



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
BRNO UNIVERSITY OF TECHNOLOGY



**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**  
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**

FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF INTELLIGENT SYSTEMS

**NÁZEV PRÁCE**

THESIS TITLE

**DIPLOMOVÁ PRÁCE**

MASTER'S THESIS

**AUTOR PRÁCE**

AUTHOR

**JMÉNO PŘÍJMENÍ**

**VEDOUcí PRÁCE**

SUPERVISOR

**Doc. RNDr. JMÉNO PŘÍJMENÍ, Ph.D.**

BRNO 2016

## **Abstrakt**

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém jazyce.

## **Abstract**

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

## **Klíčová slova**

Sem budou zapsána jednotlivá klíčová slova v českém jazyce, oddělená čárkami.

## **Keywords**

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

## **Citace**

Jméno Příjmení: Thesis title, diplomová práce, Brno, FIT VUT v Brně, 2016

# Thesis title

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....

Jméno Příjmení

May 12, 2016

## Poděkování

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

© Jméno Příjmení, 2016.

*Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.*

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Malware Detection</b>	<b>5</b>
2.1	Obfuscation and Malware Taxonomy . . . . .	5
2.1.1	Packing, encryption and oligomorphism . . . . .	6
2.1.2	Polymorphism . . . . .	6
2.1.3	Metamorphism . . . . .	8
2.2	Syntactic Detection . . . . .	9
2.2.1	Algorithmic Scanning . . . . .	10
2.2.2	Code Emulation . . . . .	10
2.3	Behavioral Detection . . . . .	10
2.3.1	Simulation-based Detection . . . . .	12
2.3.2	Detection via Formal Verification . . . . .	13
<b>3</b>	<b>Preliminaries</b>	<b>14</b>
3.1	Static Analysis . . . . .	14
3.1.1	Partially Ordered Sets and Lattices . . . . .	14
3.1.2	Abstract Interpretation . . . . .	16
3.2	Tree Automata . . . . .	17
3.2.1	Ranked Alphabets, Terms and Trees . . . . .	17
3.2.2	Finite Tree Automata . . . . .	18
3.2.3	Tree Automata Inference . . . . .	19
3.3	The LLVM Compiler Infrastructure . . . . .	21
3.3.1	Intermediate Representation . . . . .	21
<b>4</b>	<b>Detector Implementation</b>	<b>24</b>
4.1	Mutation Generation . . . . .	25
4.1.1	Obfuscator-LLVM . . . . .	25
4.2	Interpretation . . . . .	26
4.2.1	Static Taint Analysis . . . . .	27
4.2.2	Graph Unfolding . . . . .	28
4.3	Classifier Inference . . . . .	29
4.4	Testing and Results . . . . .	30
<b>5</b>	<b>Conclusion</b>	<b>33</b>
5.1	Discussion . . . . .	33
5.2	Future Work . . . . .	33

<b>Literature</b>	<b>34</b>
<b>Appendices</b>	<b>35</b>
List of Appendices . . . . .	36

# 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 occurrence. 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 development.

*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 response, 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. Finally conclusion with a discussion about choices that were made with possible alternatives and potential for future work is given in chapter 5.

## Chapter 2

# Malware Detection

Early research in the field of computer virology shows that perfectly reliable malware detection is theoretically impossible[?] and more recent research shows that practical malware detection can be made computationally infeasible[?]. Despite these negative results, antivirus software is commercially 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*[?] or *generalization*[?].

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 [?], [?] and [?]. From the detection point of view, the advantages, drawbacks and overall principles behind syntactic and semantic detection will be introduced, taking from [?] and again [?].

## 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 the 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.



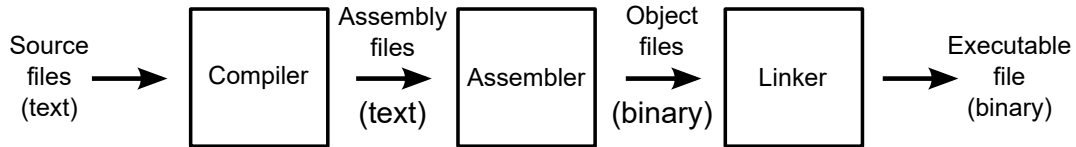


Figure 2.1: Compilation chain of a C-like language

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[?]. 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. Similarly 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 original code. Combined with other obfuscations however, the inserted code may be further transformed and made very hard to recognize as dead.

<pre> INC EBX BSR EAX, EDX TEST EAX, DC78A946 MOV EAX, EDX PUSH EDX MOV DH, 86 MOV BL, 27 </pre>		<pre> INC EBX BSR EAX, EDX TEST EAX, DC78A946 MOV EAX, EDX NOP NOP INC EDX PUSH EDX DEC BYTE PTR SS:[ESP] DEC EDX MOV DH, 86 MOV BL, 27 </pre>
--	--	--

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.

