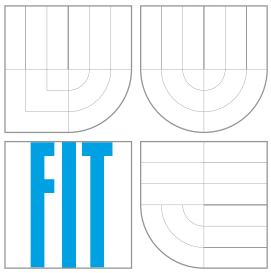
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC BEHAVIORAL MALWARE DETECTION OVER LLVM IR

STATICKÁ DETEKCE MALWARE NAD LLVM IR

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE
Bc. MAREK SUROVIČ

SUPERVISOR
VEDOUCÍ PRÁCE
Prof. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2016

Abstract

In this thesis we study methods for behavioral malware detection, which use techniques of formal verification. In particular we build on the works, which use inference of tree automata from syscall dependency graphs, obtained by static analysis of LLVM IR. We design and implement a prototype detector using the LLVM compiler framework. For experiments with the detector we use an obfuscating compiler capable of generating mutations of malware from C/C++ source code. We discuss preliminary experiments which show the capabilities of the detector and possible future extensions to the detector.

Abstrakt

Tato práce se zabývá metodami pro behaviorální detekci malware, které využívají techniky formální analýzy a verifikace. Základem je odvozování stromových automatů z grafů závislostí systémových volání, které jsou získány pomocí statické analýzy LLVM IR. V rámci práce je implementován prototyp detektoru, který využívá překladačovou infrastrukturu LLVM. Pro experimentální ověření detektoru je použit překladač jazyka C/C++, který je schopen generovat mutace malware za pomoci obfuskujících transformací. Výsledky předběžných experimentů a případná budoucí rozšíření detektoru jsou diskutovány v závěru práce.

Keywords

Behavioral malware detection, LLVM IR, Tree automata inference, Static analysis, Formal verification

Klíčová slova

Behaviorální detekce malware, LLVM IR, Odvození stromových automatů, Statická analýza, Formální verifikace

Reference

SUROVIČ, Marek. *Static Behavioral Malware Detection over LLVM IR*. Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

Static Behavioral Malware Detection over LLVM IR

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Prof. Ing. Tomáš Vojnar, PhD. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Marek Surovič
May 25, 2016

Acknowledgements

I would like to thank my supervisor, prof. Tomáš Vojnar for his advice and help and my family and friends for their support in times of need.

© Marek Surovič, 2016.
This thesis was created as a school work at the Brno University of Technology, Faculty of Information Technology. The thesis is protected by copyright law and its use without author's explicit consent is illegal, except for cases defined by law.

Contents

Chapter 1

Introduction

Information technologies and systems have found use in almost all aspects of human life. From personal communication and entertainment to industrial manufacturing, services and commerce. Because of this widespread adoption, safety and security of information systems and the data they hold have become a major concern. Throughout the years, development and administration of information systems has shown to be a complex task and errors are a common occurence. These errors may pose a safety risk to anyone using the system. Moreover, some of these errors can be abused by malicious parties to achieve goals that are not in alignment with the wishes of a legitimate user. The answer to this problem is twofold. By advancing the processes and technologies that we use to develop information systems, we lower the number of errors and weaknesses that the system has prior to its use. By carefully monitoring usage of the system after deployment, we lower the risk of abuse of errors that were not discovered and corrected during developement.

Malware is a term used to describe any software primarily designed for malicious use against a host information system. Common uses include causing damage to the host system, denying usage of the host to legitimate users, and theft of data and computing resources. Depending on its purpose and method of transfer, malware can be classified into several families. For example, a *trojan* is malware that is introduced into a host, often by a legitimate user, disguising itself as harmless software. On the other hand, a *worm* often abuses an error in the host system to gain access. Both of these are examples of malware that function as standalone pieces of software. *Viruses*, in contrast to trojans and worms, need a host file or software to propagate. The term *payload* is often used to describe the code that will carry out the intended use of the malware. A *keylogger* will log keystrokes to gather sensitive information about the systems users, while a *backdoor* will grant access to the host system to an illegitimate user.

As a means to protect information systems and user data, security systems were developed. Examples include *firewalls*, which come in the form of hardware and software or *intrusion detection* and *intrusion prevention* systems. But probably the most common security system is *antivirus* software. The former can be described as passive measures since they mitigate threats that are already being carried out by a human agent or malware. Antiviruses, on the other hand, were originally designed to protect against malware by actively scanning files present in the host system.

With the introduction of antiviruses, malware authors had to come up with ways to hide the presence of their malware in a host system. One method of achieving this is *obfuscation*. The goal of obfuscation is to make malware hard to distinguish from legitimate software by changing the malware in a way that preserves its original functionality or purpose.

Obfuscations that primarily change how the malware appears as a file, are called *syntactic*, while those which change how the malware functions while preserving its intended purpose are referred to as *semantic*.

As a reponse, antivirus developers needed to implement methods that can detect malware despite being obfuscated. As with malware, detection methods that focus on the appearance of malware as a file in a host system are syntactic, or often referred to as *signature based*, while methods that focus on the functionality of the malware are semantic, or often called *behavioral*. Historically, signature-based detection was preferred for its speed and scalability. However as obfuscation techniques evolved, the number of malware that could bypass signature-based detection increased. Today the focus has shifted towards behavioral detection.

The area of *formal analysis and verification* offers several approaches to behavioral detection. One of these approaches is *static analysis*. The idea behind static analysis is to gather data about possible behaviors of software directly from its code. In the context of behavioral detection, the gathered data can be used to decide whether the analyzed code is malware by comparing it to data gathered from known malware samples.

The goal of this work is to explore applications of static analysis to behavioral malware detection and implement a prototype detector that applies static analysis in its detection process. The work begins by giving a brief introduction to the challenges of detecting obfuscated malware in Chapter 2. In order to detect and classify malicious behavior descriptions obtained by static analysis, suitable models and methods are proposed in Chapter 3. The LLVM compiler infrastructure and its intermediate representation is also introduced in Chapter 3 as a means to implement the mentioned static analyses. The design and implementation of a detector prototype is described in Chapter 4. An experimental evaluation of the detector prototype is made in Chapter 5. Finally conclusion with a discussion about choices that were made with possible alternatives and potential for future work is given in chapter 6.

Chapter 2

Malware Detection

Early research in the field of computer virology shows that perfectly reliable malware detection is theoretically impossible [4] and more recent research shows that practical malware detection can be made computationally infeasible [8]. Despite these negative results, antivirus software is comercially successful and research in the field of malware detection is meaningful.

Since any method of malware detection is bound to be imperfect, there are several metrics that are used in order to describe the performance of a malware detection method. The number of *false positive* and *false negative* results over a test set of legitimate software and malware respectively gives insight into the detection capabilities, while simple time and memory consumption metrics give insight into the cost and potential scalability of the detection method. Another useful distinction to make is whether the detection method is capable of detecting malware that it has not seen before, but has seen similar malware in terms of syntax or semantics, depending on the type of detector. This capability is referred to as *forward detection* [3] or *generalization* [2].

This chapter aims to introduce topics that impact these metrics. From the point of view of the malware itself, various types of obfuscation will be presented and a taxonomy of malware based on obfuscation techniques will be presented. This part of the chapter is based on [25], [21] and [23] From the detection point of view, the advantages, drawbacks and overall principles behind syntactic and semantic detection will be introduced, taking from [14] and again [23].

2.1 Obfuscation and Malware Taxonomy

A crucial part of any malware today is the ability to hide itself from detection in a host system. The most commonly used technique to achieve this is obfuscation, which in te context of computer programs can be defined as a transformation aimed to hide functionality and hinder analysis. As a transformation it can be applied on any form the program takes, from the original source code, down to the binary file. Figure 2.1 shows the typical compilation chain of a C-like language with all the representations a program takes. In the case of malware the most common obfuscations are applied to the binary file and on the level of assembly code. It is interesting to note that obfuscations also find use in the field of intellectual property protection in which they protect against unwanted reverse-engineering. In this case the obfuscations are applied on the source code or the intermediate representation used by the compiler or interpreter.

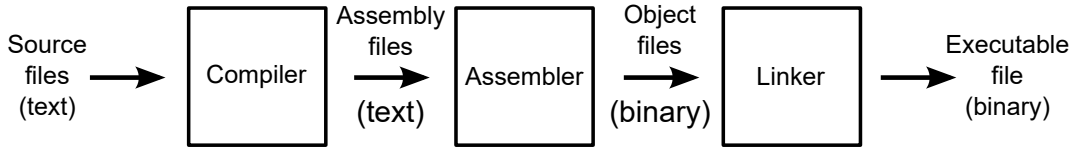


Figure 2.1: Compilation chain of a C-like language

A prominent feature when using any kind of obfuscation is that, when applied in a randomized manner, one can generate a number of different versions of the final malware. These versions are referred to as *mutations* and the quality of an obfuscation technique is often measured by the number of possible mutations it allows.

2.1.1 Packing, encryption and oligomorphism

Packing is a technique which uses compression algorithms to scramble the content of the binary file [13]. A decompression procedure is then added to the compressed malware binary so that when the executable is loaded into memory, it will first decompress the malware code and then proceed to execute it. Packing also reduces file size which helps propagation via size-limited channels e.g. email attachments.

Another common way of obfuscating binary files is through encryption. Similarily to packing, the main malware body is encrypted using a simple cipher and a decryption procedure is added to the result. The encryption key is often carried with the decryption procedure or can be easily computed from data available to the decryption procedure. Most commonly used ciphers include XOR ciphers, RC4 or TEA.

By using multiple packers or encryption procedures and keys one can obtain a sizeable number of possible mutations. Malware that generates its mutations in this manner is *oligomorphic*. Although the number of possible mutations is high, the decompression and decryption procedures themselves are not mutated. This produces an opportunity for malware analysts to create signatures that target these procedures and detect oligomorphic malware based on them.

2.1.2 Polymorphism

The solution to the shortcomings of oligomorphic malware is to mutate their decompression and decryption procedures. *Polymorphic* malware achieves this by applying obfuscations to the code of those procedures. Depending on the obfuscations applied the number of possible mutations rises dramatically. The result is malware whose mutations share almost no resemblance to each other as binary files and thus cannot be generally detected by matching against byte signatures of binary files. A list of commonly used obfuscations follows.

Dead code insertion Code with no effect is inserted into the original code. Common examples include inserting `NOP` instructions between original assembly instructions or performing `XOR` operations over two identical operands. While very simple to implement, it is also very simple to reverse the transformation and recover the orignal code. Combined with other obfuscations however, the inserted code may be further transformed and made very hard to recognize as dead.

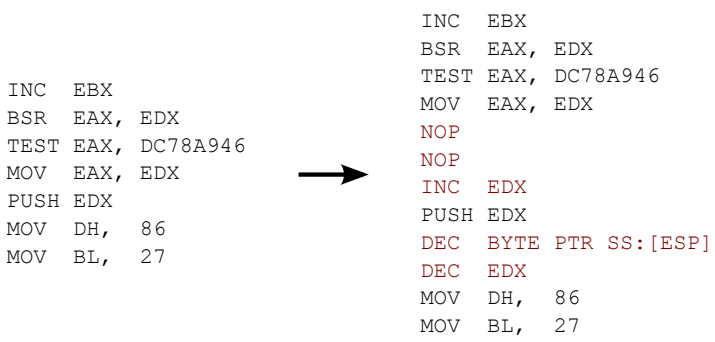


Figure 2.2: Example of dead code insertion

Register reassignment Operands of assembly code instructions are often stored in registers. The obfuscation changes the registers in which the operands of instructions are stored. The number of possible mutations this obfuscations allows can be large depending on the instructions used and number of registers available, however it can be easily circumvented via using inexact byte signatures. This technique was prominently used in the Win95/Regswap virus.

