks of a certain size. The highest bit of each chunk is used to signal if there is (1) or is not (0) another chunk afterwards. Let's take, for example, a 4-bit VBR (vbr4): The value $5_{10} = 101_2$ would be stored as 0101_2, while $42_{10} = 2A_{16} = 101010_2$ is broken up into two chunks, 1010 and 0101. The first junk equals the value 2_{10} with the first bit indicating that there is a continuation. The second chunk has no following chunk (signaled by the 0 as as the most left bit and equals to 40_{10} ($101_2 << 3_{10} = 101000$). By adding the two chunks, the original value is composed: $2_{10} + 40_{10} = 42_{10}$.

4.2.4. Arithmetic and Bitwise Instructions Detection

According to the manual [47], the LLVM IL has the following **arithmetic** and **bitwise binary** operations:

- **add** - integer addition
- **fadd** - floating point addition
- **sub** - integer subtraction
- **fsub** - floating point subtraction
- **mul** - integer multiplication
- **fmul** - floating point multiplication
- **udiv** - unsigned integer division
- **sdiv** - signed integer division
- **fdiv** - floating point division
- **urem** - unsigned integer modulo
- **srem** - signed integer modulo
- **frem** - floating point modulo
- **shl** - shift left
- **lshr** - logical shift right
- **ashr** - arithmetic shift right
- **and** - logical bitwise and
- **or** - logical bitwise or
- **xor** - logical bitwise exclusive or

Since in LLVM the concept of functions exists, two opportunities arise.
First, the information about functions can be used and for each function the number of arithmetic, bitwise and other instructions is counted separately. This information can then be listed or visualized by representing a function as a colored block, choosing the shade of the fill according to the ratio of arithmetic and bitwise to other operations.
The second option is to depict the beginning of a new function just as any other instruction. This would give even more information by also expressing the fragmentation, but may only be useful if functions with close proximity in the LLVM IR either belong to the same functionality or are near each other in the binary code.
As of now, only the latter has been implemented. This has been done, because the depiction could then also be used for non-LLVM IR code, where functions do not exist. Furthermore, the first approach can be replaced by an ordered table containing the ratio or number of arithmetic and bitwise operations compared to the other instructions.

5. Evaluation

To evaluate the proposed approach, two programs have been written. At first, a program utilizing the gcrypt library [139] for encrypting user input, using AES cryptographic routines, was built. The Listing A.7 was built statically using the Makefile in Listing A.8.

When statically linking a library during compilation, the libraries are not dynamically linked, but instead added to the code. This includes the code from the libraries into the executables and, therefore, increases the file size, but does not require the installation of the library on the target system.

Afterwards, for a second sample, henceforth referenced to by g2048, the AES implementation from `https://github.com/dhuertas/AES` by Dani Huertas has been added to the game 2048 from `https://github.com/mevdschee/2048.c` by Maurits van der Schee.

After compilation, the binaries were disassembled to LLVM IR, which then builds the foundation for the further analysis and the process of picture generation.

The LLVM IR Code was sequentially scanned and each operation was depicted as a block with a different background color. Different instructions were given different colors, but resembling colors are used for similar instructions, e.g. shades of red and orange symbolize arithmetic operations (add, sub, etc.) or violet and pink illustrate bitwise instructions (xor, and, etc.).

At first, every possible command was colorized. This proved to be confusing for the small g2048 sample, while yielding a greater outcome for the gcrypt sample.

For the g2048 sample, this meant that the visualization had to be simplified, and focused on arithmetic and bitwise operations. When only coloring arithmetic, bitwise operations and the beginnings and ends of functions, the clusters with high occurrence of those can be spotted a little easier. Although, even after changing the color scheme, it is still hard to visually find AES functionality in this sample.

In Figure 5.1 some hotspots of arithmetic and bitwise operations can be found, but it is not easy to distinguish between cryptographic and non-cryptographic operations. The functions containing functionality for encryption are highlighted green. The ones used for decryption (inverse functions) are covered in blue, and those responsible for both, or operations on the key, are red. The non-highlighted domains contain the rest

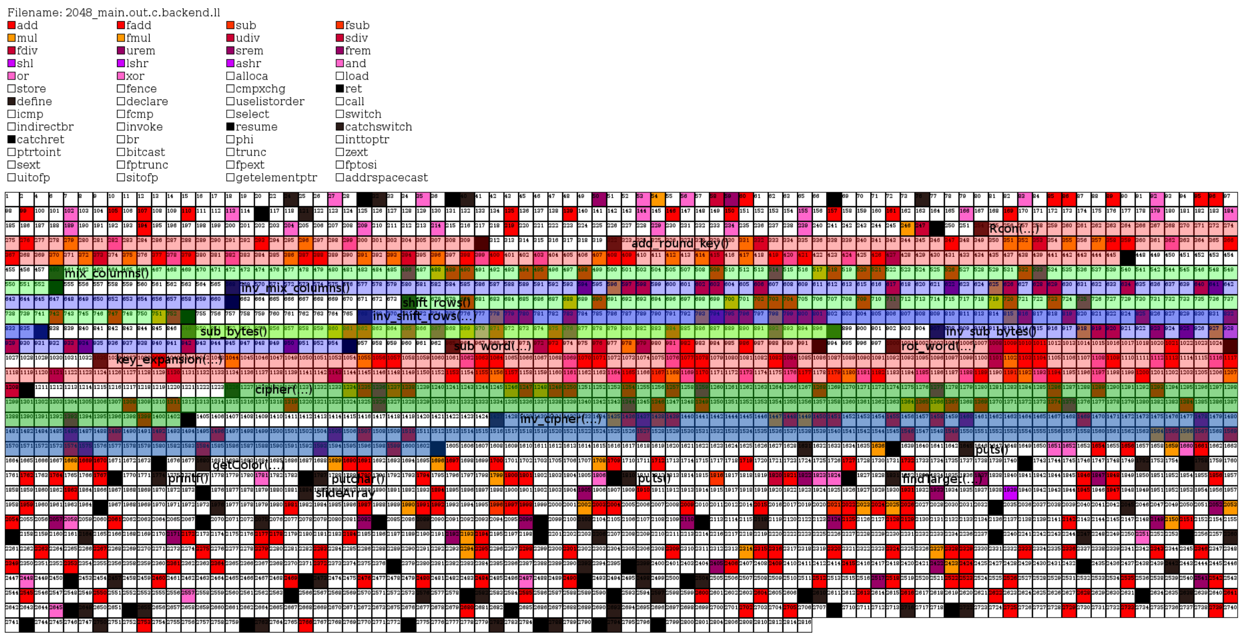


Figure 5.1.: AES Visualization - 2048 sample

of the program.

For this case, the visualization does not promise outstanding results, whereas the absolute values in Table 5.1 give greater indications to what areas compose algorithmic routines.

function	arithmetic and bitwise instructions	other instructions
key_expansion(...)	58	79
add_round_key()	45	55
cipher()	35	112
inv_cipher()	35	114

Table 5.1.: Comparison arithmetic, bitwise and other instructions - 2048 sample

Table 5.1 comprises the top four functions, when ordering all functions by the absolute count of arithmetic and bitwise instructions. Those functions are all a part of the AES encryption or decryption process.