<pre> MOV ESI, EAX MOV AL, BYTE PTR DS:[EAX] TEST AL, AL JE SHORT Test.00401054 PUSH EBX </pre>		<pre> MOV ESI, EBX MOV BL, BYTE PTR DS:[EAX] TEST BL, BL JE SHORT Test.00401054 PUSH EDX </pre>
---	--	---

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.

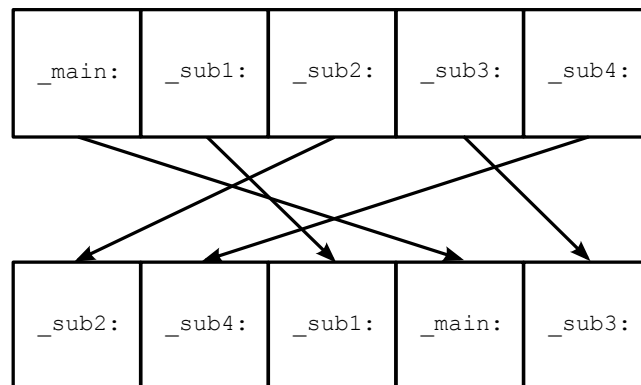


Figure 2.4: Example of subroutine reordering

**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.

<pre> MOV  ESI, EAX MOV  AL,  BYTE PTR DS:[EAX] TEST AL,  AL JE   SHORT Test.00401054 . . . BSR  EAX, EDX TEST EAX, DC78A946 MOV  EAX, EDX </pre>		<pre> MOV  ESI, EAX MOV  AL,  BYTE PTR DS:[EAX] OR   AL,  AL JE   SHORT Test.00401054 . . . BSR  EAX, EDX OR   EAX, DC78A946 MOV  EAX, EDX </pre>
---	--	---

Figure 2.5: Example of instruction substitution

**Code transposition** Similar to subroutine reordreding, 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.

<pre> instruction4 instruction5 jump out garbage _main: instruction1 instruction2 jump 3 garbage instruction3 jump 4 garbage </pre>		<pre> instruction2 jump 3 garbage instruction3 jump 4 garbage instruction5 jump out _main: instruction1 jump 2 instruction4 jump 5 </pre>		<pre> instruction3 instruction4 jump 5 garbage instruction5 jump out _main: instruction1 jump 2 garbage instruction2 jump 3 garbage </pre>
---	--	---	--	--

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.

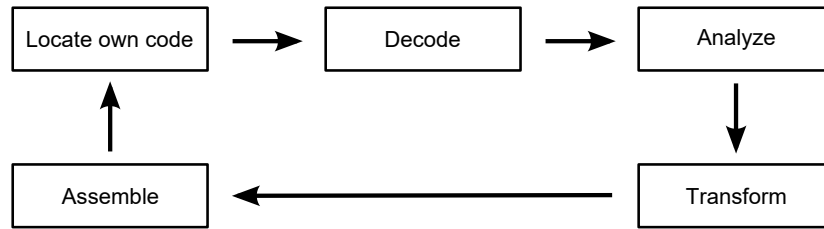


Figure 2.7: Mutation stages of a metamorphic malware

## 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 occurrence 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 benign 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[?].

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.