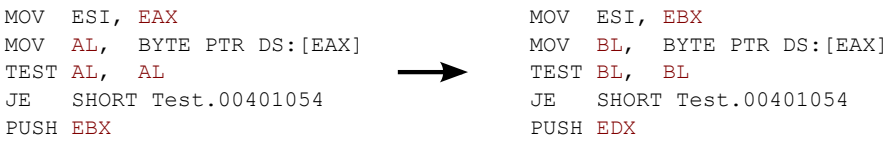


Figure 2.3: Example of register reassignment

Subroutine reordering Also called code permutation is a transformation where standalone code segments such as subroutines or basic blocks can be randomly reordered to produce up to $N!$ mutations, where N is the number of such segments. W32/Ghost is a common example of a virus that uses this obfuscation.

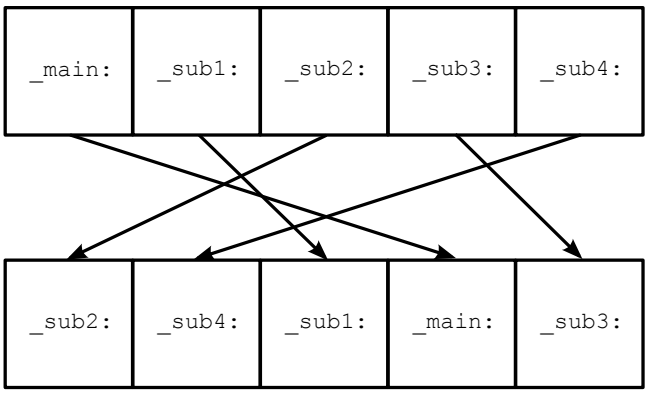


Figure 2.4: Example of subroutine reordering

Instruction substitution The effect of a single instruction in code can be often emulated via a sequence of instructions. Typical examples are simple arithmetic and boolean operations which can be emulated in a number of ways each. Other substitutions include moves between registers being replaced by `PUSH` and `POP` instructions on the assembly level.

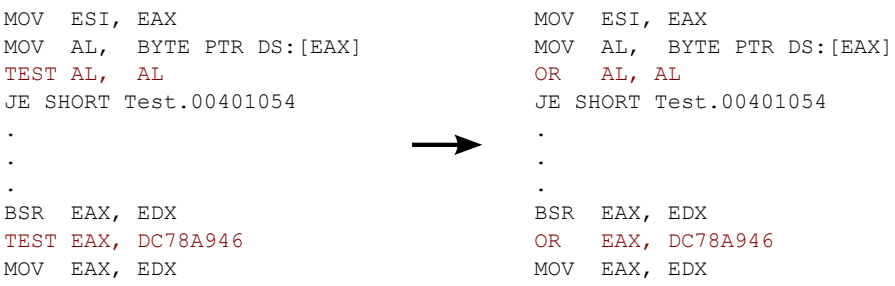


Figure 2.5: Example of instruction substitution

Code transposition Similar to subroutine reodreding, this obfuscation reorders instructions to change the binary file. If the reordered instructions are dependent, original execution order is restored via unconditional jumps. Depending on the code representation, determining whether two instructions are dependent may require a non-trivial analysis.

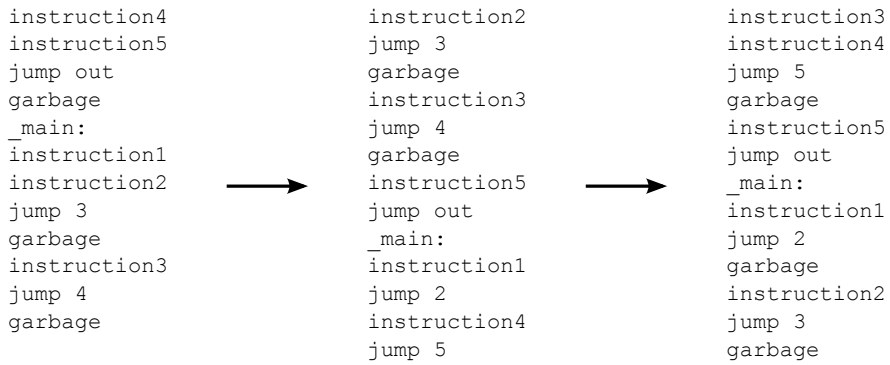


Figure 2.6: Example of code transposition

Code integration A technique that uses another executable binary file for obfuscation. The malware excutable targets a host executable, disassembles it, inserts its own code into it and reassembles the host, preserving its original functionality. This way the malware can generate as many mutations as there are executable binaries in the host system, while providing other advantages like obfuscating the malware's entry point, since the execution of malware code is interleaved with the execution of the host binary. This obfuscation was introduced in the famous W95/Zmist virus.

2.1.3 Metamorphism

The main weakness of polymorphic malware was the fact that even though the unpacking and decryption procedures were mutated, the main malicious code was not. By allowing the malware to run in a controlled environment, the malicious code would eventually appear in memory and could be matched against a byte signature.

Metamorphic malware answers this weakness by applying obfuscating code transformations to the whole code of the malware, including the malicious parts. This way the malware changes significantly from one mutation to another and leaves little space for simple byte signature detection. The downside to this malware is the complexity of a mutation engine capable of such code transformations. Figure 2.7 shows the process of generating a new

mutation from an old one, once a metamorphic malware is executed. The best example of an implementation is the virus Simile and its MetaPHOR mutation engine.

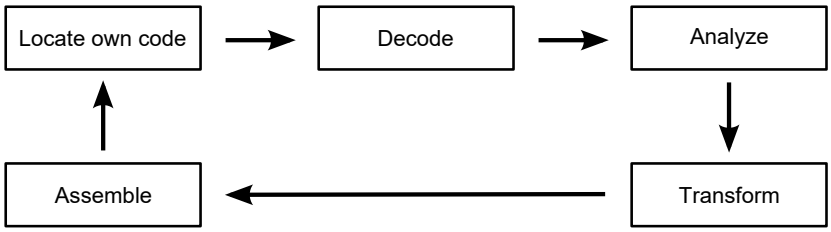


Figure 2.7: Mutation stages of a metamorphic malware

2.2 Syntactic Detection

Historically malware detection techniques were centered around syntactic properties of a binary file under inspection. These methods often look for specific byte sequences in files and any file that contains a byte sequence that is deemed malicious is marked as malware. The detection rates of these methods were heavily dependent on the length of the byte sequence which was looked for and the structure of the malware code. The advantages of this approach, to this day, are speed and scalability. The main disadvantage is that they are easy to bypass using any kind of obfuscation. To circumvent this, several modifications were introduced.

Wildcards The modification adds symbols with special semantics into the byte signature. Consider Figure 2.8. The upper sequence of bytes is a part of the W32/Beast virus and the lower sequence is the wildcard byte signature used to detect it. The `?` symbol marks an optional occurence of a half-byte, while `%2` says that the next byte may occur twice in the following two bytes.

```
83EB 0274 1683 EB0E 740A 81EB 0301 0000
83EB 0274 ??83 EB0? 740A 81EB %2   0301 0000
```

Figure 2.8: An example of a wildcard byte signature

Mismatches Allow an inexact matching of a byte signature. The following example illustrates an algorithm that allows a mismatch of three bytes. After encountering a mismatch, the algorithm notes the mismatch and the next byte of the signature is used for matching. Figure 2.9 shows a byte signature and three malware byte sequences that the signature matches.

```
signature: 1122 3344 5566 7788 99AA
malware 1: A311 2233 C944 5566 0A77 8899 AABB
malware 2: 1134 2233 C444 5566 6777 8899 AABB
malware 3: 1122 3344 D4DD E555 6677 8899 AA
```

Figure 2.9: An example of mismatch byte signatures

2.2.1 Algorithmic Scanning

As obfuscated malware grew more commonplace, detection moved to methods that used malware-specific heuristics for detection. An example of such a heuristic is *smart scanning* which was able to beat simple obfuscations by ignoring `NOP` instructions or by performing matching only in the scope of basic blocks. *X-Ray scanning* on the other hand targeted malware encrypted with weak ciphers. The method used a brute-force attack to uncover the encryption key and once uncovered performed a byte signature matching over the decrypted body. *Filtering* restricted detection to only those files that could contain malicious code.

Finally a generalized version of heuristic detection was dubbed *algorithmic scanning* and introduced specialized malware description languages that aided the creation of detection procedures. An algorithmic detector was then essentially an interpreter of such a language coupled with a database of detection procedures.

2.2.2 Code Emulation

Emulation-based detection methods were developed as an answer to strong polymorphic malware. In an emulation-based detector, the suspicious executable is allowed to run in a controlled environment. In case of a polymorphic malware, the packed or encrypted malicious code will eventually be exposed in the memory of the environment, where it can be again detected by a byte signature.

The success of this method is however dependent on how precisely the controlled environment simulates a real host. If the malware detects that its running in a controlled environment it can halt any malicious activity and perform bening computations. Or the malware can exhibit its malicious behavior only under certain conditions, such as time or a certain host system language localization.

The main disadvantage however is that an antivirus must not hinder a legitimate user in his use of a system and its files. Therefore the detection method must be fast, which in the context of an emulation-based detector means that it has only limited time to spend on a single file. So the simplest way for a malware to avoid detection is to perform benign computation until the detector runs out of time.

2.3 Behavioral Detection

With the advent of strongly polymorphic and metamorphic malware syntactic detectors proved to be insufficient. The reason for this was the large number of syntactic signatures that detection of such malware would require. Research has even proved that a syntactically undetectable malware is possible [7].

The problem of strongly obfuscated malware was further demonstrated by the outbreak of the Storm worm in 2007. The authors of Storm did not equip the worm with its own mutation engine, instead they released a large number of mutations in bursts into the public internet. This way the antivirus companies were not able to reverse-engineer the worms mutation engine, thus come up with a suitable heuristic detection procedure in time.

The proposed solution to this are behavioral detection methods, which abstract from how the malicious behavior is implemented and aim to detect the malicious behavior itself. Many of the detection methods are directly related to methods used in software testing and quality assurance. The advantage of this approach is its generality and robustness against any kind of syntactic obfuscation. The main disadvantage is the high computational

complexity of these methods. Figure 2.10 shows the structure of a generic behavioral detector and serves as a reference point when classifying behavioral detectors.

