Hauptseminar: Security -
Zwischen formalen Methoden und Praxis
Malicious code detection

Oliver Arafat (arafat@in.tum.de)
Technische Universität München

Abstract

In any defense mechanism, malicious code detection is a crucial component. To subvert malicious code detectors, e.g. anti-virus software, malicious code writers try to subvert these detectors by obfuscating the malicious code. As testing results surprisingly showed, commercial virus scanners were not able to detect infected binaries which were transformed by applying simple obfuscation techniques. This paper presents an architecture called SAFE (a static analyzer for executables) developed at University of Wisconsin which is resilient to common obfuscation transformations by not relying upon signature matching but model-checking techniques.

Contents

1 Introduction .................................................. 2
  1.1 What is Malicious Code? ................................... 2
  1.2 Classical virus detection technique .......................... 2
  1.3 Obfuscation techniques ..................................... 3

2 SAFE - a static analyzer for executables .................... 5
  2.1 Overview of the architecture ............................... 6
  2.2 Executable loader .......................................... 7
  2.3 Program Annotator ......................................... 8
  2.4 Malicious Code Automaton ................................ 11
  2.5 Detector .................................................. 12

3 Conclusion .................................................. 14
1 Introduction

This chapter will give a short overview about malicious code in general and the classical detection technique most malicious code detectors use i.e. signature matching. Furthermore the most common obfuscation techniques will be presented.

1.1 What is Malicious Code?

Malicious code is a dangerous and omnipresent threat to every participant in the interconnected world of computers. This threat ranges from malicious code that tries to destroy data or even add physical damage to a computer system, to malicious code that spies out private user data or even allowing non privileged users to gain control over a network by subverting the local security policies. According to its propagation method and its goal, malicious code is usually classified into the following categories:

- viruses
- worms
- Trojan horses
- back doors
- spyware

Although SAFE can be used to detect all above listed kinds of malicious code, this paper will focus on the detection of viruses. A computer virus propagates by inserting a copy of itself into a host program. When this program is executed, the first thing that happens is that the virus copy runs and infects more programs. The virus will then trigger its payload, i.e it performs an action to fulfill its goal. This can be something relatively harmless like showing messages or something destructive like deleting the entire hard drive. Then the original program continues to execute.

1.2 Classical virus detection technique

The classical virus detection technique relies upon signature matching, i.e. the anti-virus software looks for a virus specific instruction sequence (the virus signature) in the executable. If this signature is found, the program is likely to be infected. Consider the following example. The Chernobyl/CIH virus is detected by searching for the following fixed virus signature:
The virus signature itself consists of op-codes which correspond to IA-32 instructions. The following code fragment should clarify this. The op-codes are on the left side, the corresponding instructions are on the right side.

```
E8 00000000  call 0h
5B  pop ebx
8D 4B 42  lea ecx, [ebx + 42h]
51  push ecx
50  push eax
50  push eax
0F01 4C 24 FE  sidt [esp - 02h]
5B  pop ebx
```

Intuitively, it is clear that signature matching only yields to working results if the virus code does not change. If it does, the virus signature is likely to change too. Suppose we add a `nop`-instruction just after the first instruction. This instruction has no other effect but incrementing the instruction pointer. This is a simple form of dead-code insertion, an obfuscation technique we are going to examine in more detail in the next section. With `nop` having the op-code `90` this yields to the following signature:

```
E8 00000000  90 5B 8D 4B 42 51 50 50 0F 01 4C 24 FE 5B
```

which is clearly different to the one above and thus a virus scanner relying on a fixed signature matching would not be able to detect this virus. In this special case, a virus scanner could extend its fixed virus signature it is searching for, with regular expressions to detect it.

```
E8 0000 00(90)* 5B(90)* 8D4B 42(90)* 51(90)*
50(90)* 50(90)* 0F01 4C24 FE(90)* 5B(90)*
```

But the sole use of regular expressions will not meet all the requirements necessary to detect all forms of dead-code insertion as do other forms of code obfuscation as we shall see in the next section.

### 1.3 Obfuscation techniques

This section will examine the four most used obfuscation techniques used by virus programmers.

- Dead code insertion
• Code transposition
• Register reassignment
• Instruction substitution

There are other obfuscation techniques too, but they are not used very often as they are quite difficult to implement. Besides, these four techniques are the only one SAFE is supporting at this moment. The next four subsections will examine each technique mentioned above in detail.