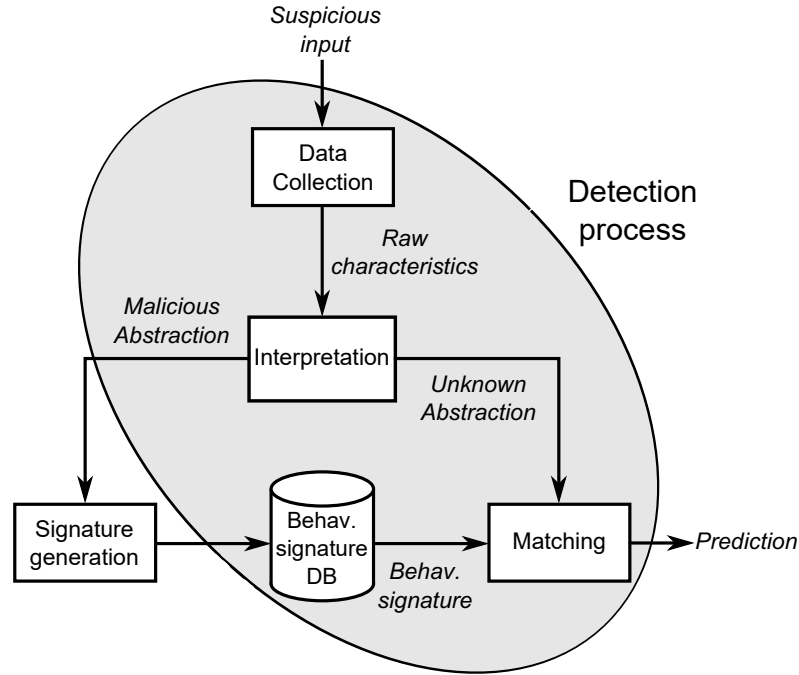


Figure 2.10: A generalized behavioral detector

Figure 2.11 shows a taxonomy of behavioral detectors as presented in [?]. 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.

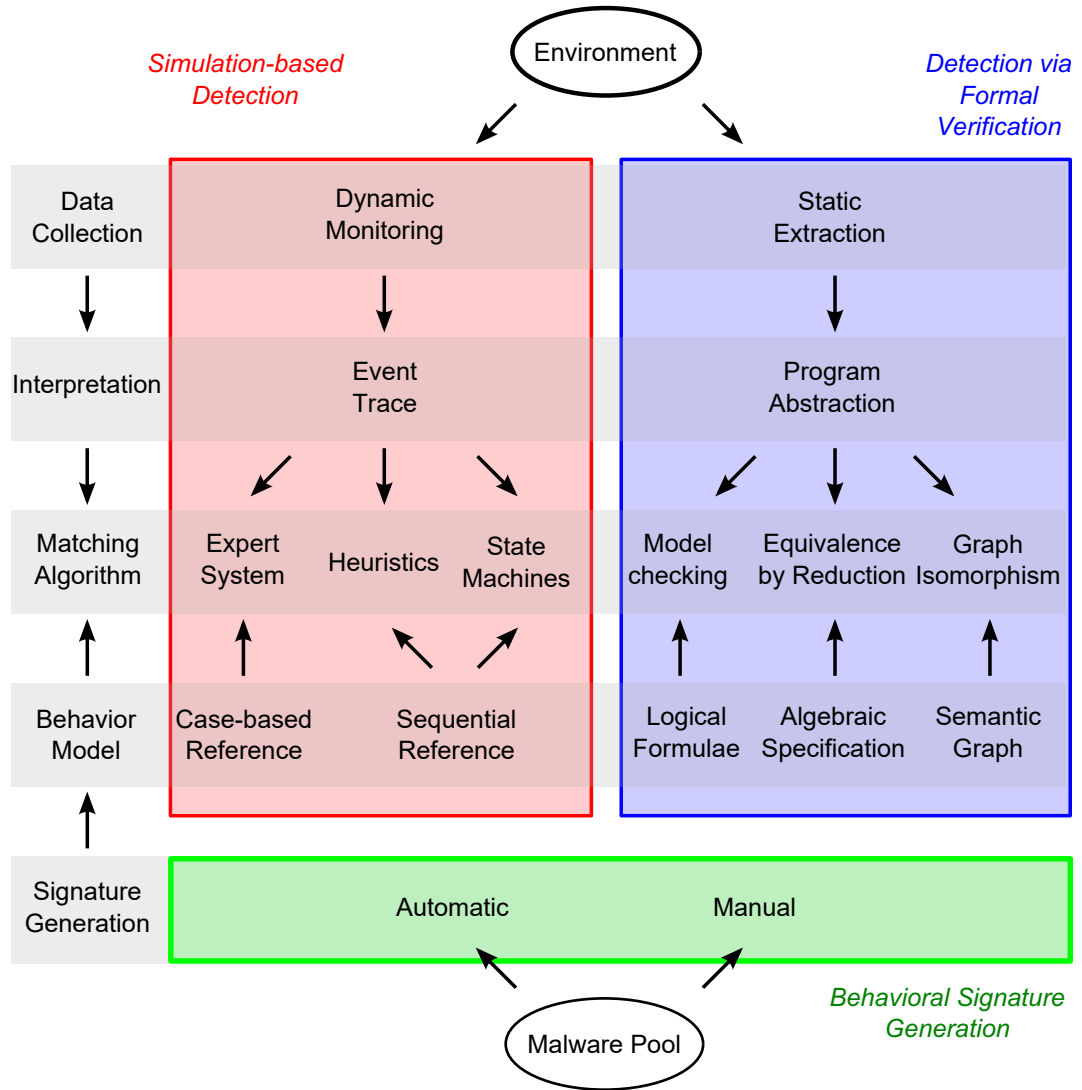


Figure 2.11: A taxonomy of behavioral detectors

### 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 short-sighted 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[?] problem while LTL model checking is a PSPACE-complete problem[?].



