Published on 10 February 2003 as Technical Report # 1467 at the Computer Sciences Department of the University of Wisconsin, Madison, US.

Static Analysis of Executables to Detect Malicious Patterns

Mihai Christodorescu and Somesh Jha
`{mihai,jha}@cs.wisc.edu`

10 February 2003

Abstract

Malicious code detection is a crucial component of any defense mechanism. In this paper, we present a unique viewpoint on malicious code detection. We regard malicious code detection as an obfuscation-deobfuscation game between malicious code writers and researchers working on malicious code detection. Malicious code writers attempt to obfuscate the malicious code to subvert the malicious code detectors, such as anti-virus software. We tested the resilience of three commercial virus scanners against code obfuscation attacks. The results were surprising: the three commercial virus scanners could be subverted by very simple obfuscation transformations! We present an architecture for detecting malicious patterns in executables that is resilient to common obfuscation transformations. Experimental results demonstrate the efficacy of our prototype tool, SAFE (a static analyzer for executables).

1 Introduction

In the interconnected world of computers, malicious code has become an omnipresent and dangerous threat. Malicious code can infiltrate hosts using a variety of methods such as attacks against known software flaws, hidden functionality in regular programs, and social engineering. Given the devastating effect malicious code has on our cyber infrastructure, identifying malicious programs is an important goal. Detecting the presence of malicious code on a given host is a crucial component of any defense mechanism.

Malicious code is usually classified [29] according to its propagation method and goal into the following categories:

- *viruses* are programs that self-replicate within a host by attaching themselves to programs and/or documents that become carriers of the malicious code;
- *worms* self-replicate across a network;
- *trojan horses* masquerade as useful programs, but contain malicious code to attack the system or leak data;
- *back doors* open the system to external entities by subverting the local security policies to allow remote access and control over a network;
- *spyware* is a useful software package that also transmits private user data to an external entity.

Combining two or more of these malicious code categories can lead to powerful attack tools. For example, a worm can contain a payload that installs a back door to allow remote access. When the worm replicates to a new system (via email or other means), the back door is installed on that system, thus providing an attacker with a quick and easy way to gain access to a large set of hosts. Staniford *et. al.* have demonstrated that worms can propagate extremely quickly through a network, and thus potentially cripple the entire cyber infrastructure [40]. In a recent outbreak, the Sapphire/Slammer worm reached the peak infection rate in about 10 minutes since launch, doubling every 8.5 seconds [30]. Once the back-door tool gains a large installed base, the attacker can use the compromised hosts to launch a coordinated attack, such as a distributed denial-of-service (DDoS) attack [5].

In this paper, we develop a methodology for detecting malicious patterns in executables. Although our method is general, we have initially focused our attention on viruses. A computer virus replicates itself by inserting a copy of its code (the *viral code*) into a host program. When a user executes the infected program, the virus copy runs, infects more programs, and then the original program continues to execute. To the casual user, there is no perceived difference between the clean and the infected copies of a program until the virus activates its malicious payload.

The classic virus-detection techniques look for the presence of a virus-specific sequence of instructions (called a *virus signature*) inside the program: if the signature is found, it is highly probable that the program is infected. For example, the Chernobyl/CIH virus is detected by checking for the hexadecimal sequence [44]:

```
E800   0000   005B   8D4B   4251   5050
0F01   4C24   FE5B   83C3   1CFA   8B2B
```

This corresponds to the following IA-32 instruction sequence, which constitutes part of the virus body:

```
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
83 C3 1C            add ebx, 1Ch
FA                  cli
8B 2B               mov ebp, [ebx]
```

This classic detection approach is effective when the virus code does not change significantly over time. Detection is also easier when viruses originate from the same source code, with only minor modifications and updates. Thus, a virus signature can be common to several virus variants. For example, Chernobyl/CIH versions 1.2, 1.3, and 1.4 differ mainly in the trigger date on which the malicious code becomes active and can be effectively detected by scanning for a single signature, namely the one shown above.

The virus writers and the antivirus software developers are engaged in an *obfuscation-deobfuscation* game. Virus writers try to obfuscate the "vanilla" virus so that signatures used by the antivirus software cannot detect these "morphed" viruses. Therefore, to detect an obfuscated virus, the virus scanners first must undo the obfuscation transformations used by the virus writers. In this game, virus writers are obfuscators and researchers working on malicious code detection are deobfuscators. A method to detect malicious code should be resistant to common obfuscation transformations. This paper introduces such a method. The main contributions of this paper include:

The obfuscation-deobfuscation game and attacks on commercial virus scanners
We view malicious code detection as an obfuscation-deobfuscation game between the virus writers and the researchers working to detect malicious code. Background on some common obfuscation techniques used by virus writers is given in Section 3. We also have developed an obfuscator for executables. Surprisingly, the three commercial virus scanners we considered could be easily thwarted by simple obfuscation transformations (Section 4). For example, in some cases the Norton antivirus scanner could not even detect insertions of `nop` instructions.

A general architecture for detecting malicious patterns in executables
We introduce a general architecture for detecting malicious patterns in executables. An overview of the architecture and its novel features is given in Section 5. External predicates and uninterpreted symbols are two important elements in our architecture. External predicates are used to summarize results of various static analyses, such as points-to and live-range analysis. We allow these external predicates to be referred in the abstraction patterns that describe the malicious code. Moreover, we allow uninterpreted symbols in patterns, which makes the method resistant to renaming, a common obfuscation transformation. Two key components of our architecture, *the program annotator* and *the malicious code detector*, are described in Sections 6 and 7 respectively.

Prototype for x86 executables
We have implemented a prototype for detecting malicious patterns in x86 executables. The tool is called a *static analyzer for executables* or *SAFE*. We have successfully tried SAFE on multiple viruses; for brevity we report on our experience with four specific viruses. Experimental results (Section 8) demonstrate the efficacy of SAFE. There are several interesting directions we intend to pursue as future work, which are summarized in Section 9.

Extensibility of analysis
SAFE depends heavily on static analysis techniques. As a result, the precision of the tool directly depends on the static analysis techniques that are integrated into it. In other words, *SAFE is as good as the static analysis techniques it is built upon*. For example, if SAFE uses the result of points-to analysis, it will be able to track values across memory references. In the absence of a points-to analyzer, SAFE makes the conservative assumption that a memory reference can access any memory location (i.e. everything points to everything). We have designed SAFE so that various static analysis techniques can be readily integrated into it. Several simple static analysis techniques are already implemented in SAFE.