1.3.1 Dead-Code insertion

The goal of dead-code insertion is to add code to a program without modifying its behavior. As we have seen in the last section, inserting \textit{nop}-instructions is the simplest example. Other methods include modifying the program state only to restore it immediately. The following example should clarify this. Suppose we’ve got the instruction

\begin{verbatim}
push eax
\end{verbatim}

somewhere in our code. The following instruction sequence is equivalent to the one above:

\begin{verbatim}
inc eax
push eax
dec [esp - 0h]
dec eax
\end{verbatim}

What we are doing here is incrementing the register \textit{eax} by one (we are modifying the program state). Then we push it on the stack. Now we decrement the value on the stack by one and we also decrement the value of the register \textit{eax} by one (we are restoring the original program state). Thus these two code fragments are equivalent. It is important to note that not all dead-code sequences can be detected, as this problem reduces to the question “Is this code sequence equivalent to an empty program?”, which is undecidable according to Rice’s theorem.

1.3.2 Code Transposition

There are two versions of code transposition. The first one randomly shuffles the instructions and then uses unconditional jumps to restore the original control-flow. Anti-virus software that solely
relies upon signature matching is not likely to detect the virus as the instruction order assumed in the virus signature differs from the actual one in the binary image.

The second version of code transposition simply swaps instructions if they are not interdependent. At the moment, SAFE is not supporting this variation of code transposition. It should be mentioned, that the “obfuscation value” gained by the first version is quite small as SAFE uses an intermediate representation of the executable called a Control-Flow-Graph (CFG) which is not sensitive to changes in the control-flow. We will examine the concept of a CFG in more detail in Section 2.2.1 on page 7.

1.3.3 Register Reassignment

Register reassignment replaces the use of one register with another in a specific live range. If a register, say eax, is dead throughout the live range of another register, say ecx, the use of ecx can be replaced with the use of eax. As the op-code of an instruction also depends on its argument - in this case the register - anti-virus software relying solely on signature matching is not likely to find a matching pattern. For instance, the instruction `push eax` has the op-code 50, while `push ecx` has the op-code 51. As we shall see in the second chapter, due to the use of so-called uninterpreted symbols, SAFE is resistant against register reassignment.

1.3.4 Instruction Substitution

Instruction substitution uses a dictionary of instruction sequences that are equivalent to each other. If an instruction sequence A is found in the code, it can be replaced with an equivalent instruction sequence B found in the dictionary (if there is any). As the instruction set of the IA-32 assembly language is very rich, there are of course many opportunities to perform the same operation. To detect malicious code in executables obfuscated by instruction sequence, the anti-virus software must maintain a dictionary as well, similar to that one used to obfuscate the code. This may not be the best solution, but according to [1], it can cope with common cases. In addition, a theorem prover like “Simplify” or “PVS” can sometimes also be used to prove that two instruction sequences are equivalent.

2 SAFE - a static analyzer for executables

This second chapter will give a detailed view on the malicious code detection tool called SAFE developed at University of Wisconsin (see Figure 1). It will
first give a short overview on the general architecture of SAFE, followed by an in depth view of each component and formal definitions associated with each of it.

2.1 Overview of the architecture

In order to detect malicious code patterns in executables, SAFE basically needs three kinds of input. The first one is an abstract representation of the malicious code we are searching for, that is a so-called malicious code automaton. The second kind of input is the binary executable in which we are searching for malicious code. The third one is a finite set of abstraction patterns which will be used as alphabet symbols for the malicious code automaton. Each of these input types will be discussed in detail in the subsequent sections. I will now give a rough description of each component of SAFE.

Malicious code automaton

The malicious code is generalized into an malicious code automaton (MCA) with uninterpreted symbols. Through the use of uninterpreted symbols one does not need to specifically refer to the storage location of each variable. Apart from that, the set of alphabet symbols consists of abstraction patterns. Intuitively, it should be clear that if and only if the MCA is in an final state at the end of the detection algorithm, a virus was most likely found in the executable.
Pattern definition loader
These component inputs a finite set of abstraction patterns and creates an internal representation of them. As mentioned above, these patterns are used as alphabet symbols by the malicious code automaton.

The executable loader
This component transforms the binary executable into an abstract representation, a so-called control-flow-graph (CFG). In the next subsection, we will examine a CFG in more detail. The executable loader uses two commercial products, IDA Pro and CodeSurfer that both perform a variety of static analyses which is necessary in order to create a CFG. This paper, however, will not discuss the functionality of these two components.

The annotator
The annotator inputs a CFG and transforms it in an annotated CFG using abstraction patterns. That is, an annotated CFG includes information, where exactly in the binary executable a specific abstraction pattern was found.