## 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 [?] and corresponding chapters from [?]. Parts on tree automata and related notions are based on [?] and [?]. Finally the overview of the LLVM project and its intermediate representation is based on [?].

### 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. Furthermore  $\perp = \sqcup \emptyset = \sqcap L$  is the *least element* and  $\top = \sqcap \emptyset = \sqcup L$  is the *greatest element*. A *lattice*  $(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$  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.

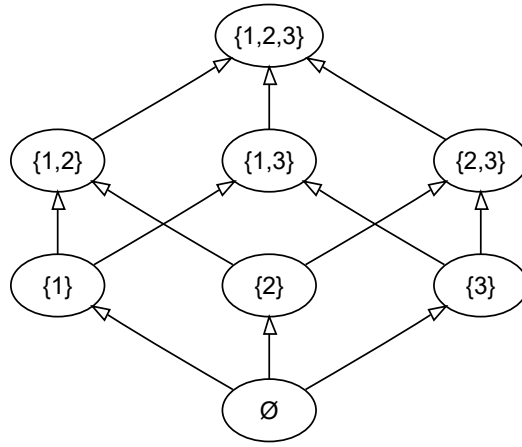


Figure 3.1: A Hasse diagram of the  $(2^{\{1,2,3\}}, \subseteq, \cup, \cap, \emptyset, \{1, 2, 3\})$  lattice

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

**Definition 3.1.6** ( $\sqcup$ -continuous function). Let  $(A, \sqsubseteq_A)$  and  $(B, \sqsubseteq_B)$  be lattices and  $f : A \rightarrow B$  a function. Function  $f$  is *monotone* if  $\forall x, y \in A : x \sqsubseteq_A y \Rightarrow f(x) \sqsubseteq_B f(y)$ . 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  $f : L \rightarrow L$  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  $(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$  be a lattice and  $f : A \rightarrow A$  a  $\sqcup$ -continuous function. The least fixed point of  $f$  is  $\mu f = \sqcup\{f^i(\perp) | i \geq 0\}$ .

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 [?], *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

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

where

- $Q$  is the abstract domain (a set of abstract contexts),
- $\circ : Q \times Q \rightarrow Q$  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  $(Q, \circ, \top)$ ,
- $\perp$  is the least element of  $Q$ ,
- $\top$  is the greatest element of  $Q$ ,
- $\tau : \text{Instr} \times Q \rightarrow Q$  defines an interpretation of abstract transformers.

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

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

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

- $\Delta : Q \times Q \rightarrow Q$ ,
- $\forall C, D \in Q : D \sqsubseteq C \Rightarrow (D \sqsubseteq (C \Delta D) \sqsubseteq C)$ ,
- for all infinite sequences  $(C_0, C_1, \dots, C_n, \dots) \in Q^\omega$ , it holds that the infinite sequence  $(s_0, s_1, \dots, s_n, \dots)$  defined recursively as  $s_0 = C_0$ ,  $s_n = s_{n-1} \Delta C_n$  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  $arity : \mathcal{F} \rightarrow \mathbb{N}$  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  $\mathcal{F}_n = \{f \mid f \in \mathcal{F} \wedge arity(f) = n\}$  is the set of symbols of arity  $n$ . Symbols of arity 0, 1,  $\dots$ ,  $p$  are respectively called constants, unary symbols,  $\dots$ ,  $n$ -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  $n \geq 1$ ,  $f \in \mathcal{F}_n$ ,  $t_1, \dots, t_n \in T(\mathcal{F})$ , then  $f(t_1, \dots, t_n) \in T(\mathcal{F})$ .

**Definition 3.2.3** (*Tree Domain*). Let  $\mathbb{N}$  be the set of natural numbers and  $\mathbb{N}^*$  the *free monoid*<sup>1</sup> generated by  $\mathbb{N}$  with concatenation  $(\cdot)$  as the operation and the empty string  $\varepsilon$  as the identity. The *prefix order*  $(\leq) \subseteq \mathbb{N}^* \times \mathbb{N}^*$  is defined as:  $u \leq v \stackrel{\text{def}}{\iff} \exists w \in \mathbb{N}^* : v = u \cdot w$ . 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  $t : D \rightarrow \mathcal{F}$  satisfying the following:

---

<sup>1</sup>for a definition see [?]