Libgcrypt [139] is a crypto library providing several cryptographic algorithms and implementations. It features several symmetric ciphers (AES, DES, Serpent, ...), cipher modes (Electronic CodeBook (ECB), CBC, ...), public key algorithms (Rivest-Shamir-Adleman (RSA), Elliptic-Curve Diffie-Hellman (ECDH), ...) and hash algorithms (Secure Hash Algorithm (SHA), RACE Integrity Primitives Evaluation Message

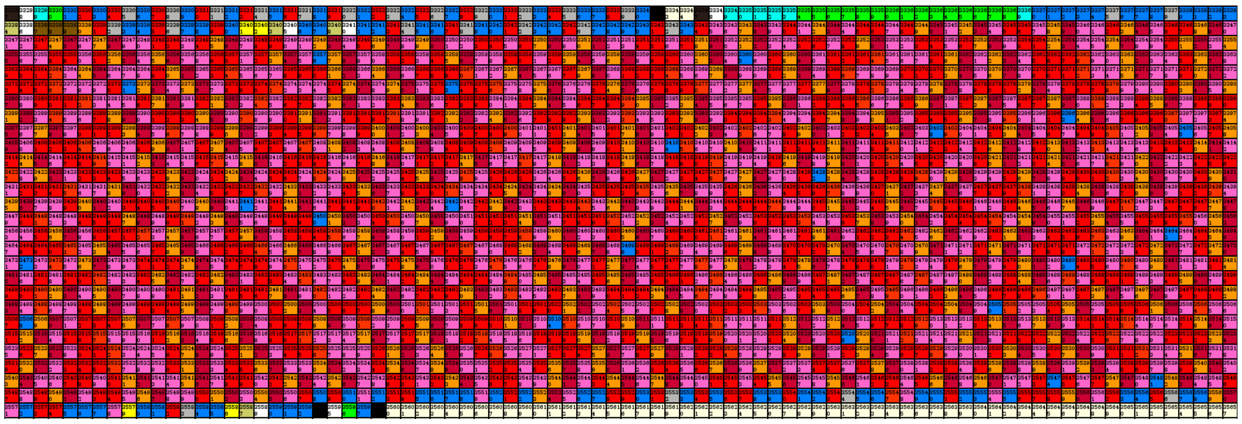


Figure 5.2.: RIPEMD-160 - Visualization

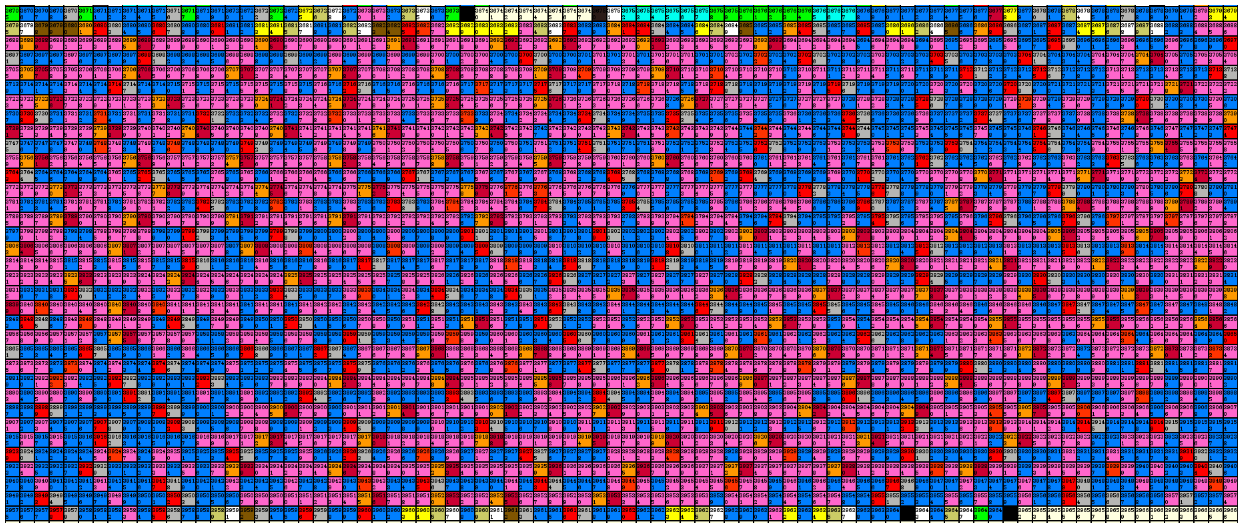


Figure 5.3.: Serpent - Visualization

Digest (RIPEMD), Whirlpool, ...).

The picture generated from the sample using libgcrypt is a good example for showing which types of cryptography can be detected using this approach.

Some cryptographic functionality can, by visual means, quite easily be detected. Browsing through the graphic depiction, there are several very distinctive patterns that show a huge occurrence of arithmetic and bitwise instructions.

The well working examples are shown in the following figures: Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5. When looking at the representation of code, these areas stand out and can therefore be quickly discovered.

Figure 5.4 depicts part of the SHA1 algorithm.

In Figure 5.5, it is possible to spot the Whirlpool hash algorithm.

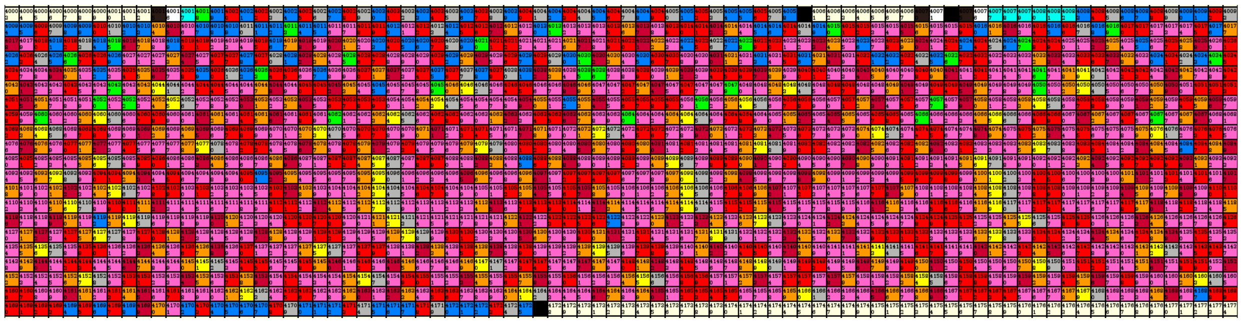


Figure 5.4.: SHA1 - Visualization

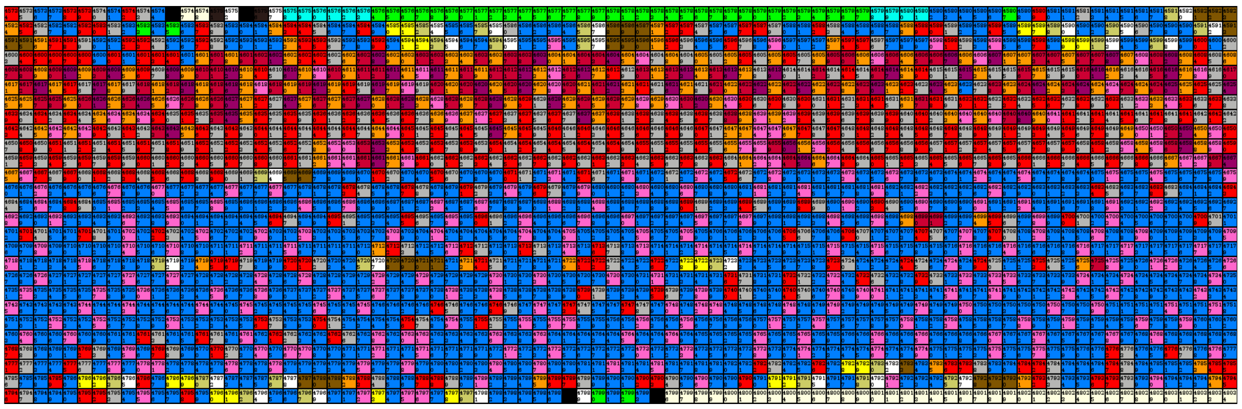


Figure 5.5.: Whirlpool - Visualization

These visual findings are also backed up by the count of instructions in Table 5.2.

function	arithmetic and bitwise instructions	other instructions
transform.13(...)	2097	63
serpent_setkey_internal(...)	1654	502
_transform()	1369	65
whirlpool_transform(...)	885	576

Table 5.2.: Comparison arithmetic, bitwise and other instructions - libgcrypt sample

The top four functions ordered by absolute occurrence of arithmetic and bitwise operations are: transform.13(...) is part of the function rmd160_final that is one unit of the RIPEMD-160 hash function. serpent_set_key_internal(...) initialized the context of the serpent cipher with a key. _transform() is part of the sha1_final routine. The function whirlpool_transform(...) is part of the Whirlpool hash function.
When ordering by the ratio of arithmetic and bitwise operations compared to all instructions, as in Table 5.3, the top four functions are slightly different.

function	arithmetic and bitwise instructions	other instructions	ratio [%]
transform.13(...)	2097	63	97.08
_transform()	1369	65	95.47
__umoddi3(...)	37	8	82.22
mul_inv(...)	20	5	80.00

Table 5.3.: Comparison arithmetic, bitwise and other instructions - libgcrypt sample

__umoddi3(...) is part of the function _gpgrt_estream_format(...).
The mul_inv(...) helper function is used by the International Data Encryption Algorithm (IDEA) cipher to invert the key.
The functions with a smaller number of overall instructions can not easily be visually picked out. This is also partly due to the technique used to convert the linear sequence of operations into a two dimensional picture. The first instruction in a line is close to the next instruction to its right, but also to the instruction below, which can be, depending on the width, quite far away.