The first phase of the detection process is data collection. In this phase the input malware executable is analyzed, and raw characteristics about its behavior are collected. These characteristics can be logs of actions the malware made while executed in a controlled environment or assembly code obtained by disassembly of the executable file. Raw characteristics tend to be very fine grained and need refining. This is done in the interpretation phase, where the raw characteristics are analyzed and a high-level abstraction is built. Action logs are typically interpreted into high-level events, like establishing a network connection. Assembly code can be further analyzed to obtain dataflow or control flow graphs. High-level abstractions can be used in two ways. If the input executable is known to be malicious, the abstraction is used to generate a behavioral signature to enrich the signature database used for matching. If the abstraction is unknown, it is matched against the signatures in the database. The matching algorithm, the form of a behavioral signature and the high-level abstraction are closely related. If the abstraction is a sequence of high-level events, a state machine can be used as a behavioral signature and matching is performed by checking if the state machine accepts the sequence. In the case of a dataflow graph, the signature is typically a smaller dataflow graph representing malicious dataflow and the matching is done by checking if the malicious dataflow graph is present in the unknown dataflow graph.

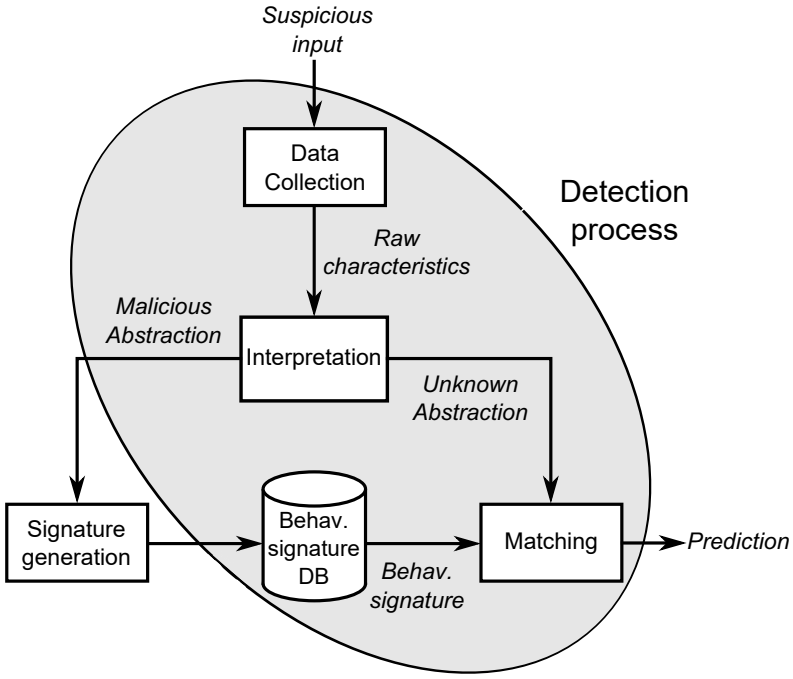


Figure 2.10: A generalized behavioral detector

Figure 2.11 shows a taxonomy of behavioral detectors as presented in [14]. The diagram is vertically split into two parts according to the general approach to detection. The parts themselves are divided further horizontally to describe the data structures and algorithms used in each of the detection stages introduced in Figure 2.10. Finally the lower part of the Figure shows how behavioral signatures are generated from known malware, while the upper part shows how behavior is extracted from the environment in which malware can

appear.

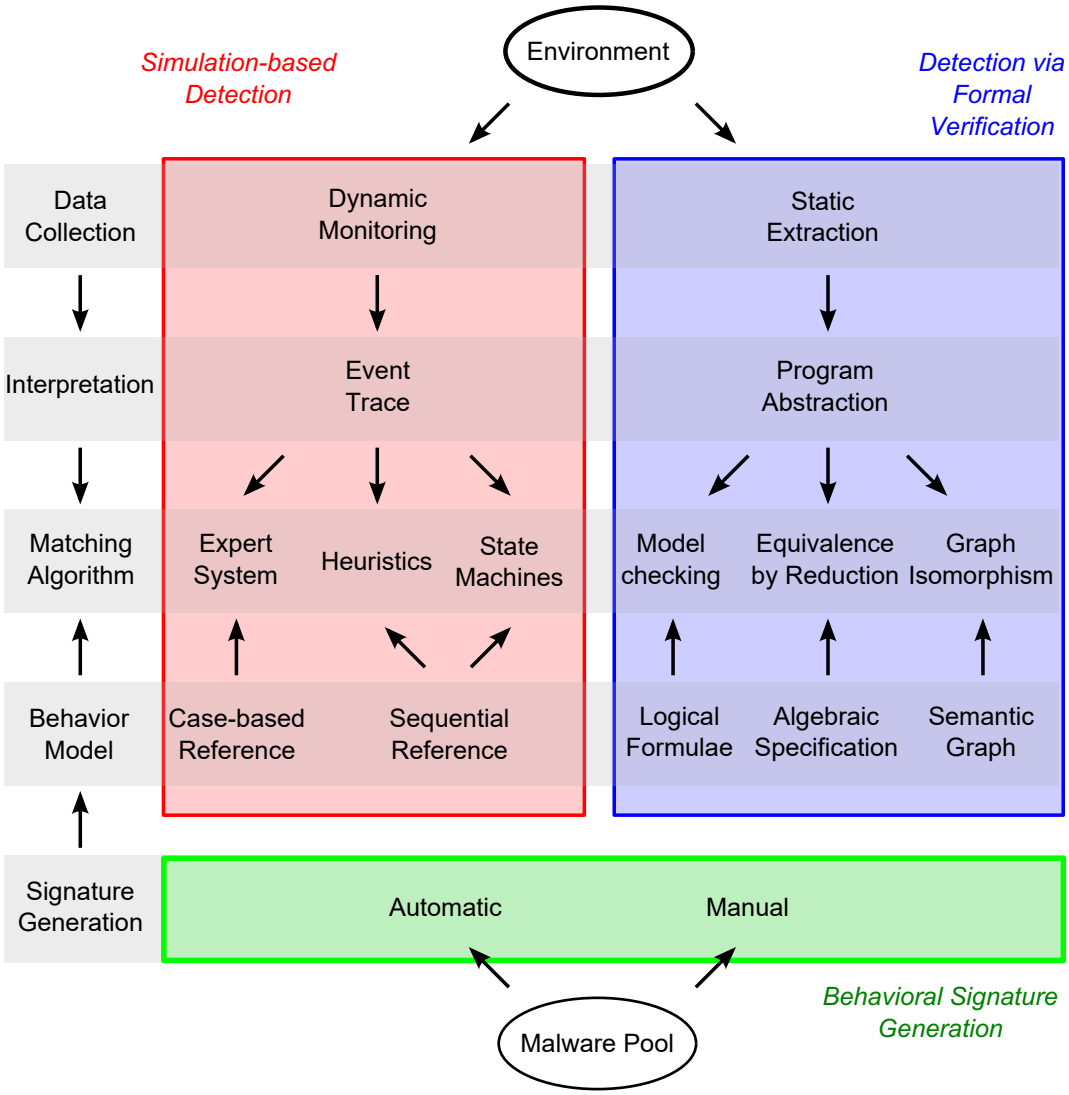


Figure 2.11: A taxonomy of behavioral detectors

2.3.1 Simulation-based Detection

Based in black-box testing, this method analyzes malware behavior in a monitored environment during execution. This environment may be controlled, in which case code emulation is employed, or in some applications the malware may be left to run on a live host.

Data collection Detectors of this type perform data collection by monitoring events that the malware causes in the host system, be it a controlled environment or a live host. Sequences of these events are called *traces*. An example of a trace is a sequence of system calls or a sequence of sent network packets.

Interpretation Traces are then often interpreted into *atomic behaviors*. These often represent a high level event in the host system e.g. opening of a file or establishing a

network connection. Atomic behaviors can then be organized into sequences forming a behavioral fingerprint of an inspected process.

Matching Matching is performed based on the model of malicious behavior. Common models include weighted rule tables, graphs of atomic behaviors or state machines. The prediction is then made based on a sum of weights and a threshold, existence of a path in a graph or the acceptance of a state machine.

The advantages and disadvantages of this approach are closely tied to its dynamic nature. One can only observe one of many possible behaviors at once, if code emulation is employed, the malware can simply not perform malicious actions in the controlled environment and allowing execution on a live host presents considerable risks.

2.3.2 Detection via Formal Verification

Behavioral detection is commonly associated with dynamic analysis. This may seem shortsighted given that the behavior of a program is fully specified by its code. In the context of information system, formal verification is used to mathematically prove or disprove that the system satisfies a given specification or that the system has a certain property. Applied to malware detection the verified system is a representation of a computer program and the property is the presence of known malicious behavior.

Data collection Since binary files are the most common representation of malware data collection is done by static extraction. This task mainly involves disassembling the binary file into assembly code, but unpacking and decryption are often needed as preliminary steps.

Interpretation Once a suitable code representation is obtained, methods for formal analysis and verification can be applied to build an abstraction of the program behavior. The abstraction should model the program behavior in such a way that is robust against obfuscations, but on the other hand should also provide a precise enough description so that false positives stay low. Common abstractions include annotated semantic graphs derived from the program data or control flow, expressions in abstract algebras or the state space of the malware.

Matching Algorithms used for matching are directly linked to the chosen abstractions and often can be reduced to solving problems tied to the abstraction. In the case of annotated semantic graphs, matching can be reduced to the subgraph isomorphism problem, where malicious behavioral signatures are represented as graph fragments and matched. In the case of abstract algebra, the program abstraction is reduced using rewriting rules and checked for equivalence with signatures represented by expressions as well. Finally in the case of a state space abstraction, model checking algorithms are used to investigate whether malicious properties represented by temporal logic formulae are satisfied by the state space.

The main advantage of detection by formal verification is that all possible execution paths, therefore all possible behaviors of a potential malware are analyzed. This means that techniques used to thwart dynamic analysis have little effect. The main disadvantages however are the complexity of static extraction, since binary code is generally difficult to analyze and can be obfuscated easily and the computational complexity of the algorithms used for interpretation and matching. For example general subgraph isomorphism is a NP-complete [11] problem while LTL model checking is a PSPACE-complete problem [22].

Chapter 3

Preliminaries

This chapter introduces theoretical concepts that underlie the methods and algorithms used in the behavioral detector presented in chapter 4 as well as the LLVM compiler infrastructure used to implement it. Abstract interpretation is introduced as a general framework for building static program analyses and tree automata are presented as the formal model used for matching abstracted program behavior with known malicious behavioral signatures. The sections on static analysis and abstract interpretation are based on [24] and corresponding chapters from [19]. Parts on tree automata and related notions are based on [5] and [2]. Finally the overview of the LLVM project and its intermediate representation is based on [16].

3.1 Static Analysis

The definition of what constitutes a static program analysis varies. Generally the term is used to describe any kind of automated collection of information about a computer program without executing the program. Under this definition, static analyses range from a simple search for syntactic patterns to precise computations over a program abstraction and even model checking which systematically searches the state space of a program.

As a collection of general methods and algorithms, static analysis is traditionally used in compiler optimizations to identify redundant or ineffective code, code generators, where it provides information needed to bridge the gap between code representations and tools for formal verification, where it is used for quality assurance. Recently however information security specialists and malware analysts have also found uses for static analysis when evaluating system vulnerabilities and analyzing malware.