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

The domain of tree  $t$  shall be written as  $\text{dom}(t)$  and each element in  $\text{dom}(t)$  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  $\text{height}(t)$  is defined inductively as

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

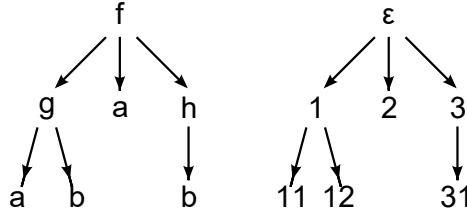


Figure 3.2: An example of a tree  $t$  (left) and its tree domain  $D$  (right).  $\text{height}(t) = 3$

**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:

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

### 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  $\Sigma = \{x, y, z\}$  can 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. Deterministic 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 [?]. 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  $\delta_0 : \mathcal{F}_0 \rightarrow Q$  and  $\delta_n : (\mathcal{F}_n \times Q^n) \rightarrow Q$  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  $\delta_{\mathcal{A}} : T(\mathcal{F}) \rightarrow Q$  describes the run of FDTA  $\mathcal{A}$  on  $t$  recursively as:

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

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

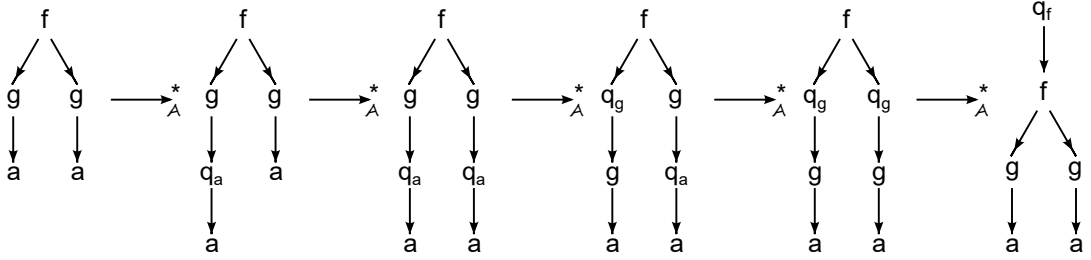


Figure 3.3: An accepting run of a FDTA  $\mathcal{A} = (Q, \mathcal{F}, \delta, Q_f)$  on term  $t = f(g(a), g(a))$ .  $Q = \{q_a, q_g, q_f\}$ ,  $\mathcal{F} = \{f(\cdot, \cdot), g(\cdot), a\}$ ,  $\delta_0(a) = q_a$ ,  $\delta_1(g, q_a) = q_g$ ,  $\delta_2(f, q_g, q_g) = q_f$ ,  $Q_f = \{q_f\}$ .

### 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[?].

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[?]. 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[?].

*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  $\Xi = \{\xi_f \mid f \in \bigcup_{i \geq 0} \mathcal{F}_i\}$  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

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

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  $\xi_f$  on position  $n$  and  $t$  needs to have symbol  $f$  as its root. If these conditions are met, the placeholder symbol in  $\xi_f$  in  $t$  is replaced by  $t'$ . Otherwise the operation does nothing and the result is the unchanged tree  $t$ . An example of a link operation is shown in Figure 3.4.

TODO

Figure 3.4: Link Operation

**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  $t'$ . Figure 3.5 illustrates the construction of a tree quotient. Furthermore, the tree quotient operation can also be extended to sets in the following manner:  $t^{-1}S = \{t^{-1}t \mid t' \in S\}$ .

TODO

Figure 3.5: Tree Quotient

**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:

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

**Theorem 3.2.1** (*K-Testable Tree Language[?]*). 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  $\text{root}_k(t_1) = \text{root}_k(t_2)$ , 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  $\text{root}_k(t_1) = \text{root}_k(t_2)$ . 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:

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

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

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

### 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 3.6 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 3.6: 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.

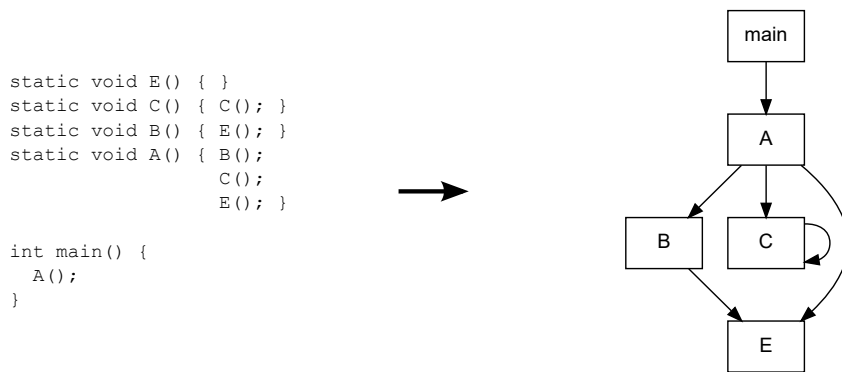


Figure 3.7: An example of a call graph

**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*.

**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  *$\phi$ -instruction* must be present in the basic block before the use of the variable.