The detector
The detector decides whether malicious code, represented by the MCA, was found in the executable or not. That is the case, if the MCA is in a finite state. The algorithm used by the detector is based upon language containment and unification. A detailed discussion about that algorithm can be found in Subsection 2.5.1 on page 13.

2.2 Executable loader
As mentioned above, the executable loader creates an abstract representation of the binary executable, a so-called control-flow-graph. We will now examine a CFG in more detail in the following subsection.

2.2.1 Control-Flow-Graph
Figure 2 should give a good impression on what a control-flow-graph is. Formally, the CFG of a program $P$ is defined as $CFG(P) = \langle V, E \rangle$. $V$ is the set of all basic blocks. A basic block $B$ is the maximal sequence of instructions that contain at most one control-flow instruction, e.g. conditional jumps, which must appear at the end of each block. Therefore, the execution within a basic block is sequential by definition. Furthermore, each basic block has at least one element. $E \subseteq V \times V \times \{T, F\}$ is the set of all control flow transitions between basic blocks. Corresponding to the condition of the control-flow instruction, each edge is marked with either $T(\text{true})$ or $F(\text{false})$. Unconditional jumps have outgoing edges always marked with $T$. 
2.3 Program Annotator

The program annotator inputs the CFG created by the executable loader and the finite set of abstraction patterns and outputs an annotated CFG. Each node $n$ of the CFG is associated with one or more abstraction patterns that match the “node sequence” $\langle \ldots, \text{previous}^2(n), \text{previous}(n), n \rangle$. Figure 3 shows the annotated CFG corresponding to the CFG from figure 2. The following subsection will focus on the exact definitions needed to understand the semantics of matching as well as the formal syntax of an abstraction pattern. A formal view on the annotator operation will be provided in Section 2.3.2 on page 11.

2.3.1 Definitions

This section will focus on the formal definitions of abstraction patterns, uninterpreted symbols and bindings and their meanings in terms of semantics of matching.

**Abstraction patterns**

An abstraction pattern $\Gamma$ is a 3-tuple $(V, O, C)$, where $V$ is a list of typed variables, $O$ is a sequence of instructions and $C$ is a boolean expression combining one or more static analysis predicates over program points. To see, what kind of predicates SAFE supports, I refer the reader to the paper this one is based on, i.e. “Static analysis of executables to detect malicious pat-
terns” by Mihai Christodorescu and Somesh Jha. The typed variable system in SAFE offers the possibility during the static analysis process to determine, what kind of type $\tau$ a register or an argument of an instruction represents at runtime. For instance, suppose we have the instruction

\texttt{lea ecx, [ebx + 42h]}

The type of \texttt{ecx} in this case is denoted by $ecx : \bot(32)$ as all we know is that ecx can have any value consisting of n bits, whereas the type of \texttt{[ebx + 42h]} is denoted by $ebx : ptr \bot(32)$ as this argument represents a pointer to a specific memory location. Let us consider a second example.

\texttt{add ebx, 1Ch}

The type of \texttt{ebx} is denoted by $ebx : int(0 : 1 : 31)$ as we add a 32-bit integer to the register ebx. The first number in brackets describes the number of highest bytes that are ignored, the second number are the number of bytes that represent the sign whereas the last number represents the number of lowest bits that represent the value. For a complete overview of the complete type system of SAFE, I refer the reader to [1]. Back to our abstraction pattern which is defined as followed:

\[ V = \{ x_1 : \tau_1, \ldots, x_k : \tau_k \} \]
\[ O = \langle I(v_1, \ldots, v_m) \mid I : \tau_1 \times \ldots \times \tau_m \rightarrow \tau \rangle \]
\[ C = \text{boolean expression involving static analysis predicates and logical operators} \]

Note that each argument of an instruction can be either a literal value or a free variable $x_j$. Let us consider the following example of an abstraction pattern.
\[ \Gamma(X : \text{int}(0 : 1 : 31)) = (\{X : \text{int}(0 : 1 : 31)\}, \langle p1 : \text{pop}\ X, p2 : \text{add}\ X, 03\text{AFh} \rangle, p1 \in \text{LiveRangeStart}(p2, X)) \]

This pattern represents an instruction sequence where a register \( X \) is popped off the stack and then a constant value (03AFh) is added to it. In this very case, the first argument of the second instruction is \( X \) which is a free variable, whereas the second argument 03AFh is a literal value. The type of \( X \) is an integer with 31-bit of storage and 1 sign bit. The \( X \) represents a so-called uninterpreted symbol.