6. Conclusion

This final chapter concludes and outlines future work.

Functionality can be detected by searching for the associated characteristics. It is essential to know, that there are always several levels on which certain functionality can be found. They reach from a very low, specific level, when searching for a particular implementation, up to very broad and generic characteristics. In the case of cryptography, it has become explicit over the years that the hight percentage of arithmetic and binary operations is a good feature for detection. Also, specific constants, such as magic boxes, can give an indication of which type or implementation of cryptographic routines are being used.

For an analyst, a picture can not only give overview and be an orientation aid, but allows, with little effort, the identification of functionality though visual pattern matching. Therefore, by simply looking at a picture, areas interesting for further analysis can be rapidly spotted.

In the evaluation, it has been shown that a quick glance at generated images can point an analyst to the areas full of instructions consisting of cryptographic functionality. It was also shown that the current visualization is not ideal for certain implementations or algorithms, while working great with others. In particular, visually identifying cryptographic hash procedures, such as whirlpool or SHA1, and ciphers, such as Serpent, proved fruitful.

6.1. Future Work

The visual depiction of binaries and IR codes can assist an investigator in his or her examination of a binary. As revealed during the evaluation, the current zig-zag line-up is not as intuitive as other ways of depiction. The visualization can therefore be improved by providing different space filling curves, such as the Hilbert curve [140]. This would enhance the proximity of the instructions in the picture, meaning that instructions that are in the one-dimensional list closer to each other will also be closer in the two-dimensional depiction. Using curves is but only one possibility to improve the proximity of code. It could also prove useful to build a graph, e.g. CFG and use the colorization of instruction to visualize functionality.

Another idea for simplifying or automating the process is to use machine learning or other means of image detection and recognition on the pictures and information gained through the analysis. Furthermore, it could also be interesting to employ a concolic or symbolic execution engine and explore paths that lead to the identified cryptographic spots.
In addition, a bigger evaluation, featuring different cryptographic implementations and visually comparing them, has to be carried out.
An evaluation regarding the applicability of this approach to detect other functionality still has to be performed as future work.

A. Source Code

A.1. CFG Generation

For plotting the CFG of examples, like the compiled code of Listing A.5, and generating figures like Figure 2.9, the angr framework has been used. This code for can be seen in Listing A.1.

```
#!/usr/bin/python
import angr
import sys
import pkg_resources #from setuptools

from angrutils import plot_cfg
#https://github.com/axt/angr-utils
#https://github.com/axt/bingraphvis


if __name__ == "__main__":

    strFile = "loop.elf"
    #strFile = "if.elf"

    # Print the version of the packages used;
    print("python version: "+sys.version.replace("\n",""))
    print("angr version: "+pkg_resources.require("angr")[0].version)
    print("angr-utils version: "+pkg_resources.require("angr-utils")[0].
        version)
    print("")


    # Load the binary without the shared libraries;
    oProj = angr.Project("./"+strFile, load_options={'auto_load_libs': False})

    print("File: "+oProj.filename)
    print("Architecture: "+oProj.arch.name)
    print("Endness: Mem="+oProj.arch.memory_endness+" Reg="+oProj.arch.
        register_endness)



    print("-> Generating CFG")

    oEntry = oProj.entry
    oMain = oProj.loader.main_bin.get_symbol("main").addr
    oMainState = oProj.factory.blank_state(addr=oMain)

    print("main address: "+hex(oMain))

    #oMain = oEntry

    oCfg = oProj.analyses.CFGAccurate(keep_state=True, starts=[oMain],
        initial_state=oMainState)

    # Use angr-utils to plot the CFG;
    plot_cfg(oCfg, strFile.replace(".","_")+"_cfg", format="png", asminst=True
        , remove_imports=True,          remove_path_terminator=True)
```

Listing A.1: CFG Generation using angr framework

```
# Dump all graphs using gcc.

# Configuration
CC=gcc

# Variables
strIn=$1
strOriginalext=".c"
strOut="${strIn/$strOriginalext/.out}"


# fdump all graphs
echo "DUMPING GRAPHS"
${CC} -lm -fdump-tree-all-graph -o $strOut $strIn

# Convert the .dot files to png
for f in $strIn*.dot
do
    echo "CONVERTING $f"
    dot -Tpng $f -o "${f/.dot/.png}"
done
```

Listing A.2: CFG Generation using gcc

```
# Makefile

# Shell
SHELL = /bin/sh

# Suffixes
# .out is used for executables
.SUFFIXES:
.SUFFIXES: .c .out

# Compiler
CC = gcc

# Other programs
# redefine rm to prompt before every removal
RM = rm -i

# Compiler flags
# -Wall ... show compiler warnings
CFLAGS = -Wall

# Linker flags
# -lm   ... math
LDFLAGS =
LDFLAGS_MATH = -lm

# Directories
DIR_SRC = .
DIR_BUILD = build

# Building targets
TARGETS = $(DIR_BUILD)/csumprod.out $(DIR_BUILD)/csumprod_sliced.out $(
    DIR_BUILD)/xorblockchiper.out
TARGETS_MATH = $(DIR_BUILD)/quadformula.out $(DIR_BUILD)/pythagoreantheorem.
    out $(DIR_BUILD)/surcone.out

.PHONY: all clean

all: $(TARGETS) $(TARGETS_MATH)

# Math-targets have different flags
$(TARGETS_MATH): LDFLAGS := $(LDFLAGS_MATH)

$(DIR_BUILD)/%.out: %.c
	$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $<

clean:
	$(RM) $(DIR_BUILD)/*
```

Listing A.3: Makefile used for building the sample C programs in section A.2

A.2. Examples

The samples have been compiled using the setup described in Appendix B and using the Makefile from Listing A.3.

```
include <stdio.h>
int main(void){
    int i=0;

    do{
        printf("%d\n",i);
        i++;
    }while(i<10);

    return 0;
}
```

Listing A.4: C loop example

```
int main(int agrc, char *argv[]){
    if(agrc>1){
        return 0;
    }
    else{
        return 1;
    }
}
```

Listing A.5: C if example

```c
/*************************************************\
 * Title: Pythagorean Theorem
 * Author: Patrick KOCHBERGER
 *
 * Description:
 * Calculates the hypotenuse from the two arguments
 * passed to the program, treating them as the
 * catheti (legs).
\*************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>


int main(int argc, char **argv){

    // Check if the usage is correct;
    if(argc!=3){
        printf("USAGE: %s a b\n", argv[0]);
        return 1;
    }
    // Define the variables and get the arguments;
    double a = atof(argv[1]);
    double b = atof(argv[2]);
    double c;

    // Calculate the hypotenuse;
    c = sqrt(a*a+b*b);

    // Output the sides;
    printf("a=%f\nb=%f\nc=%f", a, b, c);
    printf("\n");

    return 0;
}
```