3.1.1 Partially Ordered Sets and Lattices

Definition 3.1.1 (*Partially Ordered Set*)**.** Let L be a set. A partial ordering $(\sqsubseteq) \subseteq L \times L$ is a relation that is *reflexive* $(\forall l \in L : l \sqsubseteq l)$, *transitive* $(\forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3)$ and *anti-symmetric* $(\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2)$. For $(x, y) \in \sqsubseteq$ we shall write $x \sqsubseteq y$, and $x \sqsubset y$ if also $x \neq y$. The tuple $(L, \sqsubseteq)$ is called a *partially ordered set*.

Definition 3.1.2 (*Least Upper Bound*)**.** Let $(L, \sqsubseteq)$ be a partially ordered set. A subset $Y \subseteq L$ has $l \in L$ as an *upper bound* if $\forall l' \in Y : l' \sqsubseteq l$. An upper bound l of Y is a *least upper bound*, written as $\sqcup Y$, if $\forall l_0 \in L : l \sqsubseteq l_0$, where l_0 is an upper bound of Y. Symbol $\sqcup$ shall denote the *join* operator and for $\sqcup\{l_1, l_2\}$ we shall write $l_1 \sqcup l_2$.

Definition 3.1.3 (*Greatest Lower Bound*)**.** Let $(L, \sqsubseteq)$ be a partially ordered set. A subset $Y \subseteq L$ has $l \in L$ as a *lower bound*, written $\sqcap Y$, if $\forall l' \in Y : l \sqsubseteq l'$. A lower bound l of Y is a *greatest lower bound* if $\forall l_0 \in L : l_0 \sqsubseteq l$, where l_0 is a lower bound of Y. Symbol $\sqcap$ shall denote the *meet* operator and for $\sqcap\{l_1, l_2\}$ we shall write $l_1 \sqcap l_2$.

Definition 3.1.4 (*Lattices*)**.** A *join-semilattice* or *meet-semilattice* is a partially ordered set $(L, \sqsubseteq)$ in which every pair of elements in L has a least upper bound or greatest lower bound, respectively. A semi-lattice is *complete* if every subset of L has a least upper bound or greatest lower bound. Furthemore $\bot = \sqcup\emptyset = \sqcap L$ is the *least element* and $\top = \sqcap\emptyset = \sqcup L$ is the *greatest element*. A *lattice* $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a partially ordered set that is a meet-semilattice and a join-semilattice. A lattice is complete if it is a complete join-semilattice and a complete meet-semilattice.

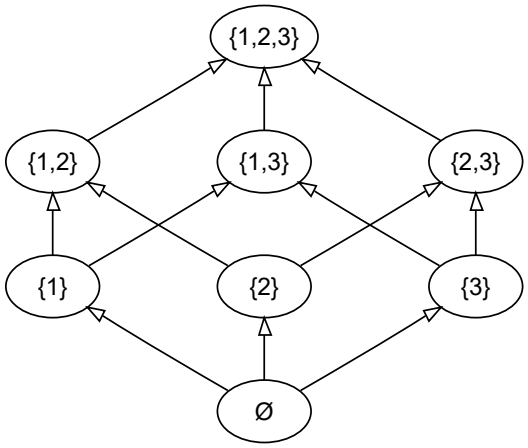


Figure 3.1: A Hasse diagram of the $(2^{\{1,2,3\}}, \subseteq, \cup, \cap, \emptyset, \{1,2,3\})$ lattice

Definition 3.1.5 (*Chain*)**.** Let $(L, \sqsubseteq)$ be a partially ordered set. A *chain* from x_0 to x_n is a set $C = \{x_0, x_1, \ldots, x_n\} \subseteq L$ where $x_0 \sqsubset x_1 \sqsubset x_2 \sqsubset \cdots \sqsubset x_n$.

Definition 3.1.6 ($\sqcup-$*continuous function*)**.** Let $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ be lattices and $f : A \to B$ a function. Function f is *monotone* if $\forall x, y \in L : x \sqsubseteq_A y \Rightarrow f(x) \sqsubseteq_B f(y)$. Furthermore f is $\sqcup-$*continuous* if $f(\sqcup C) = \sqcup\{f(c) | c \in C\}$ for every chain $C \subseteq A$.

Definition 3.1.7 (*Least Fixed Point*)**.** Let $(L, \sqsubseteq)$ be a lattice and $f : L \to L$ a function. An element $x \in L$ is a *fixed point* of f if $f(x) = x$ and a *least fixed point* if $\forall y \in L : f(y) = y \Rightarrow x \sqsubseteq y$.

Theorem 3.1.1 (*Kleene Fixed Point*)**.** *Let $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ be a lattice and $f : A \to A$ a $\sqcup-$continuous function. The least fixed point of f is $\mu f = \sqcup\{f^i(\bot) | i \geq 0\}$.*

The Kleene fixed point theorem essentially tells that by repeatedly applying a $\sqcup-$continuous function f over a lattice starting from the least element of the lattice, one will eventually reach the least fixed point of f while forming a chain along the way. This is very useful when designing iterative algorithms over data that form a lattice. Such an algorithm iteratively refines an approximate solution and is guaranteed to terminate if the lattice is finite and the algorithm has the properties of a $\sqcup-$continuous function.

3.1.2 Abstract Interpretation

Pioneered by Patrick and Radhia Cousot in [6], *abstract interpretation* is a framework for building static analyses based on lattices. The basic idea is to model the states a computer program can be in via *abstract contexts* over an *abstract domain* that is suitable for gathering the information that we want from the static analysis. If we wish to know the possible values of integer variables in a program, we may use intervals of natural numbers or convex polyhedra as our abstract domain. If we're collecting information about strings, we will probably use an abstract domain based in formal language theory.

Each instruction in the program is assigned an *abstract transformer* which emulates the execution of the instruction in the abstract domain. An addition over two integer variables will modify the intervals corresponding to those variables in the abstract context of the program. Finally the analysis is done by iteratively computing the abstract contexts of the program until the computation reaches a fixed point. In some cases however, e.g. programs with loops, the reaching of a fixed point is not guaranteed. A *widening* operator over abstract contexts which over-approximates the precise iterative solution may be introduced to guarantee termination and speed up the analysis. The application of widening however degrades the results of the analysis. *Narrowing* operators can be applied to the results of a widening to refine the final result of the analysis.

Definition 3.1.8 (*Abstract Interpretation*)**.** An *abstract interpretation* I of a program P with the instruction set `Instr` is a tuple

$$I = (Q, \circ, \sqsubseteq, \bot, \top, \tau)$$

where

- Q is the abstract domain (a set of abstract contexts),
- $\circ : Q \times Q \to Q$ is the *join operator* for accumulation of abstract contexts and $(Q, \circ, \top)$ is a complete join-semilattice,
- $(\sqsubseteq) \subseteq Q \times Q$ is a partial ordering defined as $\forall x, y \in Q : x \sqsubseteq y \Leftrightarrow x \circ y = y$ in $(Q, \circ, \top)$,
- $\bot$ is the least element of Q,
- $\top$ is the greatest element of Q,
- $\tau : \texttt{Instr} \times Q \to Q$ defines an interpretation of abstract transformers.

Definition 3.1.9 (*Widening*)**.** Let $I = (Q, \circ, \sqsubseteq, \bot, \top, \tau)$ be an abstract interpretation. For the binary *widening* operator $\triangledown$ the following holds:

- $\triangledown : Q \times Q \to Q$,
- $\forall C, D \in Q : (C \circ D) \sqsubseteq (C \triangledown D)$,
- for all infinite sequences $(C_0, C_1, \ldots, C_n, \ldots) \in Q^\omega$, it holds that the infinite sequence $(s_0, s_1, \ldots, s_n, \ldots)$ defined recursively as $s_0 = C_0,\ s_n = s_{n-1} \triangledown C_n$ is not strictly increasing.

Definition 3.1.10 (*Narrowing*)**.** Let $I = (Q, \circ, \sqsubseteq, \bot, \top, \tau)$ be an abstract interpretation. For the binary *narrowing* operator $\triangle$ the following holds:

- $\triangle : Q \times Q \to Q$,
- $\forall C, D \in Q : D \sqsubseteq C \Rightarrow (D \sqsubseteq (C \triangle D) \sqsubseteq C)$,
- for all infinite sequences $(C_0, C_1, \ldots, C_n, \ldots) \in Q^\omega$, it holds that the infinite sequence $(s_0, s_1, \ldots, s_n, \ldots)$ defined recursively as $s_0 = C_0,\ s_n = s_{n-1} \triangle C_n$ is not strictly decreasing.

To ensure certain properties of the abstract interpretation with regard to the analyzed program, one may employ *Galois connections*. The most common property investigated is *soundness* which ensures that the abstraction over-approximates the behavior of the analyzed program and thus can only produce false positives. In the context of malware detection, this means that it may find malicious behavior that is present in the abstract domain, but not in the program itself, assuming a precise specification of what malicious behavior is given.

3.2 Tree Automata

Introduced as a generalization of word languages in formal language theory, tree languages describe sets of directed trees or *terms*. Their defining formalisms *tree automata* and *regular tree grammars* found a wide array of uses. Automata in areas of formal verification and automated reasoning as parts of decision procedures for logics. Grammars in areas of computational linguistics and programming languages.

3.2.1 Ranked Alphabets, Terms and Trees

Definition 3.2.1 (*Ranked Alphabet*)**.** Let $\mathcal{F}$ be a finite set of symbols, $\mathbb{N}$ the set of natural numbers and $arity : \mathcal{F} \to \mathbb{N}$ a function. A *ranked alphabet* is the tuple $(\mathcal{F}, arity)$. The *arity* of a symbol $f \in \mathcal{F}$ is denoted $arity(f)$ and the set $\mathcal{F}_n = \{f \mid f \in F \wedge arity(f) = n\}$ is the set of symbols of arity p. Symbols of arity $0, 1, \ldots, p$ are respectively called constants, unary symbols, ..., n-ary symbols.

Definition 3.2.2 (*Terms*)**.** Let $\mathcal{F}$ be a ranked alphabet. The set $T(\mathcal{F})$ of *terms* over the ranked alphabet $\mathcal{F}$ is the smallest set defined by:

- $\mathcal{F}_0 \subseteq T(\mathcal{F})$,
- if $n \geq 1,\ f \in \mathcal{F}_n,\ t_1, \ldots, t_n \in T(\mathcal{F})$, then $f(t_1, \ldots, t_n) \in T(\mathcal{F})$.

Definition 3.2.3 (*Tree Domain*)**.** Let $\mathbb{N}$ be the set of natural numbers and $\mathbb{N}^*$ the *free monoid*[1] generated by $\mathbb{N}$ with concatenation $(\cdot)$ as the operation and the empty string ε as the identity. The *prefix order* $(\leq) \subseteq \mathbb{N}^* \times \mathbb{N}^*$ is defined as: $u \leq v \overset{\text{def}}{\iff} \exists w \in \mathbb{N}^* : v = u \cdot w$. For $u \in \mathbb{N}^*$ and $n \in \mathbb{N}$, the *length* $|u|$ is defined inductively: $|\varepsilon| = 0,\ |u \cdot n| = |u| + 1$. A set S is *prefix-closed* if $u \leq v \wedge v \in S \Rightarrow u \in S$. A *tree domain* is a finite non-empty prefix-closed set $D \subset \mathbb{N}^*$ satisfying the following property: $u \cdot n \in D \Rightarrow \forall 1 \leq j \leq n : u \cdot j \in D$.