**Uninterpreted symbols**

Through the use of uninterpreted symbols, the pattern above matches any instance of this pattern where \( X \) is assigned a concrete register, that is it can match multiple sequences of instructions. A pattern is instantiated when it is assigned to a node in the control-flow-graph. In other words, when matching an abstraction pattern against a sequence of instructions, unification is used to bind the free variables of \( \Gamma \) to actual values. Note that the obfuscation method register reassignment has no real obfuscation value anymore because of the use of uninterpreted symbols.

**Binding**

A binding is defined as a set of pairs which consist of an uninterpreted symbol and its concrete value, i.e a register or an address. Two bindings \( B_1 \) and \( B_2 \) are said to be compatible if they do not bind the same uninterpreted symbol to different values. Formally this is defined as

\[
\text{Compatible}(B_1, B_2) = \forall x \in V. ([x, y_1] \in B_1 \land [x, y_2] \in B_2) \Rightarrow (y_1 = y_2)
\]

Another definition we will need for the explanation of the detection algorithm discussed in Section 2.5.1 on page 13 is the union of two compatible bindings \( B_1 \) and \( B_2 \). The union of \( B_1 \) and \( B_2 \) includes all the pairs from both bindings. If they are not compatible, the union operations returns an empty binding. As mentioned above, when matching an abstraction pattern against a sequence of instructions unification is used. This is denoted by the function \( \text{Unify}(S(n), \Gamma) \) with \( S(n) \) being defined as

\[
S(n) = \langle \ldots, \text{previous}^2(I_p), \text{previous}(I_p), I_p \rangle. \quad I_p \text{ is the instruction at program point } p. \quad \text{Unify returns a binding } B \text{ if } S(n) \text{ can be unified with the sequence of instructions } O \text{ specified in the pattern } \Gamma, \text{ otherwise } \text{Unify returns false.}
\]
2.3.2 Annotator operation

The annotator associates a set of matching patterns with each node in the CFG. An annotation is a pair \([\Gamma, B]\) and is unique in the annotation set of a given node. A node \(n\) in the CFG is associated with the list of pairs of patterns and bindings that satisfies the following condition:

\[
Annotation(n) = \{[\Gamma, B] : \Gamma \in \{\Gamma_1, ..., \Gamma_m\} \land B = Unify(S(n), \Gamma)\}
\]

If \(Unify(S(n), \Gamma)\) returns false, that is because unification is not possible, then the node \(n\) is not annotated with \([\Gamma, B]\). An example of an annotated CFG is shown in figure 3.

2.4 Malicious Code Automaton

The malicious code automaton (MCA) is an abstract representation of the malicious code. The MCA \(A\) is defined as a 6-tuple \((V, \Sigma, S, \delta, S_0, F)\), where

- \(V = \{v_1 : \tau_1, ..., v_k : \tau_k\}\) is a set of typed variables
- \(\Sigma = \{\Gamma_1, ..., \Gamma_n\}\) is a finite alphabet of abstraction patterns
- \(S\) is a finite set of states
- \(\delta : S \times \Sigma \rightarrow 2^S\) is a transition function
Figure 4: An example of a malicious code automaton (MCA) with two different instantiations

- $S_0 \subseteq S$ is a non-empty set of initial states
- $F \subseteq S$ is a non-empty set of final states

As we can see, a MCA is a finite state automaton in which the alphabet symbols are a finite set of abstraction patterns defined over the set of typed variables. A concrete instantiation of an MCA $A$ with a binding $B$ is denoted by $B(A)$. In other words, all variables in the set $V = \{v_1, \ldots, v_k\}$ are substituted by the concrete values of the binding $B$. Figure 4 shows an example of a MCA and its corresponding instantiations with two different register assignments. An interesting point is the role of the set of abstraction patterns $\Sigma$; they are used both to construct an annotated $CFG$ and as the alphabet symbols of the malicious code automaton.

### 2.5 Detector

The detector inputs the annotated $CFG$ of program $P$ denoted by $P_\Sigma$ and the malicious code automaton $A$. If the pattern described by the MCA $A$ is found in the annotated $CFG$ $P_\Sigma$, the detector returns $true$ as well as the malicious code trace found in the program. If the pattern cannot be found in $P_\Sigma$, the detector returns $false$. Let us examine the above notion in a more formal
way. We can think of the annotated $\text{CFG} \ P_\Sigma$ as a finite-state automaton, where nodes are states, edges represent transitions, the node corresponding to the entry point is the initial state, and every node is a final state. Thus, both automata, namely the MCA $A$ and the annotated $\text{CFG} \ P_\Sigma$, represent a language, i.e $L(P_\Sigma)$ and $L(A)$ respectively. The detector now determines whether the following language is empty:

$$L(P_\Sigma) \cap \left[ \bigcup_{B \in B_{\text{All}}} L(B(A)) \right]$$

$B_{\text{All}}$ is the set of all bindings to the variables in the set $V$. Thus the detector determines, if there exists a binding $B$ such that the intersection of the two languages $L(P_\Sigma)$ and $L(B(A))$ is non-empty, i.e malicious code was found. The following subsection will examine the algorithm used by the detector in detail.