2 Related Work

2.1 Theoretical Discussion

The theoretical limits of malicious code detection (specifically of virus detection) have been the focus of many researchers. Cohen [10] and Chess-White [9] showed that in general the problem of virus detection is undecidable. Similarly, several important static analysis problems are undecidable or computationally hard [27, 34].

However, the problem considered in this paper is slightly different than the one considered by Cohen [10] and Chess-White [9]. Assume that we are given a vanilla virus V which contains a malicious sequence of instructions σ. Next we are given an obfuscated version $\mathcal{O}(V)$ of the virus. The problem is to find whether there exists a sequence of instructions σ' in $\mathcal{O}(V)$ which is "semantically equivalent" to σ. A recent result by Vadhan *et. al.* [3] proves that in general program obfuscation is impossible. This leads us to believe that a computationally bounded adversary will not be able to obfuscate a virus to completely hide its malicious behavior. We will further explore these theoretical issues in the future.

2.2 Other Detection Techniques

Our work is closely related to previous results on static analysis techniques for verifying security properties of software [1, 4, 8, 7, 24, 28]. In a larger context, our work is similar to existing research on software verification [2, 13]. However, there are several important differences. First, viewing malicious code detection as an obfuscation-deobfuscation game is unique. The obfuscation-deobfuscation viewpoint lead us to explore obfuscation attacks upon commercial virus scanners. Second, to our knowledge, all existing work on static analysis techniques for verifying security properties analyze source code. On the other hand, our analysis technique works on executables. In certain contexts, such as virus detection, source code is not available. Finally, we believe that using uninterpreted variables in the specification of the malicious code is unique (Section 6.2).

We plan to enhance our framework by using the ideas from existing work on type systems for assembly code. We are currently investigating Morrisett *et. al.*'s *Typed Assembly Language* [31, 32]. We apply a simple type system (Section 6) to the binaries we analyze by manually inserting the type annotations. We know of no compiler that can produce Typed Assembly Language, and thus we plan to support external type annotations to enhance the power of our static analysis.

Dynamic monitoring can also be used for malicious code detection. Cohen [10] and Chess-White [9] propose a virus detection model that executes code in a sandbox. Another approach rewrites the binary to introduce checks driven by an enforceable security policy [17] (known as the *inline reference monitor* or the *IRM* approach). We believe static analysis can be used to improve the efficiency of dynamic analysis techniques, e.g., static analysis can remove redundant checks in the IRM framework. We construct our models for executables similar to the work done in specification-based monitoring [20, 43], and apply our detection algorithm in a context-insensitive fashion. Other research used context-sensitive analysis by employing push-down systems (PDSs). Analyses described in [7, 24] use the model checking algorithms for pushdown systems [18] to verify security properties of programs. The data structures used in interprocedural slicing [23], interprocedural DFA [38], and Boolean programs [2] are hierarchically structured graphs and can be translated to pushdown systems.

2.3 Other Obfuscators

While deciding on the initial obfuscation techniques to focus on, we were influenced by several existing tools. *Mistfall* (by *z0mbie*) is a library for binary obfuscation, specifically written to blend malicious code into a host program [46]. It can encrypt, morph, and blend the virus code into the host program. Our binary obfuscator is very similar to Mistfall. Unfortunately, we could not successfully morph binaries using Mistfall, so we could not perform a direct comparison between our obfuscator and Mistfall. *burneye* (by *TESO*) is a Linux binary encapsulation tool. burneye encrypts a binary (possibly multiple times), and packages it into a new binary with an extraction tool [42]. In this paper, we have not considered encryption based obfuscation techniques. In the future, we will incorporate encryption based obfuscation techniques into our tool, by incorporating or extending existing libraries.

3 Background on Obfuscating Viruses

To detect obfuscated viruses, antivirus software have become more complex. This section discusses some common obfuscation transformations used by virus writers and how antivirus software have historically dealt with obfuscated viruses.

A *polymorphic virus* uses multiple techniques to prevent signature matching. First, the virus code is encrypted, and only a small in-clear routine is designed to decrypt the code before running the virus. When the polymorphic virus replicates itself by infecting another program, it encrypts the virus body with a newly-generated key, and it changes the decryption routine by generating new code for it. To obfuscate the decryption routine, several transformations are applied to it. These include: `nop`-insertion, code transposition (changing the order of instructions and placing jump instructions to maintain the original semantics), and register reassignment (permuting the register allocation). These transformations effectively change the virus signature (Figure 1), inhibiting effective signature scanning by an antivirus tool.

Original code

```
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
83 C3 1C        add ebx, 1Ch
FA              cli
8B 2B           mov ebp, [ebx]
```

Obfuscated code

```
E8 00000000     call 0h
5B              pop ebx
8D 4B 42        lea ecx, [ebx + 45h]
90              nop
51              push ecx
50              push eax
50              push eax
90              nop
0F01 4C 24 FE   sidt [esp - 02h]
5B              pop ebx
83 C3 1C        add ebx, 1Ch
90              nop
FA              cli
8B 2B           mov ebp, [ebx]
```

Signature

```
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
```

New signature

```
E800 0000 005B 8D4B 4290 5150
5090 0F01 4C24 FE5B 83C3 1C90
FA8B 2B
```

Figure 1: Original code and obfuscated code from Chernobyl/CIH, and their corresponding signatures. Newly added instructions are highlighted.

The obfuscated code in Figure 1 will behave in the same manner as before since the `nop` instruction has no effect other than incrementing the program counter[1]. However the signature has changed. Analysis can detect simple obfuscations, like `nop`-insertion, by using regular expressions instead of fixed signatures. To catch `nop` insertions, the signature should allow for any number of `nop`s at instruction boundaries (Figure 2). In fact, most modern antivirus software use regular expressions for virus signatures.

E800	0000	00(90)*	5B(90)*	8D4B	42(90)*
51(90)*	50(90)*	50(90)*	0F01	4C24	FE(90)*
5B(90)*	83C3	1C(90)*	FA(90)*	8B2B	

Figure 2: Extended signature to catch `nop`-insertion.

Antivirus software deals with polymorphic viruses by performing heuristic analyses of the code (such as checking only certain program locations for virus code, as most polymorphic viruses attach themselves only at the beginning or end of the executable binary [36]), and even emulating the program in a sandbox to catch the virus in action [35]. The emulation technique is effective because at some point during the execution of the infected program, the virus body appears decrypted in main memory, ready for execution; the detection comes down to frequently scanning the in-memory image of the program for virus signatures while the program runs.