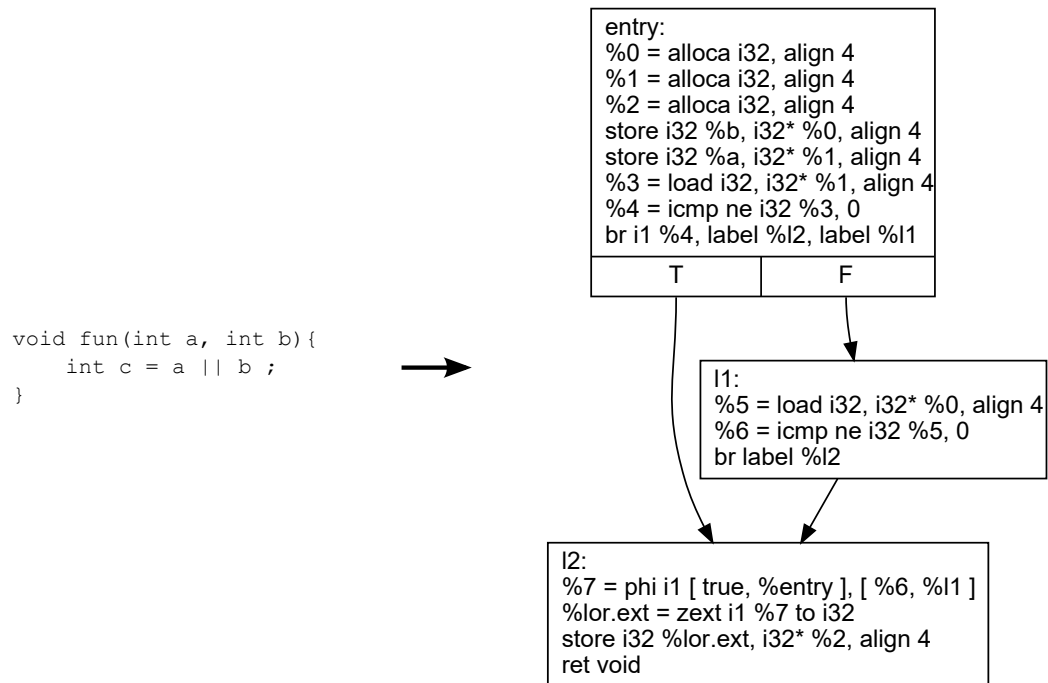


Figure 3.8: Control flow graph with a  $\phi$ -instruction

## 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. Figure 4.1 shows a schema of the implemented detector.

The data collection phase is replaced by an LLVM-based obfuscating compiler [?]. 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.

The interpretation phase is done by analyzing the obfuscated LLVM IR files using *static taint analysis*. The analysis is formalized as an abstract interpretation of LLVM IR code and implemented as a pass over LLVM IR using the tools and libraries available in the LLVM framework. The analysis outputs a *syscall dependency graph* which is used as the high-level abstraction of the malware’s behavior. Section 4.2 covers the details about static taint analysis and syscall dependency graphs.

The signature generation phase is done by inferring a  $k$ -testable tree automaton from a training set of syscall dependency graphs. The matching phase is done by using a  $k$ -testable tree automaton on a syscall dependency graph. Both of the phases use algorithms presented in [?] and are implemented in a research prototype available at [?]. Section 4.3 covers the details about signature generation and matching.

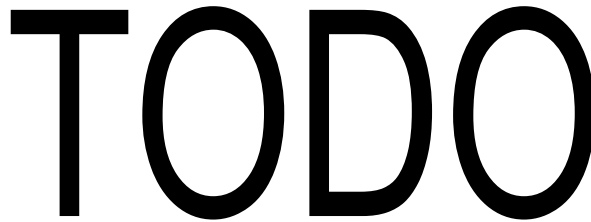
The image consists of the word "TODO" in a very large, bold, black, sans-serif font. The letters are widely spaced and occupy most of the width of the page. This is a placeholder for the schema of the prototype detector.

Figure 4.1: Schema of the prototype detector

## 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 [?] 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`.

TODO

Figure 4.2: Basic Block Splitting

**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.

TODO

Figure 4.3: Control Flow Flattening

**Bogus Control Flow** An advanced form of dead code insertion which makes use of *opaque predicates*. Extra basic blocks are inserted 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 basic 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. In figure 4.4 the original basic block %2 is cloned into %3 and preceded by %1 which contains the opaque predicate. The opaque predicate is  $((x - 1) \cdot x) \bmod 2 = 0 \vee y \leq 10$  and always evaluates as true when  $x$  and  $y$  are integers.