Definition 3.2.4 (*Finite Tree*)**.** Let $T(\mathcal{F})$ be a set of terms and D a tree domain. A term $t \in T(\mathcal{F})$ can be represented as a *finite tree* which is a mapping $t : D \to \mathcal{F}$ satisfying the following:

[1] for a definition see [18]

- $\forall u \in D$: if $t(u) \in \mathcal{F}_n,\ n \geq 1$ then $\{j \mid u \cdot j \in D\} = \{1, \ldots, n\}$
- $\forall u \in D$: if $t(u) \in \mathcal{F}_0$ then $\{j \mid u \cdot j \in D\} = \emptyset$

The domain of tree t shall be written as $dom(t)$ and each element in $dom(t)$ represents a *position* in t. The symbol $t(\varepsilon)$ is called the *root symbol* of t.

Definition 3.2.5 (*Height*)**.** Let $t \in T(\mathcal{F})$ be a term. The *height* of the term t, denoted $height(t)$ is defined inductively as

$$height(t) = \begin{cases} 1 & \text{if } t \in \mathcal{F}_0 \\ 1 + max(\{height(t_i) \mid i \in \{1, \ldots, n\}\}) & \text{if } t(\varepsilon) \in \mathcal{F}_n \end{cases}$$

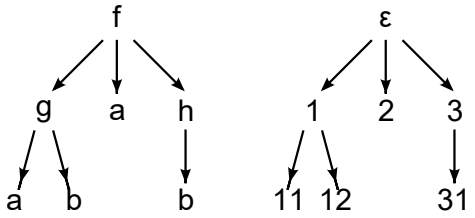


Figure 3.2: An example of a tree t (left) and its tree domain D (right). $height(t) = 3$

Definition 3.2.6 (*Subterm*)**.** Let $t \in T(\mathcal{F})$ be a term. A *subterm* of t at position p denoted $t|_p$ is defined by the following:

- $dom(t|_p) = \{j \mid pj \in dom(t)\}$,
- $\forall q \in dom(t|_p) : t|_p(q) = t(pq)$.

3.2.2 Finite Tree Automata

An extension of finite automata which work on words, finite tree automata deal with finite trees over a ranked alphabet. Indeed words over a finite alphabet can be viewed as terms over a ranked alphabet of unary symbols and a constant acting as a terminator. For example the word xyz over the alphabet $\Sigma = \{x, y, z\}$ ca be viewed as the term $t = x(y(z(\#)))$ over the ranked alphabet $\mathcal{F} = \{x(), y(), z(), \#\}$ where $\#$ is the added constant.

Similarly to finite automata, finite tree automata can be deterministic and nondeterministic. Finite tree automata however have different properties depending on how the computation of the automaton is performed. When the computation starts from the leaves of the tree working its way to the root, the automaton is referred to as *bottom-up*. In the opposite case the automaton is a *top-down* one.

Depending on the direction of computation, determinism has different consequences on the expressive power of the automaton. Determnistic bottom-up finite tree automata have the same power as nondeterministic ones and nondeterministic top-down ones. However deterministic top-down finite tree automata are less powerful [5]. This work will only consider finite deterministic bottom-up tree automata onwards, referring to them as FDTA.

Definition 3.2.7 (*Finite Deterministic Bottom-Up Tree Automaton*)**.** A FDTA over a ranked alphabet $\mathcal{F}$ is a tuple

$$\mathcal{A} = (Q, \mathcal{F}, \delta, Q_f)$$

where

- Q is a set of states,
- $Q_f \subseteq Q$ is a set of final states,
- $\delta = \bigcup_i \delta_i$ is a set of transition relations defined $\delta_0 : \mathcal{F}_0 \to Q$ and $\delta_n : (\mathcal{F}_n \times Q^n) \to Q$ for $n > 0$.

Definition 3.2.8 (*Recognized Tree Language*)**.** Let $t \in T(\mathcal{F})$ be a term and $\mathcal{A} = (Q, \mathcal{F}, \delta, Q_f)$ an FDTA. The relation $\delta_{\mathcal{A}} : T(\mathcal{F}) \to Q$ describes the run of FDTA $\mathcal{A}$ on t recursively as:

$$\delta_{\mathcal{A}}(t) = \begin{cases} \delta_0(f) & \text{if } t(\varepsilon) = f \wedge f \in \mathcal{F}_0 \\ \delta_n(f, \delta_{\mathcal{A}}(t_1), \ldots, \delta_{\mathcal{A}}(t_n)) & \text{if } t(\varepsilon) = f(t_1, \ldots, t_n) \wedge f \in \mathcal{F}_n \end{cases}$$

The tree language *recognized* by FDTA $\mathcal{A}$ is the set $L(\mathcal{A}) = \{t \mid t \in T(\mathcal{F}) \wedge \delta_{\mathcal{A}}(t) \in Q_f\}$.

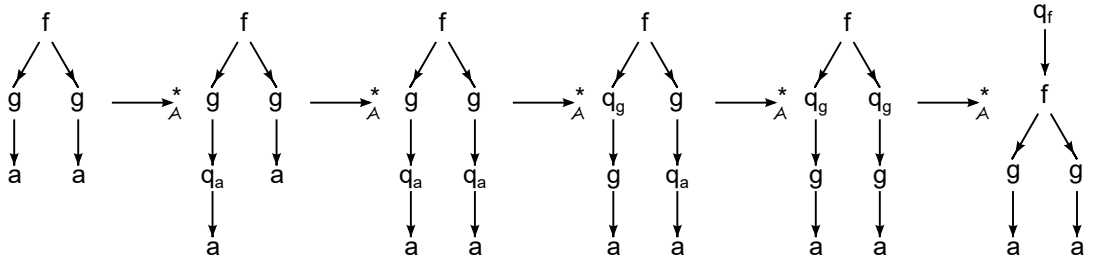


Figure 3.3: An accepting run of a FDTA $\mathcal{A} = (Q, \mathcal{F}, \delta, Q_f)$ on term $t = f(g(a), g(a))$. $Q = \{q_a, q_g, q_f\}, \mathcal{F} = \{f(,), g(), a\}, \delta_0(a) = q_a, \delta_1(g, q_a) = q_g, \delta_2(f, q_g, q_g) = q_f, Q_f = \{q_f\}$.

3.2.3 Tree Automata Inference

The field of *grammatical inference* studies methods and algorithms for machine learning of formal languages by inferring their models from a set of samples. The set of samples may be positive or negative. Positive samples are elements of the inferred language, while negative are not. Research tells that regular word languages cannot be inferred from positive samples only [12].

The solution is either to learn from positive and negative samples or restricting the inference to less expressive languages. For regular word languages, inferring a minimal finite automaton from positive and negative samples is NP-complete [12]. Inferring a non-minimal automaton is also an option, but there is a risk of exponential blowup of the automaton size. Using less expressive language families on the other hand presents the possibility of inferring a state-minimal automaton from positive samples only [10].

K-testable languages present a subclass of regular languages for which the membership of a string can be decided based on the membership of substrings of a bounded length. This section presents *k-testable tree languages* and the construction of a minimal tree automaton from positive samples only.

Definition 3.2.9 (*Placeholders*)**.** Let $\mathcal{F}$ be a ranked alphabet and $\Xi = \{\xi_f \mid f \in \bigcup_{i>0} \mathcal{F}_i\}$ be a new set of constants called *placeholders* such that $\Xi \cap \mathcal{F} = \emptyset$. A constant $\xi_f \in \Xi$ can only be substituted with a tree t for which $t(\varepsilon) = f$. Let $T(\mathcal{F}, \Xi)$ denote the set of trees over the ranked alphabet and placeholders.

Definition 3.2.10 (*Link*)**.** Let $t, t' \in T(\mathcal{F}, \Xi)$ be trees. The link operation $t\#t'$ is defined as

$$(t\#t')(n) = \begin{cases} t(n) & \text{if } t(n) \notin \Xi \vee (t(n) = \xi_f \wedge f \neq t'(\varepsilon)) \\ t'(z) & \text{if } n = y \cdot z,\ t(y) = \xi_{t'(\varepsilon)},\ y \in dom(t),\ z \in dom(t') \end{cases}$$

Intuitively the link $(t\#t')(n)$ over represents a substitution of tree t' into tree t at point n in t. For the operation to be successful t needs to have a placeholder symbol ξ_f on position n and t' needs to have symbol f as its root. If these conditions are met, the placeholder symbol in ξ_f in t is replaced by t'. Otherwise the operation does nothing and the result is the unchanged tree t.

Definition 3.2.11 (*Tree Quotient*)**.** Let $t, t' \in T(\mathcal{F})$ be trees. The *tree quotient* $t^{-1}t'$ is defined as

$$t^{-1}t' = \{t'' \in T(\mathcal{F}, \Xi) \mid t' = t''\#t\}$$

Given trees t and t' without placeholder symbols, the tree quotient $t^{-1}t$ is set of trees containing placeholder symbols and for each tree t'' in the set has the property, that t can be linked into t'' to form t. Furthermore, the tree quotient operation can also be extended to sets in the following manner: $t^{-1}S = \{t^{-1}t \mid t' \in S\}$.

Definition 3.2.12 (*K-Root*)**.** Let $t \in T(\mathcal{F})$ be a tree. For $k \geq 0$ a *k-root* of t is defined as:

$$root_k(t) = \begin{cases} t & \text{if } t(\varepsilon) \in \mathcal{F}_0 \\ \xi_f & \text{if } f = t(\varepsilon),\ f \in \bigcup_{i>0} \mathcal{F}_i,\ k = 0 \\ f(root_{k-1}(t_1), \ldots, root_{k-1}(t_n)) & \text{if } t = f(t_1, \ldots, t_n),\ height(t) > k > 0 \end{cases}$$

Theorem 3.2.1 (*K-Testable Tree Language* [17])**.** *Let $L \subseteq T(\mathcal{F})$ be a tree language. L is k-testable in the strict sense if and only if for any $t_1, t_2 \in T(\mathcal{F})$ such that $root_k(t_1) = root_k(t_2)$, when $t_1^{-1}L \neq 0 \wedge t_2^{-1}L \neq 0$ then it follows that $t_1^{-1}L = t_2^{-1}L$.*

Definition 3.2.13 (*Root Equivalence*)**.** Trees $t_1, t_2 \in T(\mathcal{F})$ are *root-equivalent with degree* k for some $k \geq 0$, denoted $t_1 \sim_k t_2$ if and only if $root_k(t_1) = root_k(t_2)$. The $\sim_k$ relation is also a congruence of finite index.