Metamorphic viruses attempt to evade heuristic detection techniques by using more complex obfuscations. When they

[1]Note that the subroutine address computation had to be updated to take into account the new `nop`s. This is a trivial computation and can be implemented by adding the number of inserted `nop`s to the initial offset hard-coded in the virus-morphing code.

replicate, these viruses change their code in a variety of ways, such as code transposition, substitution of equivalent instruction sequences, and register reassignment [41, 48]. Furthermore, they can “weave” the virus code into the host program, making detection by traditional heuristics almost impossible since the virus code is mixed with program code and the virus entry point is no longer at the beginning of the program (these are designated as entry point obscuring (EPO) viruses [25]).

As virus writers employ more complex obfuscation techniques, heuristic virus-detection techniques are bound to fail. Therefore, *there is need to perform a deeper analysis of malicious code based upon more sophisticated static-analysis techniques*. In other words, inspection of the code to detect malicious patterns should use structures that are closer to the semantics of the code, as purely syntactic techniques, such as regular expression matching, are no longer adequate.

3.1 The Suite of Viruses

We have analyzed multiple viruses using our tool, and discuss four of them in this paper. Descriptions of these viruses are given below.

3.1.1 Detailed Description of the Viruses

Chernobyl (CIH)
According to the Symantec Antivirus Reseach Center (SARC), *Chernobyl/CIH* is a virus that infects 32-bit Windows 95/98/NT executable files [39]. When a user executes an infected program under Windows 95/98/ME, the virus becomes resident in memory. Once the virus is resident, CIH infects other files when they are accessed. Infected files may have the same size as the original files because of CIH’s unique mode of infection: the virus searches for empty, unused spaces in the file[2]. Next it breaks itself up into smaller pieces and inserts its code into these unused spaces. Chernobyl has two different payloads: the first one overwrites the hard disk with random data, starting at the beginning of the disk (sector 0) using an infinite loop. The second payload tries to cause permanent damage to the computer by corrupting the Flash BIOS.

zombie-6.b
The *z0mbie-6.b* virus includes an interesting feature – the polymorphic engine hides every piece of the virus, and the virus code is added to the infected file as a chain of differently-sized routines, making standard signature detection techniques almost useless.

f0sf0r0
The *f0sf0r0* virus uses a polymorphic engine combined with an EPO technique to hide its entry point. According to Kaspersky Labs [26], when an infected file is run and the virus code gains control, it searches for Portable Executable files in the system directories and infects them. While infecting, the virus encrypts itself with a polymorphic loop and writes a result to the end of the file. To gain control when the infected file is run, the virus does not modify the program’s start address, but instead writes a “`jmp ⟨virus_entry⟩`” instruction into the middle of the file.

Hare
Finally, the *Hare* virus infects the bootloader sectors of floppy disks and hard drives, as well as executable programs. When the payload is triggered, the virus overwrites random sectors on the hard disk, making the data inaccessible. The virus spreads by polymorphically changing its decryption routine and encrypting its main body.

The Hare and Chernobyl/CIH viruses are well known in the antivirus community, with their presence in the wild peaking in 1996 and 1998, respectively. In spite of this, we discovered that *current commercial virus scanners could not detect slightly obfuscated versions of these viruses.*

4 Obfuscation Attacks on Commercial Virus Scanners

We tested three commercial virus scanners against several common obfuscation transformations. To test the resilience of commercial virus scanners to common obfuscation transformations, we have developed an obfuscator for binaries. Our obfuscator supports four common obfuscation transformations: dead-code insertion, code transposition, register reassignment, and instruction substitution. While there are other generic obfuscation techniques [11, 12], those described here seem to be preferred by malicious code writers, possibly because implementing them is easy and they add

[2]Most executable formats require that the various sections of the executable file start at certain aligned addresses, to respect the target platform’s idiosyncrasies. The extra space between the end of one section and the beginning of the next is usually padded with nulls.

little to the memory footprint.

4.1 Common Obfuscation Transformations

4.1.1 Dead-Code Insertion

Also known as *trash insertion*, dead-code insertion adds code to a program without modifying its behavior. Inserting a sequence of `nop` instructions is the simplest example. More interesting obfuscations involve constructing challenging code sequences that modify the program state, only to restore it immediately.

Some code sequences are designed to fool antivirus software that solely rely on signature matching as their detection mechanism. Other code sequences are complicated enough to make automatic analysis very time-consuming, if not impossible. For example, passing values through memory rather than through registers or the stack requires accurate pointer analysis to recover values. The example shown in Figure 3 should clarify this. The code marked by (*) can be easily eliminated by automated analysis. On the other hand, the second and third insertions, marked by (**), do cancel out but the analysis is more complex. Our obfuscator supports dead-code insertion.

Original code

```
call 0h
pop ebx
lea ecx, [ebx + 42h]
push ecx
push eax
push eax
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

Code obfuscated through dead-code insertion

```
call 0h
pop ebx
lea ecx, [ebx + 45h]
nop                        (*)
nop                        (*)
push ecx
push eax
inc eax                    (**)
push eax
dec [esp - 0h]             (**)
dec eax                    (**)
sidt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

Code obfuscated through code transposition

```
        call 0h
        pop ebx
        jmp Step2
Step3:  push eax
        push eax
        sidt [esp - 02h]
        jmp Step4
        add ebx, 1Ch
        jmp Step6
Step2:  lea ecx, [ebx + 45h]
        push ecx
        jmp Step3
Step4:  pop ebx
        cli
        jmp Step5
Step5:  mov ebp, [ebx]
```

Figure 3: Examples of obfuscation through dead-code insertion and code transposition. Newly added instructions are highlighted.

Not all dead-code sequence can be detected and eliminated, as this problem reduces to program equivalence (i.e., *Is this code sequence equivalent to an empty program?*), which is undecidable. We believe that a great many common dead-code sequences can be detected and eliminated with acceptable performance. To quote the documentation of the RPME virus permutation engine [47],

> [T]rash [does not make the] program more complex [...] . If [the] detecting algorithm will be written such as I think, then there is no difference between NOP and more complex trash.

Our detection tool, SAFE, identifies several kinds of such dead-code segments.

4.1.2 Code Transposition

Code transposition shuffles the instructions so that the order in the binary image is different from the execution order, or from the order of instructions assumed in the signature used by the antivirus software. To achieve the first variation, we randomly reorder the instructions and insert unconditional branches or *jumps* to restore the original control-flow. The second variation swaps instructions if they are not interdependent, similar to compiler code generation, but with the different goal of randomizing the instruction stream.