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. This will prove important in the following sections.

TODO

Figure 4.4: Bogus Control Flow

## 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 graph (SDG) is a 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 dependency between the calls. The dependency can be described as „call  $B$  uses data computed from the result of call  $A$ “. Figure 4.5 shows an example of a malware SDG.

SDGs 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 dependencies between them generally do not change. Static taint analysis provides the information needed to build an SDG by tracking the flow of data between two points in code.

TODO

Figure 4.5: Syscall Dependency Graph

### 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  $\mathbf{Var}$  be set of program variables,  $\mathbf{Instr}$  an instruction set,  $\mathbf{Src} \subseteq \mathbf{Instr}$  a set of taint sources and  $\mathbf{Snk} \subseteq \mathbf{Instr}$  a set of taint sinks. *Static taint analysis* is an abstract interpretation

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

where

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

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

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

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

**Taint Sources and Sinks** 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 an SDG, 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 distinguish 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.

**Abstract Contexts** 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.

**Abstract Transformers** 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 on line 18, the algorithm updates a whole program SDG  $G$ . 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 SDG for that module is obtained. However, the SDG is a general directed graph and in most cases will contain cycles that need to be unfolded before the SDG can be used in the signature generation phase of the detector. The easiest way to unfold cycles in graphs is to use a breadth-first search (BFS) or depth-first search (DFS) algorithm. This results in a set of trees, which is ideal for inference of tree automata, but information about call dependencies is lost from the SDG.

TODO

Figure 4.6: Unfolding of an SDG using BFS and the custom unfold

Algorithm 2 was implemented as an alternative way of unfolding cycles in an SDG. The algorithm first uses a depth-first search to find all cycles by finding all back edges in the graph. For each back edge  $e = (u, v)$  a copy of the subgraph reachable from  $v$  is made using a modified BFS. When the algorithm encounters  $e$ , the copy of  $e$  is stored in  $n$  and edge  $c$  is made to connect the copied subgraph to the original graph. The algorithm then

continues to unfold the newly introduced back edge  $n$ .

---

**Algorithm 2:** Unfolder

---

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

```

1 foreach  $1 \leq i \leq k$  do
2   Queue  $Q \leftarrow \{v\}$ ,  $C \leftarrow \{v\}$ ,  $c \leftarrow \emptyset$ ,  $n \leftarrow \emptyset$ ,  $e \leftarrow (u, v)$ ;
3   while  $Q \neq \emptyset$  do
4      $a \leftarrow Q.dequeue()$ ;
5      $a' \leftarrow copy(a)$ ;
6     foreach node  $b$  adjacent to  $a$  do
7        $b' \leftarrow copy(b)$ ;
8        $V \leftarrow V \cup \{a', b'\}$ ;
9        $E \leftarrow E \cup \{(a', b')\}$ ;
10      if  $(a, b) = e$  then
11         $c \leftarrow (a, b')$ ;
12         $n \leftarrow (a', b')$ ;
13      end
14      if  $b \notin C$  then
15         $C \leftarrow C \cup \{b\}$ ;
16         $Q.enqueue(b)$ ;
17      end
18    end
19  end
20   $E \leftarrow E \setminus \{e\}$ ;
21   $E \leftarrow E \cup \{c\}$ ;
22   $e \leftarrow n$ ;
23 end
24  $E \leftarrow E \setminus \{e\}$ ;

```

---

The graph resulting from using Algorithm 2 introduces significant blowup to the SDG as a drawback. The blowup can be justified by the fact that when cycles are unfolded this way, it pronounces the behavior a cycle in the SDG would have at runtime. Figure 4.6 shows the two possible approaches to unfolding side by side.

### 4.3 Classifier Inference

The final part of the detector is the inference of a  $k$ -testable tree automaton classifier. The algorithms for inferring a  $k$ -testable tree automaton from trees and directed acyclic graphs are taken from [?]. The author also provides an implementation of these algorithms at [?]. The detector takes advantage of this implementation and uses it to implement the matching part of the general detector schema. This section will provide a brief description of the algorithms using formal terminology established in Chapter 3, Section 3.2.



# TODO

Figure 4.7: Folding into a maximally-shared graphs

The major advantage of algorithms presented in [?] is the ability to infer  $k$ -testable tree automata from directed acyclic graphs, thus avoiding unfolding to trees, which could cause an exponential blowup of the SDG. The idea of the approach is to eliminate redundant subgraphs in the input SDG and make symbols share subgraphs they agree on. The result is a *maximally-shared graph*. Figure 4.7 shows how a tree and a directed acyclic graph can be folded into a maximally-shared graph. Algorithm 3, which describes the computation of  $k$ -roots on trees, but can be easily extended to work on maximally-shared graphs.