Listing A.6: C implementation of the Pythagorean theorem

A.3. Cryptography examples

```c
/*************************************************\
 * Title: Libgcrypt sample
 * Author: Patrick KOCHBERGER
 *
 * Description:
 * Example using the libgcrypt for AES encrypting
 * the input "Attack at Dawn!!".
\*************************************************/

// Includes;
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <unistd.h>
```

```

#include <gcrypt.h>


void printerr(const char* msg)
{
    fprintf(stderr, "%s\n", msg);
    exit(1);
}

void printHex(const unsigned char *str, size_t size){
    size_t i;
    for(i=0;i<size;i++){
        printf("%02X ",str[i]);
    }
}
void printStrHex(const char *str){
    printHex((unsigned char *)str,strlen(str));
}

int main(void){

    if(!gcry_check_version(GCRYPT_VERSION))
    {
        printf("[ERROR] gcrypt library version mismatch.");
    }

    gcry_error_t err = 0;

    // suppress warnings
    err = gcry_control(GCRYCTL_SUSPEND_SECMEM_WARN);

    // allocate 16k secure memory
    err |= gcry_control(GCRYCTL_INIT_SECMEM, 16384, 0);

    // re-enable warnings
    err |= gcry_control(GCRYCTL_RESUME_SECMEM_WARN);

    // finish initialization
    err |= gcry_control(GCRYCTL_INITIALIZATION_FINISHED, 0);

    if(err){
        printf("[ERROR] gcrypt initialization failed.");
    }

    const int GCRY_CIPHER = GCRY_CIPHER_AES128;
    const int GCRY_C_MODE = GCRY_CIPHER_MODE_ECB;

    gcry_error_t gcryError;
    gcry_cipher_hd_t gcryCipherHd;
    char *strAesKey = "16 bytes key ...";
    char *strInitVector = "16 bytes IV. ...";
    char *strCleartxt = "Attack at Dawn!!";
    size_t sizeLength = strlen(strCleartxt);
    char *strCiphertxt = malloc(sizeLength+1); // +1 => termination '\0'
    size_t sizeKey = gcry_cipher_get_algo_keylen(GCRY_CIPHER);
    size_t sizeBlk = gcry_cipher_get_algo_blklen(GCRY_CIPHER);

    printf("AES-Key (len=%d): %s\n", (int)strlen(strAesKey),strAesKey);
    printf("Initialization Vector (len=%d): %s\n", (int)strlen(strInitVector),
         strInitVector);
    printf("Cleartext (len=%d): %s\n", (int)strlen(strCleartxt), strCleartxt);

    gcryError = gcry_cipher_open(
        &gcryCipherHd, // gcry_cipher_hd_t *
        GCRY_CIPHER,   // int
        GCRY_C_MODE,   // int
        0);            // unsigned int
    if (gcryError)
    {
        printf("[ERROR] gcry_cipher_open failed: %s/%s\n", gcry_strsource(
            gcryError), gcry_strerror(gcryError));
        return -1;
    }
    //printf("[OK] gcry_cipher_open finished.\n");


    gcryError = gcry_cipher_setkey(gcryCipherHd, strAesKey, sizeKey);
    if (gcryError)
    {
        printf("[ERROR] gcry_cipher_setkey failed: %s/%s\n", gcry_strsource(
            gcryError), gcry_strerror(gcryError));
        return -1;
    }

    gcryError = gcry_cipher_setiv(gcryCipherHd, strInitVector, sizeBlk);
    if (gcryError)
    {
        printf("[ERROR] gcry_cipher_setiv failed: %s/%s\n", gcry_strsource(
            gcryError), gcry_strerror(gcryError));
        return -1;
    }

    gcryError = gcry_cipher_encrypt(
        gcryCipherHd, // gcry_cipher_hd_t
        strCiphertxt,    // void *
        sizeLength,     // size_t
        strCleartxt,     // const void *
        sizeLength);    // size_t
    if (gcryError)
    {
        printf("[ERROR] gcry_cipher_encrypt failed: %s/%s\n", gcry_strsource(
            gcryError), gcry_strerror(gcryError));
        return -1;
    }

    printf("Ciphertext (len=%d): ", (int)strlen(strCiphertxt));
    printStrHex(strCiphertxt);
    printf("\n");

    gcry_cipher_close(gcryCipherHd);
    free(strCiphertxt);

    return 0;
}
```

Listing A.7: Libgcrypt c for sample

```
CC = gcc
```

```
CFLAGS = -Wall
LDFLAGS = -lgcrypt
RM = rm
RMFLAGS = -i

TARGET = aes_gcrypt_static_32.out

.PHONY: clean

all: $(TARGET)

clean:
	$(RM) $(RMFLAGS) $(TARGET)

aes_gcrypt_static_32.out: main.c
	$(CC) $(CFLAGS) -m32 -c main.c -o aes_gcrypt_static_32.o
	$(CC) $(CFLAGS) -m32 aes_gcrypt_static_32.o /usr/lib/i386-linux-gnu/
		libgcrypt.a /usr/lib/i386-linux-gnu/libgpg-error.a -o
		aes_gcrypt_static_32.out
```

Listing A.8: Makefile for the libgcrypt sample

B. Setup

The examples in this document have been compiled and executed using the setup described in Table B.1 and Table B.2.

Software	**Version**
Debian	8.7
gcc	4.9.2 (Debian 4.9.2-10)
python	2.7.9 (default, Jun 29 2016, 13:08:31) [GCC 4.9.2]
angr	6.7.1.31
angrutils	0.2.2
pdfTeX	This is pdfTeX, Version 3.14159265-2.6-1.40.15 (TeX Live 2015/dev/Debian) kpathsea version 6.2.1dev

Table B.1.: Setup - Software

CPU Attribute	Value
vendor_id	GenuineIntel
cpu family	6
model	58
model name	Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz
stepping	9
microcode	0x12
cpu cores	4
fpu	yes
fpu_exception	yes
cpuid level	13
wp	yes
flags	fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm ida arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase smep erms
bogomips	6385.86
clflush size	64
cache_alignment	64
address sizes	36 bits physical, 48 bits virtual

Table B.2.: Setup - CPU

List of Figures

List of Tables

Listings

Bibliography

[1] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque, "Automated identification of cryptographic primitives in binary code with data flow graph isomorphism," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15, ACM, 2015, pp. 203–214, ISBN: 978-1-4503-3245-3. DOI: `10.1145/2714576.2714639`. [Online]. Available: `http://doi.acm.org/10.1145/2714576.2714639`.

[2] Kujawa Adam, Reed Thomas, Orozco Armando, Collier Nathan, Segura Jérôme, McNeil Adam, Tsing William, and Boyd Christopher, "Cybercrime tactics and techniques Q1 2017," Malwarebytes, Tech. Rep., 2017. [Online]. Available: `https://www.malwarebytes.com/pdf/labs/Cybercrime-Tactics-and-Techniques-Q1-2017.pdf`.

[3] Tai Phoebe, Segura Jérôme, Rivero Marcelo, Reed Thomas, Orozco Armando, Collier Nathan, McNeil Adam, Tsing William, Stewart Tammy, Arntz Pieter, and Taggart Jean-Philippe, "Cybercrime tactics and techniques Q2 2017," Malwarebytes, Tech. Rep., 2017. [Online]. Available: `https://www.malwarebytes.com/pdf/white-papers/CybercrimeTacticsAndTechniques-Q2-2017.pdf`.

[4] Chandrasekar Kavitha, Cleary Gillian, Cox Orla, Lau Hon, Nahorney Benjamin, Gorman Brigid O, O'Brien Dick, Wallace Scott, Wood Paul, Wueest Candid, Aimoto Shaun, AlKhatib Tareq, Coogan Peter, Corpin Mayee, DiMaggio Jon, Doherty Stephen, Dong Tommy, Duff James, Fletcher Brian, Gossett Kevin, Groves Sara, Haley Kevin, Harnett Dermot, Johnson Martin, Kiernan Sean, Konijeti Bhavani Satish, Krall Gary, Krivo Richard, Kulkarni Yogesh, Nagel Matt, O'Gorman Gavin, Power John-Paul, Ramadass Nirmal, Sethumadhavan Rajesh, Singh Ankit, Skaar Tor, Tan Dennis, Upadhye Suyog, Vashishtha Parveen, Wright William, and Zhu Tony, "Internet security threat report," vol. 22, Apr. 2017. [Online]. Available: `https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf`.