The two versions of this obfuscation technique differ in their complexity. The code transposition technique based upon unconditional branches is relatively easy to implement. The second technique that interchanges independent instructions is more complicated because the independence of instructions must be ascertained. On the analysis side, code transposition can complicate matters only for a human. Most automatic analysis tools (including ours) use an intermediate representation, such as the control flow graph (CFG) or the program dependence graph (PDG) [23], that is not sensitive to superfluous changes in control flow. Note that an optimizer acts as a deobfuscator in this case by finding the unnecessary unconditional branches and removing them from the program code. Currently, our obfuscator supports only code transposition based upon inserting unconditional branches.

4.1.3 Register Reassignment

The register reassignment transformation replaces usage of one register with another in a specific live range. This technique exchanges register names and has no other effect on program behavior. For example, if register `ebx` is dead throughout a given live range of the register `eax`, it can replace `eax` in that live range. In certain cases, register reassignment requires insertion of prologue and epilogue code around the live range to restore the state of various registers. Our binary obfuscator supports this code transformation.

The purpose of this transformation is to subvert the antivirus software analyses that rely upon signature-matching. There is no real obfuscatory value gained in this process. Conceptually, the deobfuscation challenge is equally complex before or after the register reassignment.

4.1.4 Instruction Substitution

This obfuscation technique uses a dictionary of equivalent instruction sequences to replace one instruction sequence with another. Since this transformation relies upon human knowledge of equivalent instructions, it poses the toughest challenge for automatic detection of malicious code. The IA-32 instruction set is especially rich, and provides several ways of performing the same operation. Coupled with several architecturally ambivalent features (e.g., a memory-based stack that can be accessed both as a stack using dedicated instructions and as a memory area using standard memory operations), the IA-32 assembly language provides ample opportunity for instruction substitution.

Original code	*Obfuscated code*
`call 0h`	`call 0h`
`pop ebx`	`pop ebx`
`lea ecx, [ebx + 42h]`	`lea ecx, [ebx + 42h]`
`push ecx`	`sub esp, 03h`
`push eax`	
`push eax`	
`sidt [esp - 02h]`	`sidt [esp - 02h]`
`pop ebx`	`add [esp], 1Ch`
	`mov ebx, [esp]`
`add ebx, 1Ch`	`inc esp`
`cli`	`cli`
`mov ebp, [ebx]`	`mov ebp, [ebx]`

Figure 4: Example of obfuscation through instruction substitution. Newly added instructions are highlighted.

To handle obfuscation based upon instruction substitution, an analysis tool must maintain a dictionary of equivalent instruction sequences, similar to the dictionary used to generate them. This is not a comprehensive solution, but it can cope with the common cases. In the case of IA-32, the problem can be slightly simplified by using a simple intermediate language that "unwinds" the complex operations corresponding to each IA-32 instruction. In some cases, a theorem prover such as Simplify [16] or PVS [37] can also be used to prove that two sequences of instructions are equivalent.

4.2 Testing Commercial Antivirus Tools

We tested three commercial virus scanners using obfuscated versions of the four viruses described earlier. The results were quite surprising: *a combination of `nop`-insertion and code transposition was enough to create obfuscated versions of the viruses that the commercial virus scanners could not detect*. Moreover, the Norton antivirus software could not detect an obfuscated version of the Chernobyl virus using just `nop`-insertions. SAFE was resistant to the two obfuscation transformations. The results are summarized in Table 1. A ✓ indicates that the antivirus software detected the virus. A ✗ means that the software did not detect the virus. Notice that unobfuscated versions of all four viruses were detected by all the tools.

		Norton® Antivirus 7.0	McAfee® VirusScan 6.01	Command® Antivirus 4.61.2	SAFE
Chernobyl	original	✓	✓	✓	✓
	obfuscated	✗[1]	✗[1,2]	✗[1,2]	✓
z0mbie-6.b	original	✓	✓	✓	✓
	obfuscated	✗[1,2]	✗[1,2]	✗[1,2]	✓
f0sf0r0	original	✓	✓	✓	✓
	obfuscated	✗[1,2]	✗[1,2]	✗[1,2]	✓
Hare	original	✓	✓	✓	✓
	obfuscated	✗[1,2]	✗[1,2]	✗[1,2]	✓

Obfuscations considered:	[1] = nop-insertion (a form of dead-code insertion)
	[2] = code transposition

Table 1: Results of testing various virus scanners on obfuscated viruses.

5 Architecture

This section gives an overview of the architecture of *SAFE* (Figure 5). Subsequent sections provide detailed descriptions of the major components of SAFE.

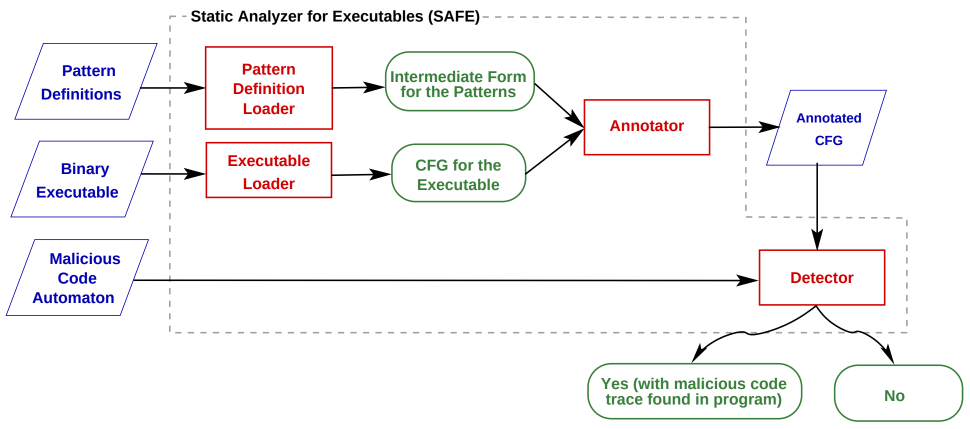

Static Analyzer for Executables (SAFE)
Pattern Definitions
Binary Executable
Malicious Code Automaton
Pattern Definition Loader
Executable Loader
Intermediate Form for the Patterns
CFG for the Executable
Annotator
Annotated CFG
Detector
Yes (with malicious code trace found in program)
No

Figure 5: Architecture of the static analyzer for executables (SAFE).

To detect malicious patterns in executables, we build an abstract representation of the malicious code (here a virus). The abstract representation is the "generalization" of the malicious code, e.g., it incorporates obfuscation transformations, such as superfluous changes in control flow and register reassignments. Similarly, one must construct an abstract representation of the executable in which we are trying to find a malicious pattern. Once the generalization of the malicious code and the abstract representation of the executable are created, we can then detect the malicious code in the executable. We now describe each component of SAFE.

