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
April 14, 2016

Poděkování

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

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 has 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. 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 it's use. By carefully monitoring usage of the system after deployment, we lower the risk of abuse of errors that were not corrected during developement.

Malware is a term used to describe any software tools 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 it's 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 an *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 it's original functionality or purpose. This also makes detection a per system task if the obfuscations are applied in a randomized manner, since the same malware can differ in some way from system to system. Obfuscations can change the malware on several levels. Obfuscations that primarily change how the malware appears as a file, are called *syntactic*, while those which change how the malware functions while preserving it's intended purpose are referred to as *semantic*. On the other end, antivirus software needs to implement methods that can detect malware despite being obfuscated. As with malware, detectors can be classified as syntactic or *signature based* and semantic or in other words *behavioral*.

The goal of this work is to research applications of formal analysis and verification to behavioral malware detection and present a proof of concept malware detector that applies these methods. 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 development framework is introduced in chapter 4 as a means to implement the mentioned static analyses. The design and implementation of a proof-of-concept detector prototype is described 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[?] and more recent research shows that practical malware detection can be made computationally infeasible[?]. 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*[?] 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 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.

TODO

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

Simple, yet effective methods of obfuscation go no further than changing the way a malware binary file looks. A common example is the use of compression and simple encryption algorithms.

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 i.e 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 it's 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.

TODO

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.

TODO

Figure 2.3: Example of register reassignment

Subroutine reordering Or 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. W95/Zperm is the common example of a virus that uses this obfuscation.

TODO

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.

TODO

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.

TODO

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 it's own code into it and reassembles the host, preserving it's 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 it's MetaPHOR mutation engine.

TODO

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.

TODO

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 malware byte sequence and three byte signatures that match it.

TODO

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. *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 it's running in a controlled environment it can halt any malicious activity and perform bening computations. Or the malware can exhibit it's 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 it's 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 it's 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 a schema of a generic behavioral detector. After initial data collection a higher abstraction is built by interpreting the collected data. This abstraction can now be used to enrich the database of malicious behavior signatures if the input is known to be malicious. If the input is not known to be malicious a matching of the abstraction is done against the database. Finally a prediction whether the input is malicious or not is made. If the input is malicious, it's behavior abstraction can be used to enrich the database further.

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 it's 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 it's 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 behavior 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 against 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

Theoretical Preliminaries

This chapter introduces the theoretical concepts that underlie the methods and algorithms used in the behavioral detector presented in chapter 5. 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 behavior 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 [**?**].

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. Using 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's 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 (*Partial Ordering*)**.** 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* ($\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \ \wedge \ l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$).

Definition 3.1.2 (*Partially Ordered Set*)**.** Let L be a set and $\sqsubseteq \subseteq L \times L$ a partial ordering over L, written as $\sqsubseteq$. A partially ordered set is a tuple $(L, \sqsubseteq)$

Definition 3.1.3 (*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 3.1.4 (*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 3.1.5 (*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

3.1.2 Abstract Interpretation

Pioneered by Patric 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.

TODO

Figure 3.2: An analysis of integer variables using intervals over natural numbers

Definition 3.1.6 (*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

3.2 Tree Automata

3.2.1 Ranked Alphabets, Terms and Trees

3.2.2 Finite Tree Automata

Chapter 4

The LLVM Compiler Framework

4.1 Structure and Design

4.2 Intermediate Representation

Chapter 5

Detector Design and Implementation

5.1 Input generation

5.2 Taint analysis

5.3 Classifier Inference

5.4 Testing and Results

Chapter 6

Conclusion

6.1 Discussion

6.2 Future Work

Bibliography

Appendices

List of Appendices