Definition 3.2.14 ($\sim_k$*-Induced Tree Automaton*)**.** Let $T' \subseteq T(\mathcal{F})$ be finite set of trees. The $A^{\sim_k}(T') = (Q, \mathcal{F}, \delta, Q_f)$ automaton induced by the root equivalence relation $\sim_k$ is defined as:

$$\begin{aligned} Q &= \{root_k(t') \mid \exists t \in T' : \exists u \in dom(T') : t'|_u\} \\ Q_f &= \{root_k(t) \mid t \in T'\} \\ \delta_0(f) &= f \text{ for } f \in \mathcal{F}_n \\ \delta_n(f, root_k(t_1), \ldots, root_k(t_n)) &= root_k(f(t_1, \ldots, t_n)) \text{ for } n \geq 1,\ f \in \mathcal{F}_n \end{aligned}$$

Theorem 3.2.2 ([2])**.** *$L(A^{\sim_k})$ is k-testable in the strict sense.*

Theorem 3.2.3 ([9])**.** $L(A^{\sim_{k+1}}) \subseteq L(A^{\sim_k})$

3.3 The LLVM Compiler Infrastructure

LLVM is a collection of tools and libraries for building compilers, toolchain technologies and code analyzers. At the core of the framework is the LLVM intermediate representation (IR). LLVM IR is a SSA-based, RISC-like assembly language used to represent code throughout the framework. The IR code can be serialized into a textual or binary form, which allows for compile-time, link-time and even „idle-time“ transformations. The core LLVM tools are built around the IR and manipulate it. The tools include an assembler to the binary format of the IR, disassembler to the textual form of the IR, an optimizer that runs transformation passes over the IR, a linker which combines serialized IR files and several others.

A number of projects have been made using the LLVM framework. The original project is Clang, a compiler front-end for C-like programming languages. However thanks to its language-agnostic nature, front ends for other static and dynamic languages have been made as well as implementations of runtime environments like the Java Virtual Machine.

Another group of projects are static and dynamic code analyzers. Clang includes its own built-in static analyzer which aims to highlight common mistakes and inefficiencies a programmer can make. Another static analyzer is KLEE, which uses symbolic execution, a form of abstract interpretation, to generate test cases for code represented in LLVM IR. The last example of a static analyzer built on LLVM is LLBMC, a bounded model checker. As for dynamic analysis, LLVM includes several code instrumentation tools, called „sanitizers“, which generate additional code for runtime analysis.

3.3.1 Intermediate Representation

The LLVM IR is a target-independent, typed assembly language. The code represented in the IR is in the *single static assignment* form which means that each variable in the IR is only assigned once. This means that each use of a variable can be easily traced back to its definition, which immensely simplifies analysis and transformations of the IR. The number of variables that can be assigned this way is not limited. Figure 3.4 shows a piece of C code and the corresponding LLVM IR.

```
int sum(int a, int b){
 return a+b;
}
```

```
; Function Attrs: nounwind uwtable
define i32 @sum(i32 %a, i32 %b) #0 {
entry:
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 %b, i32* %1, align 4
  store i32 %a, i32* %2, align 4
  %3 = load i32, i32* %1, align 4
  %4 = load i32, i32* %2, align 4
  %add = add nsw i32 %3, %4
  ret i32 %add
}
```

Figure 3.4: A C function and the LLVM IR equivalent

Module The contents of a single LLVM IR file, be it binary bitcode or textual assembly form a module. A module is the top-level data structure in LLVM IR. The module contains information about target architecture, declarations of structured data types, declarations of external functions, global variables and a list of function definitions. The relationships

between functions are apparent on this level and form a *call graph*, in which nodes represent functions and an edge from A to B means that function A calls function B in its body.

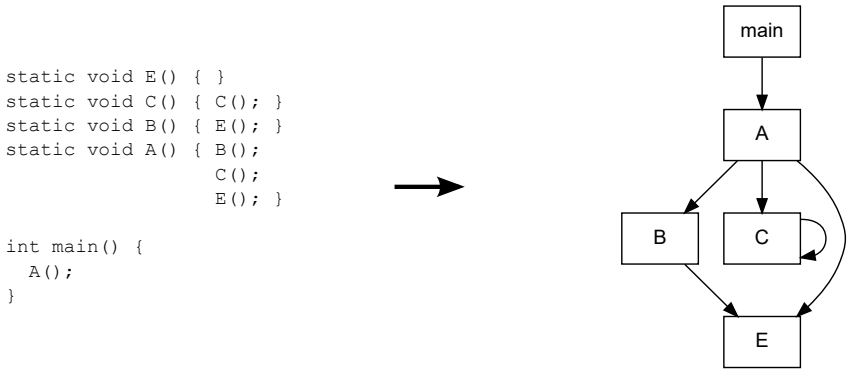


Figure 3.5: An example of a call graph

Function Each function in LLVM IR has a list of formal arguments, a return type and a body consisting of basic blocks. Additionally it can have a number of optional attributes specifying a garbage collector, exception information, linkage type and so on. The first basic block in a function is denoted by the `entry` label. Local variables have a `%` prefix while global variables have a `@` one. Relationships between basic blocks are apparent on this level and form a *control flow graph*.

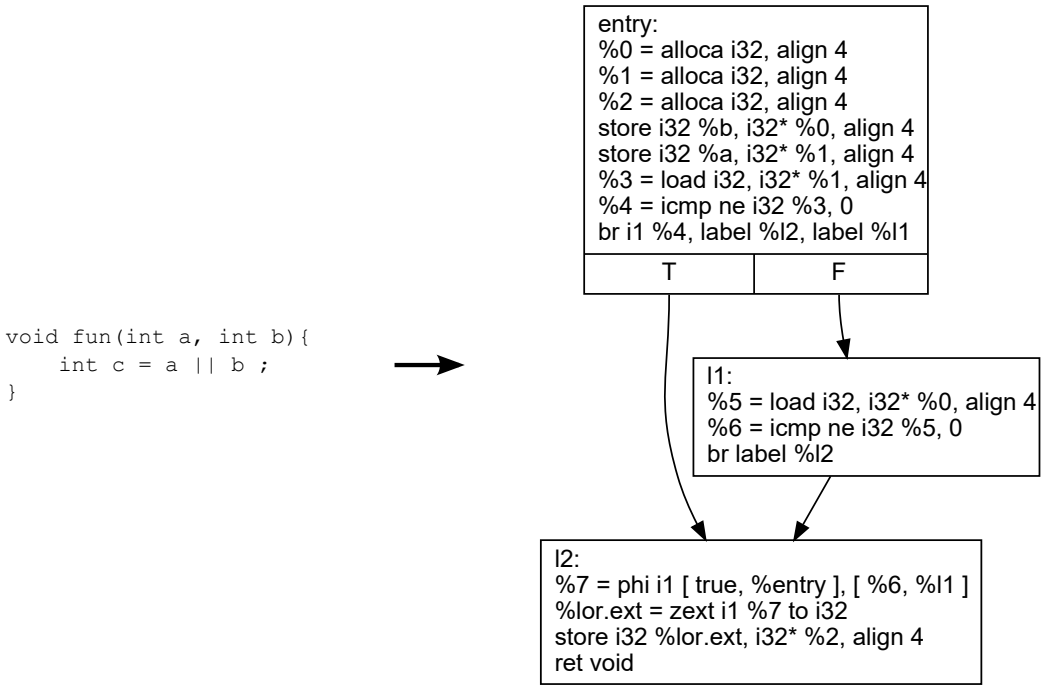


Figure 3.6: Control flow graph with a ϕ-instruction

Basic Block A collection of IR instructions that is always executed sequentially. A basic block begins with a label that is used to reference the basic block in control flow instructions and ends with a terminator instruction, usually a `br` instruction which jumps to a label of another basic block or a `ret` instruction, which exits the function. Each basic block has a list of successors and predecessors in the control flow graph. In some cases the value of a variable in the basic block is dependent on the predecessor. To maintain SSA form a *ϕ-instruction* must be present in the basic block before the use of the variable.

Chapter 4

Detector Implementation

This chapter introduces a prototype detector which uses methods of formal analysis and verification. Since a complete implementation of a behavioral detector exceeds the format of this work, the detector only implements the interpretation, signature generation and matching phase of the generic behavioral malware detector defined in Chapter 2, Section 2.3. The data collection phase is replaced by an LLVM-based obfuscating compiler [15]. Given a malware source code written in C or C++, the compiler is used to generate a mutation of the malware by randomly applying the obfuscations implemented in the compiler. The compiler is then set to output an LLVM IR file. This file represents a raw characteristics in the context of the generic detector. A set of mutations can be obtained by running the compiler repeatedly on the same source code. Details about the malware source code and applied obfuscations are given in Section 4.1.

Interpretation is done by processing the obfuscated LLVM IR files in two steps. The first step is using *static taint analysis*, which is formalized as an abstract interpretation. The analysis is implemented as a pass over LLVM IR using the tools and libraries available in the LLVM framework. The output of the analysis is a *syscall dependency graph*, which represents the high-level abstraction of malware behavior. The second step of the interpretation phase is unfolding the cycles of the syscall dependecy graph. This is done as a preliminary step before signature generation, which needs the graphs to be acyclic. Section 4.2 covers the details about static taint analysis, syscall dependency graphs and the unfolding algorithm.

Signature generation is also done in two steps. The first step is taking a syscall dependency graph, which is a directed acyclic graph and transforming it into a *maximally-shared graph*. A *maximally-shared graph* is again a directed acyclic graph, but in a form that is free from redundant subgraphs and folded into a form that can be seen as a collection of trees. The second step is inference of a k-testable tree automaton from a set of transformed syscall dependency graphs. The inferred automaton acts as a behavioral signature which can match the set of syscall dependency graphs it was inferred from, as well as variants it has not seen before. Section 4.3 describes the inference algorithm and the preceeding transformation. The section is based on [2] and the detector takes advantage of tools made by the authors.

Using an inferred k-testable tree automaton, the detector performs matching of unknown syscall dependency graphs. The unknown graph is first transformed in the same manner as training graphs during signature generation, so that it can be viewed as a collection of trees. Then the trees contained in the graph are matched against the inferred automaton and a score is computed based on the amount of trees matched and their height. The higher

the score, the more likely the graph represents malicious behavior. Section 4.4 describes the details of how the scores are computed and relationships between malicious behavior and tree height. Same as with signature generation, the section is based on [2] and uses the same tools.

4.1 Mutation Generation

As mentioned in Chapter 2, malware is most commonly found in the form of binary executable files. Obtaining malware in the form of C or C++ code was the first challenge. Thankfully various internet community forums, such as [20] host discussion about developing malware and often provide examples for educational and hobbyist purposes. The obtained source code samples target hosts running various versions of the Windows operating system and use various functions of the Windows API. The samples include three implementations of a keylogger and two file infecting viruses. The complexity of the implementations ranges from simple single-threaded code to multi-threaded code in which each thread is responsible for different malicious behavior.

4.1.1 Obfuscator-LLVM

Mutations are generated from the obtained malware source code by using Obfuscator-LLVM, or `ollvm` for short. The obfuscating compiler was built for the purpose of intellectual property protection, but its obfuscating transformations of LLVM IR can be used for the purpose of mutation generation. Four obfuscating transformations are available in the public version of `ollvm`. One of these is instruction substitution, which is covered in Chapter 2, Section 2.1.2. The description of the other three follows.