Generalizing the malicious code: Building the malicious code automaton
The malicious code is generalized into an automaton with uninterpreted symbols. Uninterpreted symbols (Section 6.2) provide a generic way of representing data dependencies between variables without specifically referring to the storage location of each variable.

Pattern-definition loader
This component takes a library of *abstraction patterns* and creates an internal representation. These abstraction patterns are used as alphabet symbols by the malicious code automaton.

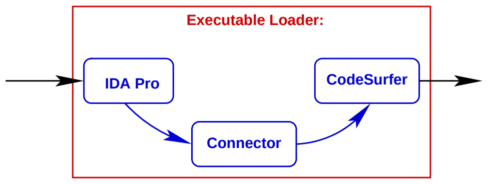


Figure 6: Implementation of executable loader module.

The executable loader
This component transforms the executable into an internal representation, here the collection of control flow graphs (CFGs), one for each program procedure. The executable loader (Figure 6) uses two off-the-shelf components, *IDA Pro* and *CodeSurfer*. IDA Pro (by DataRescue [15]) is a commercial interactive disassembler. *CodeSurfer* (by GrammaTech, Inc. [21]) is a program-understanding tool that performs a variety of static analyses. CodeSurfer provides an API for access to various structures, such as the CFGs and the call graph, and to results of a variety of static analyses, such as points-to analysis. In collaboration with GrammaTech, we have developed a connector that transforms IDA Pro internal structures into an intermediate form that CodeSurfer can parse.

The annotator
This component inputs a CFG from the executable and the set of abstraction patterns and produces an annotated CFG, the abstract representation of a program procedure. The annotated CFG includes information that indicates where a specific abstraction pattern was found in the executable. The annotator runs for each procedure in the program, transforming each CFG. Section 6 describes the annotator in detail.

The detector
This component computes whether the malicious code (represented by the malicious code automaton) appears in the abstract representation of the executable (created by the annotator). This component uses an algorithm based upon language containment and unification. Details can be found in Section 7.

Throughout the rest of the paper, the malicious code fragment shown in Figure 7 is used as a running example. This code fragment was extracted from the Chernobyl virus version 1.4.

To obtain the obfuscated code fragment depicted (Figure 8), we applied the following obfuscation transformations: dead-code insertion, code transposition, and register reassignment. Incidentally, the three commercial antivirus software (Norton, McAfee, and Command) detected the original code fragment shown. However, the obfuscated version was not detected by any of the three commercial antivirus software.

6 Program Annotator

This section describes the program annotator in detail and the data structures and static analysis concepts used in the detection algorithm. The program annotator inputs the CFG of the executable and a set of abstraction patterns and outputs an annotated CFG. The annotated CFG associates with each node n in the CFG a set of patterns that match the program at the point corresponding to the node n. The precise syntax for an abstraction pattern and the semantics of matching are provided later in the section.

Figure 9 shows the CFG and a simple annotated CFG corresponding to the obfuscated code from Figure 8. Note that one node in the annotated CFG can correspond to several nodes in the original CFG. For example, the nodes annotated with “IrrelevantInstr” corresponds to one or more `nop` instructions.

The annotations that appear in Figure 9 seem intuitive, but formulating them within a static-analysis framework requires formal definitions. We enhance the SAFE framework with a type system for x86 based on the typestate system described in [45]. However, other type systems designed for assembly languages, such as *Typed Assembly Language* [31, 32], could be used in the SAFE framework. Definitions, patterns, and the matching procedure are described in Sections 6.1, 6.2 and 6.3 respectively.

Original code

```
  WVCTF:
            mov     eax, dr1
            mov     ebx, [eax+10h]
            mov     edi, [eax]
LOWVCTF:
            pop     ecx
            jecxz   SFMM
            mov     esi, ecx
            mov     eax, 0d601h
            pop     edx
            pop     ecx
            call    edi
            jmp     LOWVCTF
   SFMM:
            pop     ebx
            pop     eax
            stc
            pushf
```

Figure 7: Original code fragment from Chernobyl virus version 1.4.

Obfuscated code

```
  WVCTF:
            mov     eax, dr1
            jmp     Loc1
   Loc2:
            mov     edi, [eax]
LOWVCTF:
            pop     ecx
            jecxz   SFMM
            nop
            mov     esi, ecx
            nop
            nop
            mov     eax, 0d601h
            jmp     Loc3
   Loc1:
            mov     ebx, [eax+10h]
            jmp     Loc2
   Loc3:
            pop     edx
            pop     ecx
            nop
            call    edi
            jmp     LOWVCTF
   SFMM:
            pop     ebx
            pop     eax
            push    eax
            pop     eax
            stc
            pushf
```

Figure 8: Obfuscated version based upon code in Figure 7.

6.1 Basic Definitions

This section provides the formal definitions used in the rest of the paper.

Program Points
An *instruction* I is a function application, $I : \tau_1 \times \cdots \times \tau_k \to \tau$. While the type system does not preclude higher-order functions or function composition, it is important to note that most assembly languages (including x86) do not support these concepts. A *program* P is a sequence of instructions $\langle I_1, \ldots, I_N \rangle$. During program execution, the instructions are processed in the sequential order they appear in the program, with the exception of control-flow instructions that can change the sequential execution order. The index of the instruction in the program sequence is called a *program point* (or *program counter*), denoted by the function $pc : \{I_1, \ldots, I_N\} \to [1, \ldots, N]$, and defined as $pc(I_j) \overset{def}{=} j,\ \forall\ 1 \le j \le N$. The set of all program points for program P is $ProgramPoints(P) \overset{def}{=} \{1, \ldots, N\}$. The pc function provides a total ordering over the set of program instructions.

Control Flow Graph
A *basic block* B is a sequence of instructions $\langle I_l, \ldots, I_m \rangle$ that contains at most one control-flow instruction, which must appear at the end. Thus, the execution within a basic block is by definition sequential. Let V be the set of basic blocks for a program P, and let $E \subseteq V \times V \times \{T, F\}$ be the set of control flow transitions between basic blocks. Each edge is marked with either T or F corresponding to the condition ($true$ or $false$) on which that edge is followed. Unconditional jumps have outgoing edges always marked with T. The directed graph $CFG(P) = \langle V, E \rangle$ is called the *control flow graph*.