---

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

---

**Input:** Tree  $t$ , factor  $k$

**Output:** Key computed for every subtree of  $t$

```
1  $tmp \leftarrow hash(t(\varepsilon));$ 
2 foreach  $1 \leq i \leq arity(t(\varepsilon))$  do
3    $t_s \leftarrow t|_i;$ 
4    $tmp \leftarrow tmp \oplus hash(t_s.key);$ 
5   ComputeKey( $t_s, k$ );
6 end
7  $t.key \leftarrow tmp;$ 
```

---

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  $hash : \mathcal{F} \rightarrow \mathbb{N}$  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  $t_s$  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. The extension of the algorithm to maximally shared graphs is done by tracking the visited nodes and making sure that each node is visited once. When processing graphs with multiple roots (or a set of graphs), 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 Testing and Results

test set, detection threshold, results

---

**Algorithm 1:** Static Taint Analysis

---

**Input:** Function  $F$ , Empty abstract contexts  $C_{B_1}, \dots, C_{B_n}$ , SDG  $G$ ;

**Output:** Filled abstract contexts  $\{C_{B_1}, \dots, C_{B_n}\}$ , Updated SDG  $G$ ;

```
1 Queue  $W \leftarrow \emptyset$ ;  
2 foreach basic block  $B$  in  $F$  do  
3   |  $W.enqueue(B)$ ;  
4 end  
5 while  $W \neq \emptyset$  do  
6   |  $B \leftarrow W.dequeue()$ ;  
7   |  $C_{new} \leftarrow \emptyset$ ;  
8   | foreach predecessor  $P$  of  $B$  do  
9     |  $C_{new} \leftarrow C_{new} \circ C_P$ ;  
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          |  $C_{new}(O) \leftarrow \emptyset$ ;  
16        | end  
17        | foreach taint  $T$  in  $C_{new}(O)$  do  
18          | add edge  $(T, I)$  to SDG  $G$ ;  
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          |  $C_{new}(O) \leftarrow C_{new}(O) \cup \{I\}$ ;  
26        | end  
27      | end  
28      |  $C_{new}(I) \leftarrow C_{new}(I) \cup \{I\}$ ;  
29    | else  
30      | foreach operand  $O$  of  $I$  do  
31        |  $C_{new}(I) \leftarrow C_{new}(I) \cup C_{new}(O)$ ;  
32      | end  
33    | end  
34  | end  
35  | if  $C_{new} \neq C_B$  then  
36    |  $C_B \leftarrow C_{new}$ ;  
37    | foreach successor  $S$  of  $B$  do  
38      |  $W.enqueue(S)$ ;  
39    | end  
40  | end  
41 end
```

---

---

**Algorithm 4:**  $k$ -Testable Tree Automaton Inference

---

**Input:** Tree  $t$ , factor  $k$ , alphabet  $\mathcal{F}$

**Output:**  $A^{\sim k} = (Q, \mathcal{F}, \delta, Q_f)$

```
1  $Q \leftarrow \emptyset, \delta \leftarrow \emptyset, Q_f \leftarrow \emptyset;$ 
2 foreach subtree  $t_s \in \{t|_u \mid u \in \text{dom}(t)\}$  traversed in postorder do
3   if  $\text{rep}[t_s.\text{key}] = \emptyset$  then
4      $q \leftarrow \text{root}_k(t_s);$ 
5      $\text{rep}[t_s.\text{key}] \leftarrow q;$ 
6      $Q \leftarrow Q \cup \{q\};$ 
7   end
8    $n \leftarrow \text{arity}(t_s(\varepsilon));$ 
9    $\delta_n \leftarrow \delta_n \cup \{((t_s(\varepsilon), \text{rep}[(t_s|_1).\text{key}], \dots, \text{rep}[(t_s|_n).\text{key}]), \text{rep}[t_s.\text{key}])\};$ 
10   $\delta \leftarrow \delta \cup \delta_n;$ 
11 end
12  $Q_f \leftarrow Q_f \cup \{\text{rep}[t.\text{key}]\};$ 
13 return  $(Q, \mathcal{F}, \delta, Q_f)$ 
```

---

## Chapter 5

# Conclusion

what was the work about, interpreting results, goals vs results

### 5.1 Discussion

taint analysis alternatives, alternative post-processing, alternative inference algos

### 5.2 Future Work

pointer analysis, smt solvers as a solution to opaque predicates, more accurate description of winapi calls as taint sources and sinks

# Bibliography

# Appendices

## List of Appendices