Basic Block Splitting As the name suggests, this transformation splits basic blocks at random locations and recovers original control flow by inserting unconditional branch instructions. The main purpose of this transformation is to amplify the effect of other obfuscations in `ollvm`.

Control Flow Flattening Designed for the purpose of making control flow analysis ineffective and expensive, the obfuscation transforms control flow inside a function to a flat structure using `switch-case` constructs. Original control flow is preserved using a control variable in a designated basic block.

Bogus Control Flow An advanced form of dead code insertion which makes use of *opaque predicates*. Extra basic blocks are insterted into the control flow of a function. These basic blocks contain a conditional jump to the original basic block and to a cloned version of it. However the jump is always made to the original basick block because the predicate always evaluates to the same value. The predicate is constructed in such a way that it is hard to evaluate statically. Futhermore some junk code is added into the original and the cloned basic block and inconsequential control flow from the cloned basic block is added. Finally one more opaque predicate is added to the original basic block, finishing a dead control flow loop.

4.2 Interpretation

The interpretation phase of the prototype detector analyzes LLVM IR by using static taint analysis and builds a syscall dependency graph. A syscall dependency graphs is a general directed graph and acts as a high-level abstraction of program behavior. Nodes of the graph represent calls to functions provided by the operating system and an edge from node A to node B represents a data dependecy between the calls. The dependency can be described as „call B uses data computed from the result of call A“. Dependency graphs were chosen as the behavior abstraction because they abstract from all internal computations of the malware (where most obfuscations happen) and focus only on how the malware interacts with its host system. Although some aspects of the interaction be changed through obfuscation, for example, the order in which the syscalls are made, the dependecies between them generally do not change. Static taint analysis provides the information needed to build the dependency graphs by tracking the flow of data between two points in code.

4.2.1 Static Taint Analysis

The idea behind static taint analysis is to mark the data we wish to track as tainted, mark the origins of tainted data as *taint sources* and mark points beyond which we do not track tainted data as *taint sinks*. Taints then flow through the code according to predefined *propagation rules*. This notion is formalized in definition 4.2.1.

Definition 4.2.1. Let `Var` be set of program variables, `Instr` an instruction set, `Src` $\subseteq$ `Instr` a set of taint sources and `Snk` $\subseteq$ `Instr` a set of taint sinks. *Static taint analysis* is an abstract interpretation

$$I = (Q, \circ, \sqsubseteq, \bot, \top, \tau)$$

where

- Q is a set of mappings of the form $C : \texttt{Var} \to 2^T$,
- $\circ : Q \times Q \to Q$ is a join operator such that $\forall a \in \texttt{Var} : (C_1 \circ C_2)(a) = C_1(a) \cup C_2(a)$,
- $(\sqsubseteq) \subseteq Q \times Q$ is a partial order such that $C_1 \sqsubseteq C_2 \iff \forall a \in \texttt{Var} : C_1(a) \subseteq C_2(a)$,
- $\bot$ is a mapping C such that $\forall a \in \texttt{Var} : C(a) = \emptyset$,
- $\top$ is a mapping C such that $\forall a \in \texttt{Var} : C(a) = T$,
- $\tau : \texttt{Instr} \times Q \to Q$ is an interpretation of abstract transformers such that

$$\forall \texttt{I} \in \texttt{Instr},\ \forall C \in Q : \tau(\texttt{I}, C) = \begin{cases} \mathit{Create}(\texttt{I}, C) & \text{if } \texttt{I} \in \texttt{Src} \\ \mathit{Sink}(\texttt{I}, C) & \text{if } \texttt{I} \in \texttt{Snk} \\ \mathit{Propagate}(\texttt{I}, C) & \text{otherwise} \end{cases}$$

For an abstract context C and an instruction `I`:

- $\mathit{Propagate}(\texttt{I}, C)$ taints outputs of `I` by the union of taints carried by inputs of `I`,
- $\mathit{Create}(\texttt{I}, C)$ uses $\mathit{Propagate}(\texttt{I}, C)$ and additionally taints outputs of `I` by `I`,
- $\mathit{Sink}(\texttt{I}, C)$ sets all taints carried by inputs of `I` to $\emptyset$.

Algorithm 1: Static Taint Analysis

Input: Function F, Empty abstract contexts $C_{B_1}, \ldots, C_{B_n}$, Graph G;
Output: Filled abstract contexts $\{C_{B_1}, \ldots, C_{B_n}\}$, Updated Graph G;

```
1  Queue W ← ∅;
2  foreach basic block B in F do
3  |  W.enqueue(B);
4  end
5  while W ≠ ∅ do
6  |  B ← W.dequeue();
7  |  C_new ← ∅;
8  |  foreach predecessor P of B do
9  |  |  C_new ← C_new ∘ C_P;
10 |  end
11 |  foreach instruction I in B do
12 |  |  if I is a taint sink then
13 |  |  |  foreach operand O of I do
14 |  |  |  |  if O is pointer-type variable then
15 |  |  |  |  |  C_new(O) ← ∅;
16 |  |  |  |  end
17 |  |  |  |  foreach taint T in C_new(O) do
18 |  |  |  |  |  add edge (T, I) to G;
19 |  |  |  |  end
20 |  |  |  end
21 |  |  end
22 |  |  if I is a taint source then
23 |  |  |  foreach operand O of I do
24 |  |  |  |  if O is pointer-type variable then
25 |  |  |  |  |  C_new(O) ← C_new(O) ∪ {I};
26 |  |  |  |  end
27 |  |  |  end
28 |  |  |  C_new(I) ← C_new(I) ∪ {I};
29 |  |  else
30 |  |  |  foreach operand O of I do
31 |  |  |  |  C_new(I) ← C_new(I) ∪ C_new(O);
32 |  |  |  end
33 |  |  end
34 |  end
35 |  if C_new ≠ C_B then
36 |  |  C_B ← C_new;
37 |  |  foreach successor S of B do
38 |  |  |  W.enqueue(S);
39 |  |  end
40 |  end
41 end
```

The first step when implementing static taint analysis over LLVM IR is to define taint sources and taint sinks. Since the goal is to build a syscall dependency graph, data flowing from one syscall to another needs to be tracked. In LLVM IR, syscalls can be recognized as calls to functions provided by the Windows API. However LLVM IR does not provide a way to efficiently distinquish between functions from the Windows API and other externally referenced functions. A robust solution to this problem would be to build a list of Windows API functions and check the name of functions against the list. This would however introduce a significant time overhead to the work. A reasonable approximation was found by inspecting the input LLVM IR. All Windows API functions had a `dllimport` tag associated with them, which made them stand out among other functions.

After recognizing taint sinks and sources, abstract contexts need to be defined. These are straightforward since LLVM provides map-like data structures for mapping LLVM IR variables onto arbitrary data structures and types, including sets of instruction references. This way every LLVM IR variable has a set of references to call instructions associated with it in a map.

Finally abstract transformers create, sink and propagate taints between abstract contexts. A call instruction in LLVM IR calls a function with a set of arguments and returns a single result which is assigned according to the SSA form. In this case the transformers can use the explicit def-use chains provided by LLVM IR and easily propagate taints from sources to sinks and tainting variables along the way. However a large number of functions in the Windows API returns results via variables passed by a pointer reference. In these cases the SSA form does not apply and the taints need to be propagated differently. Algorithm 1 describes the propagation of taints in such a setting. It is also important to note that the algorithm updates a whole program dependency graph G. This is done precisely because the output abstract contexts do not reflect the evolution of variables passed into call instructions by a pointer reference. Another solution would be track evolution of such variables in separate abstract contexts or implement a SSA form for memory in LLVM IR.

4.2.2 Graph Unfolding

After running static taint analysis on all functions in an LLVM module, a complete dependency graph for that module is obtained. However, the graph will contain cycles in cases when Windows API functions are called in a cycle and are dependent on each other. These cycles need to be unfolded before the dependency graph can be used in the subsequent parts of the detector. The usual way to unfold cycles in graphs is to use a breadth-first search or depth-first search algorithm. This would result in a set of trees, but information about call dependencies would be lost by losing certain edges in the graph.

Algorithm 2 was implemented as an alternative way of unfolding dependency graphs. The algorithm first obtains a set of all back edges by running a depth-first search algorithm on a given dependency graph. Each back edge in the set identifies a cycle in the graph. For each back edge in the set, a breadth-first search traversal of the given dependency graph is performed, starting from the end node of the back edge. During the traversal, a copy of the traversed subgraph is made. When the algorithm encounters the starting back edge, it creates an edge that bridges the original graph and the copy, and sets the copy of the starting back edge as the new back edge to be unfolded. The number of times a single back edge is unfolded is given by the factor k. When k unfoldings are reached, the algorithm removes the current back edge and terminates.

Algorithm 2: Unfolder

Input: Graph $G = (V, E)$, back edge (u, v), factor k
Output: Graph G with cycle containing (u, v) unfolded

```
1  foreach 1 ≤ i ≤ k do
2      Queue Q ← {v}, C ← {v}, c ← ∅, n ← ∅, e ← (u, v);
3      while Q ≠ ∅ do
4          a ← Q.dequeue();
5          a′ ← copy(a);
6          foreach node b adjacent to a do
7              b′ ← copy(b);
8              V ← V ∪ {a′, b′};
9              E ← E ∪ {(a′, b′)};
10             if (a,b) = e then
11                 c ← (a, b′);
12                 n ← (a′, b′);
13             end
14             if b ∉ C then
15                 C ← C ∪ {b};
16                 Q.enqueue(b);
17             end
18         end
19     end
20     E ← E \ {e};
21     E ← E ∪ {c};
22     e ← n;
23 end
24 E ← E \ {e};
```

4.3 Signature generation

The core of signature generation are k-testable tree automaton inference algorithms presented in [2] and implemented in tools available at [1]. The algorithm starts by transforming a set of dependency graphs into maximally-shared graphs. This is done by eliminating redundant subgraphs and making nodes share common subgraphs. The resulting graph can also be viewed as a collection of folded trees. Nodes that do not have any predecessors are roots and nodes that have common subtrees are made to share a single copy of the subtree. The authors in [2] reference this transformation as common subexpression elimination.

To simplify description, the following algorithms operate on trees and use formal terminology established in Chapter 3, Sections 3.2. The extension of the algorithms to maximally-shared graphs is done by tracking the visited nodes and making sure that each node is visited once.