Predicates
Predicates are the mechanism by which we incorporate results of various static analyses such as live range and points-to analysis. These predicates can be used in the definition of abstraction patterns. Table 2 lists predicates that are currently available in our system. For example, code between two program points p_1 and p_2 can be verified as dead-code (Section 4.1.1) by checking that for every variable m that is live in the program range $[p_1, p_2]$, its value at point p_2 is the same as its value at point p_1. The change in m's value between two program points p_1 and p_2 is denoted by $Delta(m, p_1, p_2)$ and can be implemented using polyhedral analysis [14].

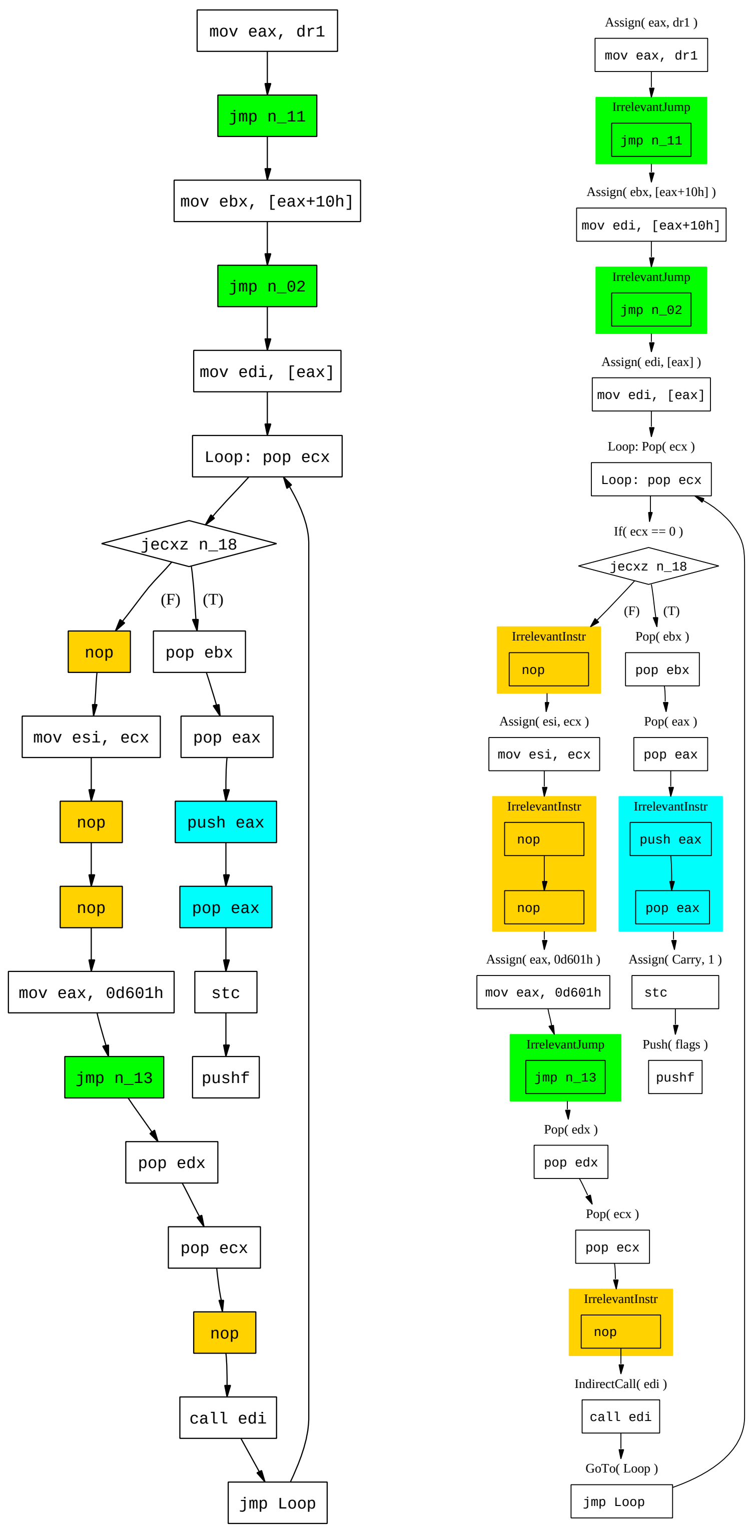


Figure 9: Control flow graph of obfuscated code fragment, and annotations.

$Dominators(B)$	the set of basic blocks that dominate the basic block B
$PostDominators(B)$	the set of basic blocks that are dominated by the basic block B
$Pred(B)$	the set of basic blocks that immediately precede B
$Succ(B)$	the set of basic blocks that immediately follow B
$First(B)$	the first instruction of the basic block B
$Last(B)$	the last instruction of the basic block B
$Previous(I)$	$\begin{cases} \bigcup_{B' \in Pred(B_I)} Last(B') & \text{if } I = First(B_I) \\ I' & \text{if } B_I = \langle \ldots, I', I, \ldots \rangle \end{cases}$
$Next(I)$	$\begin{cases} \bigcup_{B' \in Succ(B_I)} First(B') & \text{if } I = Last(B_I) \\ I' & \text{if } B_I = \langle \ldots, I, I', \ldots \rangle \end{cases}$
$Kills(p, a)$	*true* if the instruction at program point p kills variable a
$Uses(p, a)$	*true* if the instruction at program point p uses variable a
$Alias(p, x, y)$	*true* if variable x is an alias for y at program point p
$LiveRangeStart(p, a)$	the set of program points that start the a's live range that includes p
$LiveRangeEnd(p, a)$	the set of program points that end the a's live range that includes p
$Delta(p, m, n)$	the difference between integer variables m and n at program point p
$Delta(m, p_1, p_2)$	the change in m's value between program points p_1 and p_2
$PointsTo(p, x, a)$	*true* if variable x points to location of a at program point p

Table 2: Examples of static analysis predicates.

Explanation of the static analysis predicates shown in Table 2 are standard and can be found in a compiler textbook (such as [33]).

Instructions and Data Types
The type constructors build upon simple integer types (listed below as the *ground* class of types), and allow for array types (with two variations: the pointer-to-start-of-array type and the pointer-to-middle-of-array type), structures and unions, pointers, and functions. Two special types $\bot(\mathsf{n})$ and $\top(\mathsf{n})$ complete the type system lattice. $\bot(\mathsf{n})$ and $\top(\mathsf{n})$ represent types that are stored on n bits, with $\bot(\mathsf{n})$ being the least specific ("any") type and $\top(\mathsf{n})$ being the most specific type. Table 3 describes the constructors allowed in our type system.