[5] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937, ISSN: 1460-

244X. DOI: 10.1112/plms/s2-42.1.230. [Online]. Available: http://dx.doi.org/10.1112/plms/s2-42.1.230.

[6] ——, "On computable numbers, with an application to the entscheidungsproblem. a correction," *Proceedings of the London Mathematical Society*, vol. s2-43, no. 1, pp. 544–546, 1938, ISSN: 1460-244X. DOI: 10.1112/plms/s2-43.6.544. [Online]. Available: http://dx.doi.org/10.1112/plms/s2-43.6.544.

[7] J. Martin, *Introduction to Languages and the Theory of Computation*. McGraw-Hill Education, 2010, ISBN: 9780073191461. [Online]. Available: https://books.google.at/books?id=arluQAAACAAJ.

[8] Wolfgang Wögerer, "A survey of static program analysis techniques," Tech. Rep., 2005.

[9] Raimund Kirner, "On the halting problem of finite-state programs," in *14. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'07, not reviewed)*, 2007.

[10] Reynaud Daniel, *The halting problem for reverse engineers*, Dec. 19, 2010. [Online]. Available: https://indefinitestudies.org/2010/12/19/the-halting-problem-for-reverse-engineers/.

[11] J. Křoustek, "Retargetable analysis of machine code," PhD thesis, Faculty of Information Technology, Brno University of Technology, CZ, 2015, p. 190.

[12] Flemming Nielson, Hanne R. Nielson, and Chris Hankin, *Principles of program analysis*, 1st edition 1999, corrected printing 2005. Springer, 1999, ISBN: 3540654100.

[13] Nicholas Nethercote, "Dynamic binary analysis and instrumentation," PhD thesis, University of Cambridge, Nov. 2004. [Online]. Available: http://valgrind.org/docs/phd2004.pdf.

[14] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, 2nd Edition. Addison-Wesley Longman Publishing Co., Inc., 2006, ISBN: 0321486811.

[15] Kesäniemi Ari. (Nov. 2019). Llvm 3.9.1 language reference manual. Last accessed: 2018.01.04, [Online]. Available: https://www.owasp.org/images/5/53/Ari_kesaniemi_nixu_manual-vs-automatic-analysis.pdf.

[16] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi, "Bingold: towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs)," *Digital Investigation*, vol. 18, S11–S22, 2016, ISSN: 1742-2876. DOI: `http://dx.doi.org/10.1016/j.diin.2016.04.002`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S1742287616300330`.

[17] Xiaozhu Meng and Barton P. Miller, "Binary code is not easy," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, ACM, 2016, pp. 24–35, ISBN: 978-1-4503-4390-9. DOI: `10.1145/2931037.2931047`. [Online]. Available: `http://doi.acm.org/10.1145/2931037.2931047`.

[18] Juan Caballero Bayerri, "Grammar and model extraction for security applications using dynamic program binary analysis," PhD thesis, Carnegie Mellon University, 2010, ISBN: 978-1-124-96045-6.

[19] Gogul Balakrishnan and Thomas Reps, "Wysinwyx: what you see is not what you execute," *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, 23:1–23:84, Aug. 2010, ISSN: 0164-0925. DOI: `10.1145/1749608.1749612`. [Online]. Available: `http://doi.acm.org/10.1145/1749608.1749612`.

[20] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes, "Taming undefined behavior in llvm," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, ACM, 2017, pp. 633–647, ISBN: 978-1-4503-4988-8. DOI: `10.1145/3062341.3062343`. [Online]. Available: `http://doi.acm.org/10.1145/3062341.3062343`.

[21] Ken Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, Aug. 1984, ISSN: 0001-0782. DOI: `10.1145/358198.358210`. [Online]. Available: `http://doi.acm.org/10.1145/358198.358210`.

[22] Chucky Ellison and Grigore Roşu, "Defining the undefinedness of C," University of Illinois, Tech. Rep., Apr. 2012. [Online]. Available: `http://hdl.handle.net/2142/30780`.

[23] Chris Hathhorn, Chucky Ellison, and Grigore Roşu, "Defining the undefinedness of c," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, ACM, 2015, pp. 336–345, ISBN: 978-1-4503-3468-6. DOI: `10.1145/2737924.2737979`. [Online]. Available: `http://doi.acm.org/10.1145/2737924.2737979`.

[24] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama, "Towards optimization-safe systems: analyzing the impact of undefined behavior," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, ACM, 2013, pp. 260–275, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522728. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522728.

[25] Will Dietz, Peng Li, John Regehr, and Vikram Adve, "Understanding integer overflow in c/c++," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12, IEEE Press, 2012, pp. 760–770, ISBN: 978-1-4673-1067-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337313.

[26] Michael D. Ernst, "Static and dynamic analysis: synergy and duality," pp. 24–27, 2003.

[27] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, ser. SP '05, IEEE Computer Society, 2005, pp. 32–46, ISBN: 0-7695-2339-0. DOI: 10.1109/SP.2005.20. [Online]. Available: https://doi.org/10.1109/SP.2005.20.

[28] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, Dec. 2007, pp. 421–430. DOI: 10.1109/ACSAC.2007.21.

[29] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10, IEEE Computer Society, 2010, pp. 317–331, ISBN: 978-0-7695-4035-1. DOI: 10.1109/SP.2010.26. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.26.

[30] James C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976, ISSN: 0001-0782. DOI: 10.1145/360248.360252. [Online]. Available: http://doi.acm.org/10.1145/360248.360252.

[31] Cédric Valensi, "A generic approach to the definition of low-level components for multi-architecture binary analysis," PhD thesis, Université de Versailles Saint-Quentin-en-Yvelines, 2014. [Online]. Available: http://www.maqao.org/publications/theses/Thesis.CV.pdf.

[32] Wikibooks Contributors, *X86 Disassembly*. Createspace Independent Pub, 2013, ISBN: 9781466346055. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/5/53/X86_Disassembly.pdf.

[33] VECTOR 35 LLC. (2018). Binary ninja. Last accessed: 2018.01.09, [Online]. Available: `https://binary.ninja/`.

[34] Chris Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, 2011, ISBN: 9781593273958. [Online]. Available: `https://books.google.at/books?id=kOJ5G%5C_mAbZoC`.

[35] Radare2 Team, *Radare2 Book*. GitHub, 2017, Last accessed: 2017.08.30.

[36] Andy Hunt and Dave Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education, 1999, ISBN: 9780132119177. [Online]. Available: `https://books.google.at/books?id=5wBQEp6ruIAC`.

[37] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016, pp. 138–157. DOI: `10.1109/SP.2016.17`.

[38] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz, "Bap: a binary analysis platform," in *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer, Eds., Springer Berlin Heidelberg, 2011, pp. 463–469, ISBN: 978-3-642-22110-1.

[39] Cryptic Apps SARL. (2018). Hopper. Last accessed: 2018.01.09, [Online]. Available: `https://www.hopperapp.com/`.

[40] J. Křoustek, P. Matula, and P. Zemek, *Retdec: an open-source machine-code decompiler*, [talk], Presented at Botconf 2017, Montpellier, FR, Dec. 2017.

[41] Andrew W. Appel, *Modern Compiler Implementation in ML*. Cambridge University Press, 1998, ISBN: 9780521582742. [Online]. Available: `https://books.google.at/books?id=8APOYafUt-oC`.

[42] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz, "Cross-architecture bug search in binary executables," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15, IEEE Computer Society, 2015, pp. 709–724, ISBN: 978-1-4673-6949-7. DOI: `10.1109/SP.2015.49`. [Online]. Available: `http://dx.doi.org/10.1109/SP.2015.49`.

[43] R. Koschke, J.-F. Girard, and M. Würthner, “An intermediate representation for reverse engineering analyses,” in *Proceedings of the Working Conference on Reverse Engineering (WCRE’98)*, ser. WCRE ’98, IEEE Computer Society, 1998, pp. 241–, ISBN: 0-8186-8967-6. [Online]. Available: `http://dl.acm.org/citation.cfm?id=832305.837023`.