Algorithm 3 traverses tree t recursively in postorder. Every subtree has a field *key* associated with its root and the field is initially set to a zero value. The $\oplus$ operator can be any operation that can combine hashes, e.g., a bitwise XOR. The $hash : \mathcal{F} \to \mathbb{N}$ function can be any function used to compute integer hashes from string labels of symbols in the ranked alphabet. When the algorithm is called once, *key* fields of all subtree roots t_s

Algorithm 3: ComputeKey – Computation of k-Root Keys (Hashes)

Input: Tree t, factor k
Output: Key computed for every subtree of t

$tmp \leftarrow hash(t(\varepsilon))$;
foreach $1 \leq i \leq arity(t(\varepsilon))$ **do**
    $t_s \leftarrow t|_i$;
    $tmp \leftarrow tmp \oplus hash(t_s.key)$;
    ComputeKey(t_s, k);
end
$t.key \leftarrow tmp$;

Algorithm 4: k-Testable Tree Automaton Inference

Input: Tree t, factor k, alphabet $\mathcal{F}$
Output: $A^{\sim_k} = (Q, \mathcal{F}, \delta, Q_f)$

$Q \leftarrow \emptyset,\ \delta \leftarrow \emptyset,\ Q_f \leftarrow \emptyset$;
foreach *subtree* $t_s \in \{t|_u \mid u \in dom(t)\}$ *traversed in postorder* **do**
    if $rep[t_s.key] = \emptyset$ **then**
        $q \leftarrow root_k(t_s)$;
        $rep[t_s.key] \leftarrow q$;
        $Q \leftarrow Q \cup \{q\}$;
    end
    $n \leftarrow arity(t_s(\varepsilon))$;
    $\delta_n \leftarrow \delta_n \cup \{((t_s(\varepsilon), rep[(t_s|_1).key], \ldots, rep[(t_s|_n).key]),\ rep[t_s.key])\}$;
    $\delta \leftarrow \delta \cup \delta_n$;
end
$Q_f \leftarrow Q_f \cup \{rep[t.key]\}$;
return $(Q, \mathcal{F}, \delta, Q_f)$

contain a hash of their own symbol only. When called twice, the fields will contain hashes of subtree roots and their children. This way, after k calls, the *key* fields will contain a hash for the k-root of their subtrees. When processing graphs with multiple roots, an auxiliary super-root is created, the original roots are connected to the super-root and the algorithm is called k times with the super-root as the operand.

Algorithm 4 constructs the $A^{\sim_k}$ automaton, by traversing the tree (or maximally-shared graph) t in postorder. The algorithm creates states and transition relations of the automaton using representatives of $\sim_k$-induced equivalence classes stored in the *rep* hash map, with representative of t being the final state. As before, graphs with multiple roots are handled via a super-root.

4.4 Matching

Given an inferred k-testable tree automaton the detector is able to perform matching of an unknown dependency graph. Same as in the signature generation phase, the dependency graph is first transformed into a maximally-shared graph by performing common subexpression elimination. Each tree contained in the maximally-shared graph is then matched

against the inferred automaton. If the automaton accepts, the tree is marked as malicious. After checking all of the trees, a scoring function is used to determine if the input dependency graph, as a whole, is malicious or not. A scoring function can simply be the ratio of accepted trees to the total number of the trees contained in the maximally-shared graph.

$$score = \frac{\sum_i \frac{accepted_i}{total_i} \cdot i}{\sum_i i} \tag{4.1}$$

A more sophisticated scoring function is given in [2] and shown in Equation 4.1. The function takes into account the height of the tree, with trees of greater height being more significantly represented in the score. This is done because malware often executes dependent syscalls in a cycle, which produces dependency graphs containing trees of greater height. Althought the implemented detector produces dependency graphs statically, thus without executing the malware, the cyclic behavior is pronounced by the unfolding performed after static taint analysis. This makes the scoring function applicable for the detector.

Chapter 5

Detector Testing

In this chapter, we describe several experiments that we have performed with our detector. The experiments were based on the following malware samples, which were obtained from various hobbyist internet forums, such as [20]:

- keylogger1: A simple implementation of a keylogger which uses the Windows API function `GetKeyNameText` as its main way of recognizing the pressed key.
- keylogger2: This keylogger uses `GetAsyncKeyState` to get information about the pressed key. It also modifies the operating system registries to ensure that its repeatedly started.
- zwclose-CTS: A file infecting virus with self-encryption capabilities, using a simple XOR cipher. The infector works in a single thread.
- zwclose-ZeroX: An advanced file infector which works in three threads and has also desctructive capabilities.

The detector was tested using test sets of dependency graphs derived from mutations generated by `ollvm` using instruction substitution, control-flow flattening and bogus control flow. For each experiment, 100 mutations were generated. We now describe the particular experiments that we performed.

Experiment 1. In this experiment, we considered keylogger1, keylogger2 and zwclose-ZeroX. Each mutation was generated by using random combinations of the above mentioned obfuscations. The obtained mutations were analyzed by the static taint analysis described in Section 4.2. The resulting syscall dependency graphs were unfolded once. Half of the unfolded graphs were used to infer a k-testable tree automaton, the other half was used for matching as graphs of unknown origin. The detection rate obtained using a simple ratio scoring function was 0.91. When Equation 4.1 was used as the scoring function, the detection rate was 0.96. The success of the detection is thus quite high.

Experiment 2. In the second experiment, we considered the same malware samples as above and we used the same mutation schema. However, this time we did not unfold the dependency graphs. The detection rate obtained using a simple ratio scoring function was 0.72. When Equation 4.1 was used as the scoring function, the detection rate was 0.52. This shows that the graph unfolding has a significant impact on the success of the detector.

Experiment 3. In the third experiment, we considered the keylogger samples. In this case mutations of one keylogger sample were used to infer the tree automaton and the mutations of the other sample were used for matching. No unfolding was performed and the detection rate was 0 using both scoring functions. This can be attributed to the fact that both of the keyloggers use quite different means to implement their functionality. The same result was obtained regardless of which of the keyloggers was used for inference and which for the matching.

Experiment 4. The fourth experiment differs from the third only in applying unfolding. The results do not differ.

Experiment 5. The fifth experiment is again similar to the third experiment, but uses zwclose-ZeroX and zwclose-CTS as malware samples. Unfolding was again not performed. The results are still low, with detection rate being 0.05 / 0.1 using a simple scoring function or 0.02 / 0.03 using the more sophisticated scoring function (depending on which of the two malware samples is used for inference and which for matching). Despite these numbers are still low, they show that there is at least some chance of recognizing similar behavior in two different implementations of malware.

Chapter 6

Conclusion

In this thesis we have studied methods for behavioral malware detection, which use techniques of formal verification. In particular we built on the work [2], which uses inference of k-testable tree automata from syscall dependency graphs. We designed and implemented a prototype detector using the LLVM compiler framework. For experiments with the detector we used an obfuscating compiler capable of generating mutations of malware from C/C++ source code. We performed some preliminary experiments to evaluate capabilities of the detector. The detector was quite successful in recognizing mutations of known malware. However, its capabilities of recognizing unknown malware, which was tested by training the detector on one implementation of malware of a certain kind and then trying the detection on a completely different implementation of the same kind of malware, were rather limited.

In the future, more thorough experimental evaluation should be performed. Apart from that, various improvements of the detector are possible. For instance, one can try to exploit more of the semantics of functions from Windows API. For instance one could recognize whether a given function truly has the semantics of a taint source or sink. Next, one could also employ advanced methods of dead code elimination to de-obfuscate the LLVM IR. Particularly in the case of obfuscations with opaque predicates, this could be of significant help.

Bibliography

[1] Babić, D.: Domagoj Babic's Research Projects. Available on: http://www.domagoj-babic.com/index.php/ResearchProjects/MalwareAnalysis. 2011. accessed: 2016-05-01.

[2] Babić, D.; Reynaud, D.; Song, D.: Malware Analysis with Tree Automata Inference. In *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Berlin, Heidelberg: Springer-Verlag. 2011. ISBN 978-3-642-22109-5. pp. 116–131.

[3] Christodorescu, M.; Jha, S.; Kruegel, C.: Mining Specifications of Malicious Behavior. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. New York, NY, USA: ACM. 2007. ISBN 978-1-59593-811-4. pp. 5–14.

[4] Cohen, F.: *Computer Viruses*. PhD. Thesis. University of Southern California. 1986.

[5] Comon, H.; Dauchet, M.; Gilleron, R.; et al.: Tree Automata Techniques and Applications. Available on: http://www.grappa.univ-lille3.fr/tata. 2007. release October, 12th 2007.

[6] Cousot, P.; Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. New York, NY, USA: ACM. 1977. pp. 238–252.

[7] Filiol, E.: Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*. 2007: pp. 70–75.

[8] Filiol, E.: Malicious cryptology and mathematics. *Cryptography and Security in Computing*. 2012.

[9] García, P.: Learning k-testable tree sets from positive data. Technical Report DSIC-II-46-93. Universidad Politécnica de Valencia. 1993.

[10] García, P.; Vidal, E.: Inference of k-Testable Languages in the Strict Sense and Application to Syntactic Pattern Recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*. vol. 12, no. 9. September 1990: pp. 920–925. ISSN 0162-8828.

[11] Garey, M. R.; Johnson, D. S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.. 1979. ISBN 0716710447.

[12] Gold, E. M.: Complexity of Automaton Identification from Given Data. *Information and Control*. vol. 37, no. 3. June 1978: pp. 302–320.

[13] Guo, F.; Ferrie, P.; Chiueh, T.: A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*. RAID '08. Berlin, Heidelberg: Springer-Verlag. 2008. ISBN 978-3-540-87402-7. pp. 98–115.

[14] Jacob, G.; Debar, H.; Filiol, E.: Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*. vol. 4, no. 3. 2008: pp. 251–266. ISSN 1772-9890.

[15] Junod, P.; Rinaldini, J.; Wehrli, J.; et al.: Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the 2015 IEEE/ACM 1st International Workshop on Software Protection*. SPRO '15. Washington, DC, USA: IEEE Computer Society. 2015. ISBN 978-1-4673-7094-3. pp. 3–9.

[16] Lopes, B. C.; Auler, R.: *Getting Started with LLVM Core Libraries*. Packt Publishing. 2014. ISBN 1782166920.

[17] Lopez, D.; Sempere, J. M.; García, P.: Inference of Reversible Tree Languages. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*. vol. 34, no. 4. August 2004: pp. 1658–1665. ISSN 1083-4419.

[18] Lothaire, M.: *Algebraic combinatorics on words*. Encyclopedia of mathematics and its applications. New York: Cambridge university press. 2002. ISBN 0-521-81220-8.

[19] Nielson, F.; Nielson, H.; Hankin, C.: *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.. 1999. ISBN 978-3540654100.

[20] Rohitab, B.: Rohitab. Available on: http://www.rohitab.com/discuss/. 2012. accessed: 2016-05-01.

[21] Schiffman, M.: A Brief History of Malware Obfuscation. Available on: http://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_1_of_2. 2010. accessed: 2016-04-24.

[22] Sistla, A. P.; Clarke, E. M.: The Complexity of Propositional Linear Temporal Logics. *J. ACM*. vol. 32, no. 3. July 1985: pp. 733–749. ISSN 0004-5411.

[23] Szor, P.: *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional. 2005. ISBN 978-0321304544.

[24] Vojnar, T.; Lengál, O.: Lecture Notes: Formal Analysis and Verificantion. Available on: http://www.fit.vutbr.cz/study/courses/FAV/public/Lectures/fav-lecture-09.pdf. 2016. accessed: 2016-04-01.

[25] You, I.; Yim, K.: Malware Obfuscation Techniques: A Brief Survey. In *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. BWCCA '10. Washington, DC, USA: IEEE Computer Society. 2010. ISBN 978-0-7695-4236-2. pp. 297–300.

Appendices

List of Appendices