τ	::	ground	*Ground types*
	|	τ`[n]`	*Pointer to the base of an array of type τ and of size* n
	|	τ`(n)`	*Pointer into the middle of an array of type τ and of size* n
	|	τ `ptr`	*Pointer to τ*
	|	$\mathrm{s}\{\mu_1, \ldots, \mu_\mathrm{k}\}$	*Structure (product of types of μ_i)*
	|	$\mathrm{u}\{\mu_1, \ldots, \mu_\mathrm{k}\}$	*Union*
	|	$\tau_1 \times \cdots \times \tau_\mathrm{k} \rightarrow \tau$	*Function*
	|	$\top$`(n)`	*Top type of* n *bits*
	|	$\bot$`(n)`	*Bottom type of* n *bits (type "any" of* n *bits)*
μ	::	(l, τ, i)	*Member labeled l of type τ at offset i*
ground	::	`int(`g`:`s`:`v`)` | `uint(`g`:`s`:`v`)` | ...	

Table 3: A simple type system.

The type $\mu(l, \tau, i)$ represents the type of a field member of a structure. The field has a type τ (independent of the types of all other fields in the same structure), an offset i that uniquely determines the location of the field within the structure, and a label l that identifies the field within the structure (in some cases this label might be undefined).

Physical subtyping takes into account the layout of values in memory [6, 45]. If a type τ is a *physical subtype* of τ'

Code	*Type*
`call 0h`	
`pop ebx`	`ebx` : $\perp(32)$
`lea ecx, [ebx + 42h]`	`ecx` : $\perp(32)$, `ebx` : `ptr` $\perp(32)$
`push ecx`	`ecx` : $\perp(32)$
`push eax`	`eax` : $\perp(32)$
`push eax`	`eax` : $\perp(32)$
`sidt [esp - 02h]`	
`pop ebx`	`eax` : $\perp(32)$
`add ebx, 1Ch`	`ebx` : `int`$(0{:}1{:}31)$
`cli`	
`mov ebp, [ebx]`	`ebp` : $\perp(32)$, `ebx` : `ptr` $\perp(32)$

Figure 10: Inferred types from Chernobyl/CIH virus code.

(denoted it by $\tau \leq \tau'$), then the memory layout of a value of type τ' is a prefix of the memory layout of a value of type τ. We will not describe the rules of physical subtyping here as we refer the reader to Xu's thesis [45] for a detailed account of the typestate system (including subtyping rules).

The type `int`$(g$:s:$v)$ represents a signed integer, and it covers a wide variety of values within storage locations. It is parametrized using three parameters as follows: g represents the number of highest bits that are ignored, s is the number of middle bits that represent the sign, and v is the number of lowest bits that represent the value. Thus the type `int`$(g$:s:$v)$ uses a total of $g + s + v$ bits.

$$\underbrace{d_{g+s+v} \ldots d_{s+v+1}}_{\text{ignored}} \underbrace{d_{s+v} \ldots d_{v+1}}_{\text{sign}} \underbrace{d_v \ldots d_1}_{\text{value}}$$

The type `uint`$(g$:s:$v)$ represents an unsigned integer, and it is just a variation of `int`$(g$:s:$v)$, with the middle s sign bits always set to zero.

The notation `int`$(g$:s:$v)$ allows for the separation of the data and storage location type. In most assembly languages, it is possible to use a storage location larger than that required by the data type stored in it. For example, if a byte is stored right-aligned in a (32-bit) word, its associated type is `int`$(24$:1:$7)$. This means that an instruction such as *xor on least significant byte within 32-bit word* will preserve the leftmost 24 bits of the 32-bit word, even though the instruction addresses the memory on 32-bit word boundary.

This separation between data and storage location raises the issue of alignment information, i.e., most computer systems require or prefer data to be at a memory address aligned to the data size. For example, 32-bit integers should be aligned on 4-byte boundaries, with the drawback that accessing an unaligned 32-bit integer leads to either a slowdown (due to several aligned memory accesses) or an exception that requires handling in software. Presently, we do not use alignment information as it does not seem to provide a significant covert way of changing the program flow.

Figure 10 shows the types for operands in a section of code from the Chernobyl/CIH virus. Table 4 illustrates the type system for Intel IA-32 architecture. There are other IA-32 data types that are not covered in Table 4, including bit strings, byte strings, 64- and 128-bit packed SIMD types, and BCD and packed BCD formats. The IA-32 logical address is a combination of a 16-bit segment selector and a 32-bit segment offset, thus its type is the cross product of a 16-bit unsigned integer and a 32-bit pointer.

6.2 Abstraction Patterns

An abstraction pattern Γ 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. Formally, a pattern $\Gamma = (V, O, C)$ is a 3-tuple defined as follows:

$$\begin{array}{lcl} V & = & \{\, x_1 : \tau_1, \ldots, x_k : \tau_k \,\} \\ O & = & \langle\, I(v_1, \ldots, v_m) \mid I : \tau_1 \times \cdots \times \tau_m \to \tau \,\rangle \\ C & = & \text{boolean expression involving static} \\ & & \text{analysis predicates and logical operators} \end{array}$$

IA-32 Datatype	Type Expression
Unsigned Integer Types	
`byte unsigned int`	`uint`(0:0:8)
`word unsigned int`	`uint`(0:0:16)
`doubleword unsigned int`	`uint`(0:0:32)
`quadword unsigned int`	`uint`(0:0:64)
`double quadword unsigned int`	`uint`(0:0:128)
Signed Integer Types	
`byte signed int`	`int`(0:1:7)
`word signed int`	`int`(0:1:15)
`doubleword signed int`	`int`(0:1:31)
`quadword signed int`	`int`(0:1:63)
`double quadword signed int`	`int`(0:1:127)
Floating-Point Types	
`single precision float`	`float`(0:1:31)
`double precision float`	`float`(0:1:63)
`double extended precision float`	`float`(0:1:79)
Pointers to Memory Locations	
`near pointer`	$\perp(32)$
`far pointer (logical address)`	`uint`(0:0:16) $\times$ `uint`(0:0:32) $\to \perp(48)$
Registers	
`eax, ebx, ecx, edx`	$\perp(32)$
`esi, edi, ebp, esp`	$\perp(32)$
`eip`	`int`(0:1:31)
`cs, ds, ss, es, fs, gs`	$\perp(16)$
`ax, bx, cx, dx`	$\perp(16)$
`al, bl, cl, dl`	$\perp(8)$
`ah, bh, ch, dh`	$\perp(8)$

Table 4: IA-32 datatypes and their corresponding expression in the type system from Table 3.