[44] Shahid Alam, R. Nigel Horspool, and Issa Traore, “Mail: malware analysis intermediate language: a step towards automating and optimizing malware detection,” in *Proceedings of the 6th International Conference on Security of Information and Networks*, ser. SIN ’13, ACM, 2013, pp. 233–240, ISBN: 978-1-4503-2498-4. DOI: `10.1145/2523514.2527006`. [Online]. Available: `http://doi.acm.org/10.1145/2523514.2527006`.

[45] A. Brown and G. Wilson, *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*, ser. The Achrictecture of Open Source Applications. CreativeCommons, 2011, vol. 1, ISBN: 9781257638017. [Online]. Available: `https://books.google.at/books?id=pgI1AwAAQBAJ`.

[46] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha, “Testing intermediate representations for binary analysis,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 353–364.

[47] LLVM Project. (Dec. 2016). Llvm 3.9.1 language reference manual. Last updated: 2018.01.04, [Online]. Available: `http://releases.llvm.org/3.9.1/docs/LangRef.html`.

[48] Shahid Alam, “Mail: malware analysis intermediate language,” University of Victoria, Tech. Rep., 2014. [Online]. Available: `http://web.uvic.ca/~salam/PhD/TR-MAIL.pdf`.

[49] Thomas Dullien and Sebastian Porst, “Reil: a platform-independent intermediate representation of disassembled code for static code analysis,” Jan. 2009.

[50] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena, “Bitblaze: a new approach to computer security via binary analysis,” in *Proceedings of the 4th International Conference on Information Systems Security*, ser. ICISS ’08, Springer-Verlag, 2008, pp. 1–25, ISBN: 978-3-540-89861-0. DOI: `10.1007/978-3-540-89862-7_1`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-540-89862-7_1`.

[51] Nicholas Nethercote and Julian Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07, ACM, 2007, pp. 89–100, ISBN: 978-1-59593-633-2.

DOI: 10.1145/1250734.1250746. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250746.

[52] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," 2015.

[53] S. Cesare and Y. Xiang, "Wire – a formal intermediate language for binary analysis," in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, Jun. 2012, pp. 515–524. DOI: 10.1109/TrustCom.2012.301.

[54] K.K.T. Thulasiraman, S. Arumugam, A. Brandstädt, and T. Nishizeki, *Handbook of Graph Theory, Combinatorial Optimization, and Algorithms*, ser. Chapman & Hall/CRC Computer and Information Science Series. CRC Press, 2016, ISBN: 9781420011074. [Online]. Available: https://books.google.at/books?id=H%5C_IYCwAAQBAJ.

[55] Frances E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970, ISSN: 0362-1340. DOI: 10.1145/390013.808479. [Online]. Available: http://doi.acm.org/10.1145/390013.808479.

[56] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare, *Data Flow Analysis: Theory and Practice*, 1st. CRC Press, Inc., 2009, ISBN: 0849328802, 9780849328800.

[57] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05, ACM, 2005, pp. 340–353, ISBN: 1-59593-226-7. DOI: 10.1145/1102120.1102165. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165.

[58] ——, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, 4:1–4:40, Nov. 2009, ISSN: 1094-9224. DOI: 10.1145/1609956.1609960. [Online]. Available: http://doi.acm.org/10.1145/1609956.1609960.

[59] Mathias Payer, Antonio Barresi, and Thomas R. Gross, "Fine-grained control-flow integrity through binary hardening," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, Eds. Springer International Publishing, 2015, pp. 144–164, ISBN: 978-3-319-20550-2. DOI: 10.1007/978-3-319-20550-2_8. [Online]. Available: https://doi.org/10.1007/978-3-319-20550-2_8.

[60] Ben Niu and Gang Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, ACM, 2014, pp. 577–587, ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594295. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594295.

[61] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, ACM, 2015, pp. 927–940, ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813673. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813673.

[62] Mingwei Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. SEC'13, USENIX Association, 2013, pp. 337–352, ISBN: 978-1-931971-03-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534796.

[63] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13, IEEE Computer Society, 2013, pp. 559–573, ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.44. [Online]. Available: http://dx.doi.org/10.1109/SP.2013.44.

[64] Mojtaba Eskandari and Sattar Hashemi, "Ecfgm: enriched control flow graph miner for unknown vicious infected code detection," *J. Comput. Virol.*, vol. 8, no. 3, pp. 99–108, Aug. 2012, ISSN: 1772-9890. DOI: 10.1007/s11416-012-0169-9. [Online]. Available: http://dx.doi.org/10.1007/s11416-012-0169-9.

[65] Parvez Faruki, Vijay Laxmi, M. S. Gaur, and P. Vinod, "Mining control flow graph as api call-grams to detect portable executable malware," in *Proceedings of the Fifth International Conference on Security of Information and Networks*, ser. SIN '12, ACM, 2012, pp. 130–137, ISBN: 978-1-4503-1668-2. DOI: 10.1145/2388576.2388594. [Online]. Available: http://doi.acm.org/10.1145/2388576.2388594.

[66] Tony C. Smith and Eibe Frank, "Statistical genomics: methods and protocols," in. Springer, 2016, ch. Introducing Machine Learning Concepts with WEKA, pp. 353–378. [Online]. Available: http://dx.doi.org/10.1007/978-1-4939-3578-9_17.

[67] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard, "Value dependence graphs: representation without taxation," in *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '94, ACM, 1994, pp. 297–310, ISBN: 0-89791-636-0. DOI: `10.1145/174675.177907`. [Online]. Available: `http://doi.acm.org/10.1145/174675.177907`.

[68] Michael D. Ernst, "Practical fine-grained static slicing of optimized code," Microsoft Research - Advanced Technology Division - Microsoft Corporation, Tech. Rep. MSR-TR-94-14, Oct. 1994.

[69] Tip Frank, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. CS-R9438, pp. 121–189, 1995.

[70] Francoise Balmas, "Displaying dependence graphs: a hierarchical approach," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, ser. WCRE '01, IEEE Computer Society, 2001, pp. 261–, ISBN: 0-7695-1303-4. [Online]. Available: `http://dl.acm.org/citation.cfm?id=832308.837144`.

[71] Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy, "Interprocedural static slicing of binary executables," in *Source Code Analysis and Manipulation*, IEEE, 2003, pp. 118–127.

[72] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, 1987.

[73] Dan Caselden, Alex Bazhanyuk, Mathias Payer, Stephen McCamant, and Dawn Song, "Hi-cfg: construction by binary analysis and application to attack polymorphism," in *Computer Security – ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings*, Jason Crampton, Sushil Jajodia, and Keith Mayes, Eds. Springer Berlin Heidelberg, 2013, pp. 164–181, ISBN: 978-3-642-40203-6. DOI: `10.1007/978-3-642-40203-6_10`. [Online]. Available: `https://doi.org/10.1007/978-3-642-40203-6_10`.

[74] Mark Weiser, "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ser. ICSE '81, San Diego, California, USA: IEEE Press, 1981, pp. 439–449, ISBN: 0-89791-146-6. [Online]. Available: `http://dl.acm.org/citation.cfm?id=800078.802557`.

[75] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna, "Driller: augmenting fuzzing through selective symbolic execution," 2016.

[76] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *CCS '17: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ACM, Oct. 2017, pp. 2169–2185.

[77] Emily R. Jacobson, Nathan Rosenblum, and Barton P. Miller, "Labeling library functions in stripped binaries," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, ser. PASTE '11, ACM, 2011, pp. 1–8, ISBN: 978-1-4503-0849-6. DOI: 10.1145/2024569.2024571. [Online]. Available: http://doi.acm.org/10.1145/2024569.2024571.

[78] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley, "BYTEWEIGHT: learning to recognize functions in binary code," in *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, 2014, pp. 845–860, ISBN: 978-1-931971-15-7. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao.

[79] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi, "Recognizing functions in binaries with neural networks," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15, USENIX Association, 2015, pp. 611–626, ISBN: 978-1-931971-232. [Online]. Available: http://dl.acm.org/citation.cfm?id=2831143.2831182.