### 2.5.1 The Algorithm

Each node $n$ in the annotated $\text{CFG} \ P_\Sigma$ is associated with a pre and post list $L_{n}^{\text{pre}}$ and $L_{n}^{\text{post}}$ respectively. Every element of a list is a pair $[s, B]$, where $s$ is the state of the MCA $A$ and $B$ the binding of variables. If $[s, B] \in L_{n}^{\text{pre}}$ then it is possible for the MCA $A$ to be in state $s$ with the binding $B$ just before the node $n$. Analogous, if $[s, B] \in L_{n}^{\text{post}}$ then it is possible for the MCA $A$ to be in state $s$ just after the node $n$. We can divide the algorithm in two stages. The initial-stage and the update-stage.

#### Initial-Stage

Initially all pre- and post-lists are empty, except the pre-list of the initial node $n_0$ of the annotated $\text{CFG} \ P_\Sigma$. $L_{n_0}^{\text{pre}}$ is the list of all pairs $[s, B]$ where $s \in S_0$ of the MCA $A$. In every initial pair, the binding $B$ is set to the empty set. The meaning $B = \emptyset$ is that every binding to the variables in the set $V$ is possible.

#### Update-Stage

What is happening here, is the update of all pre- and post-lists of every node $n$ in $P_\Sigma$ until a fixed point is reached, that is when no list is changing anymore. Thus the update-stage is realized as a do-until loop that runs until the fixed point is reached. The first step in this loop is the update of all pre-lists. For a pre-list $L_i^{\text{pre}}$ the union of all pre-lists $L_j^{\text{pre}}$ preceding $i$ is copied to $L_i^{\text{pre}}$ if there are no repeated states. In case of repeated states the two conflicting pairs are merged into a single pair if and only if both bindings are compatible. If
these two bindings are not compatible, both pairs are thrown out.

The next step now is the update of all post-lists of every node $n$. First, we create a new post-list denoted by $\text{New}L_n^{post}$ which is initially empty. For every pair $[s, B_s] \in L_n^{pre}$ now, we do the following steps. For every annotation $[\Gamma, B]$ of a node $n$ of the annotated $CFG \ P_{\Sigma}$, we check if the binding $B_s$ (the one of the post-list) is compatible with the binding $B$ (the one of the annotation). If this is the case, we can follow a transition in the MCA $A$. Thus the following pair is added to $\text{New}L_n^{post}$:

$$[\delta(s, \Gamma), B_s \cup B]$$

Finally, if $\text{New}L_n^{post}$ is now different to $L_n^{post}$, $\text{New}L_n^{post}$ is copied to the original postlist associated with node $n$.

After the fixed point is reached we can now determine if malicious code was found. This is the case, if there exists a pair $[s, B_s]$ in any post-list, where $s$ is a final state of the MCA $A$. In other words, the union of both languages, i.e. the MCA $A$ and the language the annotated $CFG \ P_{\Sigma}$ represents, is non-empty.

The complete algorithm to check a program model against malicious code specifications written in pseudo code can be found in [1].

3 Conclusion

This paper presented an overview of the malicious code detection tool SAFE, developed at University of Wisconsin, that does not rely on the classical virus detection technique, i.e. signature matching. As we have seen at the beginning of this paper, signature matching is not resilient against most obfuscation techniques virus writers use to subvert malicious code detection tools.

As experiments showed, three commercial virus-scanners were not able to detect infected binaries that were obfuscated by dead-code insertion and code transposition. In contrast, SAFE was able to detect all infected binaries as it does not rely upon signature matching but techniques we described earlier in this paper. In terms of security, SAFE surely would be the most suitable malicious code detector in any defense mechanism. But in terms of speed, SAFE is clearly inferior to most malicious code detection tools. Suppose a file with a size of 1 MB. As experiments have showed, the annotation of this file takes about 800 seconds, whereas the detector takes again 160 seconds. This may have something to do with the fact, that the algorithm used by the detector, which combines the classic algorithm for computing the intersection of two regular languages with unification, is $NP$-complete. Despite of
SAFE’s superior detection rate, its bad performance questions the usability in any defense mechanism.

References