[80] Dennis Andriesse, Asia Slowinska, and Herbert Bos, "Compiler-Agnostic Function Detection in Binaries," in *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*, IEEE, Apr. 2017.

[81] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi, "A survey of symbolic execution techniques," *CoRR*, vol. abs/1610.00502, 2016. [Online]. Available: http://arxiv.org/abs/1610.00502.

[82] Cristian Cadar, Daniel Dunbar, and Dawson Engler, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[83] Cristian Cadar and Koushik Sen, “Symbolic execution for software testing: three decades later,” *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013, ISSN: 0001-0782. DOI: `10.1145/2408776.2408795`. [Online]. Available: `http://doi.acm.org/10.1145/2408776.2408795`.

[84] Paul Leger and Éric Tanter, “A self-replication algorithm to flexibly match execution traces,” in *Proceedings of the Eleventh Workshop on Foundations of Aspect-Oriented Languages*, ser. FOAL ’12, ACM, 2012, pp. 27–32, ISBN: 978-1-4503-1099-4. DOI: `10.1145/2162010.2162019`. [Online]. Available: `http://doi.acm.org/10.1145/2162010.2162019`.

[85] Victor Sobreira, Klérisson Paixão, Sandra Amo, Ilmério Silva, and Marcelo Maia, “Using a sequence alignment algorithm to identify specific and common code from execution traces,” Tech. Rep., Nov. 2008.

[86] Debin Gao, Michael K. Reiter, and Dawn Song, “Binhunt: automatically finding semantic differences in binary programs,” in *Information and Communications Security: 10th International Conference, ICICS 2008 Birmingham, UK, October 20 - 22, 2008 Proceedings*, Liqun Chen, Mark D. Ryan, and Guilin Wang, Eds. Springer Berlin Heidelberg, 2008, pp. 238–255, ISBN: 978-3-540-88625-9. DOI: `10.1007/978-3-540-88625-9_16`. [Online]. Available: `https://doi.org/10.1007/978-3-540-88625-9_16`.

[87] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero, “Identifying dormant functionality in malware programs,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP ’10, IEEE Computer Society, 2010, pp. 61–76, ISBN: 978-0-7695-4035-1. DOI: `10.1109/SP.2010.12`. [Online]. Available: `http://dx.doi.org/10.1109/SP.2010.12`.

[88] Sam L. Thomas, Flavio D. Garcia, and Tom Chothia, “Humidify: a tool for hidden functionality detection in firmware,” in *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings*, Michalis Polychronakis and Michael Meier, Eds. Springer International Publishing, 2017, pp. 279–300, ISBN: 978-3-319-60876-1. DOI: `10.1007/978-3-319-60876-1_13`. [Online]. Available: `https://doi.org/10.1007/978-3-319-60876-1_13`.

[89] Goran Doychev, Dominik Feld, Boris Kopf, Laurent Mauborgne, and Jan Reineke, “Cacheaudit: a tool for the static analysis of cache side channels,” in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, USENIX, 2013, pp. 431–446, ISBN: 978-1-931971-03-

4. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/doychev.

[90] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke, "Cacheaudit: a tool for the static analysis of cache side channels," *ACM Trans. Inf. Syst. Secur.*, vol. 18, no. 1, 4:1–4:32, Jun. 2015, ISSN: 1094-9224. DOI: 10.1145/2756550. [Online]. Available: http://doi.acm.org/10.1145/2756550.

[91] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen, "Differential computation analysis: hiding your white-box designs is not enough," in *Cryptographic Hardware and Embedded Systems – CHES 2016: 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, Benedikt Gierlichs and Axel Y. Poschmann, Eds. Springer Berlin Heidelberg, 2016, pp. 215–236, ISBN: 978-3-662-53140-2. DOI: 10.1007/978-3-662-53140-2_11. [Online]. Available: https://doi.org/10.1007/978-3-662-53140-2_11.

[92] DataRescue. (2005). Graphing with IDA Pro. Last accessed: 2018.01.09, [Online]. Available: https://www.hex-rays.com/products/ida/support/tutorials/graphs.pdf.

[93] Cryptic Apps SARL. (2018). Hopper tutorial. Last accessed: 2018.01.09, [Online]. Available: https://www.hopperapp.com/tutorial.html.

[94] Aldo Cortesi. (2018). Binvis.io. Last accessed: 2018.01.09, [Online]. Available: http://binvis.io/#/.

[95] Jonas Höglund. (2018). Pixd. Last accessed: 2018.01.09, [Online]. Available: https://github.com/FireyFly/pixd.

[96] Gonzalo José Carracedo Carballal. (2014). Biteye & vix. Last accessed: 2018.01.09, [Online]. Available: http://actinid.org/vix/.

[97] Katja Hahn, "Robust static analysis of portable executable malware," Last accessed: 2018.01.09, Master's thesis, Leipzig University of Applied Sciences (HTWK Leipzig), 2014. [Online]. Available: https://github.com/katjahahn/PortEx.

[98] Battelle Memorial Institute. (2012). ..cantor.dust.. Last accessed: 2018.01.09, [Online]. Available: https://sites.google.com/site/xxcantorxdustxx/home.

[99] Domas Christopher. (2012). The future of re dynamic binary visualization. Last accessed: 2018.01.09, [Online]. Available: https://www.youtube.com/watch?v=4bM3Gut1hIk.

[100] Wannes Rombouts. (2017). Binglide. Last accessed: 2018.01.09, [Online]. Available: `https://github.com/wapiflapi/binglide`.

[101] codisec. (2018). VELES. Last accessed: 2018.01.09, [Online]. Available: `https://codisec.com/veles/`.

[102] Bjorn Stahl. (2018). Senseye. Last accessed: 2018.01.09, [Online]. Available: `https://github.com/letoram/senseye`.

[103] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster, "Visual reverse engineering of binary and data files," in *Proceedings of the 5th International Workshop on Visualization for Computer Security*, ser. VizSec '08, Springer-Verlag, 2008, pp. 1–17, ISBN: 978-3-540-85931-4. DOI: `10.1007/978-3-540-85933-8_1`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-540-85933-8_1`.

[104] Gregory Conti, Sergey Bratus, Anna Shubina, Andrew Lichtenberg, Roy Ragsdale, Robert Perez-Alemany, Benjamin Sangster, and Matthew Supan, "A visual study of primitive binary fragment types," Jun. 2011.

[105] Felix Gröbert, Carsten Willems, and Thorsten Holz, "Automated identification of cryptographic primitives in binary programs," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'11, Springer-Verlag, 2011, pp. 41–60, ISBN: 978-3-642-23643-3. DOI: `10.1007/978-3-642-23644-0_3`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-642-23644-0_3`.

[106] Joan Calvet, José M. Fernandez, and Jean-Yves Marion, "Aligot: cryptographic function identification in obfuscated binary programs," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12, ACM, 2012, pp. 169–182, ISBN: 978-1-4503-1651-4. DOI: `10.1145/2382196.2382217`. [Online]. Available: `http://doi.acm.org/10.1145/2382196.2382217`.

[107] Peter Matula, "Fast cryptographic constants identification in retargetable machine-code decompilation," 2015. [Online]. Available: `http://excel.fit.vutbr.cz/submissions/2015/072/72.pdf`.

[108] Noé Lutz, "Towards Revealing Attackers' Intent by Automatically Decrypting Network Traffic," Master Thesis, Eidgenössische Technische Hochschule (ETH) Zürich, 2008. [Online]. Available: `http://www.kutter-fonds.ethz.ch/App_Themes/default/datalinks/NoeLutz-08.pdf`.

[109] Zhi Wang, Xuxian Jiang, Weidong Cui, Xinyuan Wang, and Mike Grace, "Reformat: automatic reverse engineering of encrypted messages," in *Proceedings of the 14th European Conference on Research in Computer Security*, ser. ESORICS'09, Springer-Verlag, 2009, pp. 200–215, ISBN: 3-642-04443-3, 978-3-642-04443-4. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1813084.1813102`.

[110] Felix S. Leder and Peter Martini, "Ngbpa next generation botnet protocol analysis," in *Emerging Challenges for Security, Privacy and Trust: 24th IFIP TC 11 International Information Security Conference, SEC 2009, Pafos, Cyprus, May 18–20, 2009. Proceedings*, Dimitris Gritzalis and Javier Lopez, Eds. Springer Berlin Heidelberg, 2009, pp. 307–317, ISBN: 978-3-642-01244-0. DOI: `10.1007/978-3-642-01244-0_27`. [Online]. Available: `https://doi.org/10.1007/978-3-642-01244-0_27`.

[111] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song, "Dispatcher: enabling active botnet infiltration using automatic protocol reverse-engineering," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09, ACM, 2009, pp. 621–634, ISBN: 978-1-60558-894-0. DOI: `10.1145/1653662.1653737`. [Online]. Available: `http://doi.acm.org/10.1145/1653662.1653737`.

[112] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song, "Binary code extraction and interface identification for security applications," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-133, Oct. 2009. [Online]. Available: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-133.html`.

[113] F. Leder, P. Martini, and A. Wichmann, "Finding and extracting crypto routines from malware," in *2009 IEEE 28th International Performance Computing and Communications Conference*, Dec. 2009, pp. 394–401. DOI: `10.1109/PCCC.2009.5403858`.

[114] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 497–512. DOI: `10.1109/SP.2010.37`.

[115] Juan Caballero, Pongsin Poosankam, Stephen McCamant, Domagoj Babi ć, and Dawn Song, "Input generation via decomposition and re-stitching: finding bugs in malware," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ser. CCS '10, ACM, 2010, pp. 413–425, ISBN: 978-1-4503-0245-6. DOI: `10.1145/1866307.1866354`. [Online]. Available: `http://doi.acm.org/10.1145/1866307.1866354`.

[116] Ruoxu Zhao, Dawu Gu, Juanru Li, and Ran Yu, "Detection and analysis of cryptographic data inside software," in *Proceedings of the 14th International Conference on Information Security*, ser. ISC'11, Springer-Verlag, 2011, pp. 182–196, ISBN: 978-3-642-24860-3. [Online]. Available: `http://dl.acm.org/citation.cfm?id=2051002.2051020`.

[117] Felix Gröbert, "Automatic Identification of Cryptographic Primitives in Software," Diplomarbeit, Ruhr-University Bochum, Germany, 2010.

[118] Felix Matenaar, Andre Wichmann, Felix Leder, and Elmar Gerhards-Padilla, "Cis: the crypto intelligence system for automatic detection and localization of cryptographic functions in current malware," in *Proceedings of the 2012 7th International Conference on Malicious and Unwanted Software (MALWARE)*, ser. MALWARE '12, IEEE Computer Society, 2012, pp. 46–53, ISBN: 978-1-4673-4880-5. DOI: `10.1109/MALWARE.2012.6461007`. [Online]. Available: `http://dx.doi.org/10.1109/MALWARE.2012.6461007`.

[119] Xin Li, Xinyuan Wang, and Wentao Chang, "Cipherxray: exposing cryptographic operations and transient secrets from monitored binary execution," *IEEE Trans. Dependable Secur. Comput.*, vol. 11, no. 2, pp. 101–114, Mar. 2014, ISSN: 1545-5971. DOI: `10.1109/TDSC.2012.83`. [Online]. Available: `http://dx.doi.org/10.1109/TDSC.2012.83`.

[120] Abraham Anitha, Paul Dr. Varghese, and Chinju.K, "Detection of cryptographic operations in malware binaries," *International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)*, vol. 2, no. 7, Jul. 2015.

[121] D. Xu, J. Ming, and D. Wu, "Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 921–937. DOI: `10.1109/SP.2017.56`.

[122] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele, "Paybreak: defense against cryptographic ransomware," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17, ACM, 2017, pp. 599–611, ISBN: 978-1-4503-4944-4. DOI: `10.1145/3052973.3053035`. [Online]. Available: `http://doi.acm.org/10.1145/3052973.3053035`.

[123] Muhammad Riyad Parvez, "Combining Static Analysis and Targeted Symbolic Execution for Scalable Bug-finding in Application Binaries," Master's thesis, University of Waterloo, 2016.

[124] Aravind Machiry, Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna, “Boomerang: exploiting the semantic gap in trusted execution environments,” in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[125] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Machiry Aravind, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna, “Ramblr: Making Reassembly Great Again,” in *Proceedings of the 2017 Network and Distributed System Security Symposium*, 2017.

[126] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, “Your exploit is mine: automatic shellcode transplant for remote exploits,” in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 824–839. DOI: `10.1109/SP.2017.67`.

[127] Bruce Schneier, *Secrets and Lies: Digital Security in a Networked World*, 15th Anniversary Edition. John Wiley & Sons Inc, 2015, ISBN: 9781119092438.

[128] William Stallings, *Cryptography and Network Security: Principles and Practice*, 5th edition. Prentice Hall, 2011, ISBN: 9780136097044. [Online]. Available: `https://books.google.at/books?id=wwfTvrWEKVwC`.

[129] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot, *Handbook of Applied Cryptography*, 1st. CRC Press, Inc., 1996, ISBN: 0849385237. [Online]. Available: `http://cacr.uwaterloo.ca/hac/`.

[130] Auguste Kerckhoffs, “La cryptographie militaire,” *Journal des sciences militaires*, vol. IX, pp. 5–83, Jan. 1883. [Online]. Available: `http://www.petitcolas.net/fabien/kerckhoffs/`.

[131] Jonathan Katz and Yehuda Lindell, *Introduction to Modern Cryptography, Second Edition*, 2nd. CRC Press, 2014, ISBN: 1466570261, 9781466570269.

[132] Joan Daemen and Vincent Rijmen, “The advanced encryption standard process,” in *The Design of Rijndael: AES — The Advanced Encryption Standard*. Springer Berlin Heidelberg, 2002, pp. 1–8, ISBN: 978-3-662-04722-4. DOI: `10.1007/978-3-662-04722-4_1`. [Online]. Available: `http://dx.doi.org/10.1007/978-3-662-04722-4_1`.

[133] Dworkin Morris J., Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E Roback, and James F. Dray Jr., “Federal Information Processing Standards Publication 197 - Advanced Encryption Standard (AES),” National Institute of Standards and Technology (NIST), Standard, Nov. 26, 2001. DOI: `https://dx.doi.org/10.6028/NIST.FIPS.197`.

[134] LLVM Project. (Dec. 2016). Llvm 3.9.1 documentation. Last updated: 2018.01.04, [Online]. Available: `http://releases.llvm.org/3.9.1/docs/index.html`.

[135] Chris Lattner and Vikram Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.

[136] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88, ACM, 1988, pp. 12–27, ISBN: 0-89791-252-7. DOI: `10.1145/73560.73562`. [Online]. Available: `http://doi.acm.org/10.1145/73560.73562`.

[137] Criswell John. (Jan. 2016). LLVM Tutorial. Last accessed: 2018.01.03, [Online]. Available: `https://roclocality.files.wordpress.com/2016/01/04-llvm-tutorial1.pdf`.

[138] LLVM Project. (Dec. 2016). Llvm 3.9.1 bitcode file format. Last updated: 2018.01.04, [Online]. Available: `http://releases.llvm.org/3.9.1/docs/BitCodeFormat.html`.

[139] Werner Koch and Moritz Schulte, "The libgcrypt reference manual," *Free Software Foundation Inc*, 2017.

[140] David Hilbert, "Über die stetige abbildung einer linie auf ein flächenstück," in *Dritter Band: Analysis · Grundlagen der Mathematik · Physik Verschiedenes: Nebst Einer Lebensgeschichte*. Springer Berlin Heidelberg, 1935, pp. 1–2, ISBN: 978-3-662-38452-7. DOI: `10.1007/978-3-662-38452-7_1`. [Online]. Available: `https://doi.org/10.1007/978-3-662-38452-7_1`.