An instruction from the sequence O has a number of arguments $(v_i)_{i\geq 0}$, where each argument is either a literal value or a free variable x_j. We write $\Gamma(x_1 : \tau_1, \ldots, x_k : \tau_k)$ to denote the pattern $\Gamma = (V, O, C)$ with free variables $x_1, \ldots, x_k$. An example of a pattern is shown below.

$$\begin{array}{l} \Gamma(\ X : int(0:1:31)\ ) = \\ \quad (\quad \{\ X : int(0:1:31)\ \}, \\ \qquad \langle\ p_1 :\ \text{“}pop\ X\text{”}, \\ \qquad\quad p_2 :\ \text{“}add\ X, \texttt{03AFh}\text{”}\ \rangle, \\ \qquad p_1 \in LiveRangeStart(p_2, X)\ \ ) \end{array}$$

This pattern represents two instructions that pop a register X off the stack and then add a constant value to it (`0x03AF`). Notice the use of uninterpreted symbol X in the pattern. Use of the uninterpreted symbols in a pattern allows it to match multiple sequences of instructions, e.g., the patterns shown above matches any instantiation of the pattern where X is assigned a specific register. The type $int(0:1:31)$ of X represents an integer with 31 bits of storage and one sign bit.

We define a *binding* $\mathcal{B}$ as a set of pairs [variable v, value x]. Formally, a binding $\mathcal{B}$ is defined as $\{\ [x, v] \mid x \in V,\ x : \tau,\ v : \tau',\ \tau \leq \tau'\ \}$. If a pair $[x, v]$ occurs in a binding $\mathcal{B}$, then we write $\mathcal{B}(x) = v$. Two bindings $\mathcal{B}_1$ and $\mathcal{B}_2$ are said to be *compatible* if they do not bind the same variable to different values:

$$\begin{array}{l} \mathit{Compatible}(\mathcal{B}_1, \mathcal{B}_2) \overset{def}{=} \\ \quad \forall\, x \in V.(\ [x, y_1] \in \mathcal{B}_1\ \wedge\ [x, y_2] \in \mathcal{B}_2\ ) \\ \qquad \Rightarrow (y_1 = y_2) \end{array}$$

The *union of two compatible bindings* $\mathcal{B}_1$ and $\mathcal{B}_2$ includes all the pairs from both bindings. For incompatible bindings, the union operation returns an empty binding.

$$\mathcal{B}_1 \cup \mathcal{B}_2 \overset{def}{=} \begin{cases} \{\ [x, v_x] : [x, v_x] \in \mathcal{B}_1 \vee [x, v_x] \in \mathcal{B}_2\ \} \\ \qquad\qquad \text{if } \mathit{Compatible}(\mathcal{B}_1, \mathcal{B}_2) \\ \\ \emptyset \qquad\qquad \text{if } \neg\ \mathit{Compatible}(\mathcal{B}_1, \mathcal{B}_2) \end{cases}$$

When matching an abstraction pattern against a sequence of instructions, we use unification to bind the free variables of Γ to actual values. The function

$$\mathit{Unify}\ (\ \langle \ldots, op_i(x_{i,1}, \ldots, x_{i,n_i}), \ldots \rangle_{1 \le i \le m},\ \Gamma)$$

returns a "most general" binding $\mathcal{B}$ if the instruction sequence $\langle op_1(x_{1,1}, \ldots, x_{1,n_1}), \ldots, op_m(x_{m,1}, \ldots, x_{m,n_m}) \rangle$ can be unified with the sequence of instructions O specified in the pattern Γ. If the two instruction sequences cannot be unified, Unify returns *false*. Definitions and algorithms related to unification are standard and can be found in [19].[3]

6.3 Annotator Operation

The annotator associates a set of matching patterns with each node in the CFG. The annotated CFG of a program procedure P with respect to a set of patterns Σ is denoted by P_Σ. Assume that a node n in the CFG corresponds to the program point p and the instruction at p is I_p. The annotator attempts to match the (possibly interprocedural) instruction sequence $S(n) = \langle \ldots, Previous^2(I_p), Previous(I_p), I_p \rangle$ with the patterns in the set $\Sigma = \{\Gamma_1, \ldots, \Gamma_m\}$. The CFG node n is then labeled with the list of pairs of patterns and bindings that satisfy the following condition:

$$Annotation(n) = \{\ [\Gamma, \mathcal{B}]\ : \Gamma \in \{\Gamma_1, \ldots, \Gamma_m\} \wedge \\ \mathcal{B} = \mathit{Unify}(S(n), \Gamma)\ \}$$

If $\mathit{Unify}(S(n), \Gamma)$ returns $false$ (because unification is not possible), then the node n is not annotated with $[\Gamma, \mathcal{B}]$. Note that a pattern Γ might appear several times (albeit with different bindings) in $Annotation(n)$. However, the pair $[\Gamma, \mathcal{B}]$ is unique in the annotation set of a given node.

7 Detector

The detector takes as its inputs an annotated CFG for an executable program procedure and a malicious code automaton. If the malicious pattern described by the malicious code automaton is also found in the annotated CFG, the detector returns the sequence of instructions exhibiting the pattern. The detector returns *no* if the malicious pattern cannot be found in the annotated CFG.

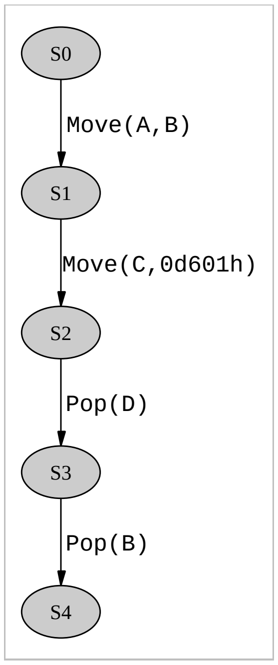
7.1 The Malicious-Code Automaton

Intuitively, the malicious code automaton is a generalization of the vanilla virus, i.e., the malicious code automaton also represents obfuscated strains of the virus. Formally, a *malicious code automaton* (or *MCA*) $\mathcal{A}$ is a 6-tuple $(V, \Sigma, S, \delta, S_0, F)$, where
• $V = \{v_1 : \tau_1, \ldots, v_k : \tau_k\}$ is a *set of typed variables*,
• $\Sigma = \{\Gamma_1, \ldots, \Gamma_n\}$ is a *finite alphabet* of patterns parametrized by variables from V, for $1 \le i \le n$, $P_i = (V_i,\ O_i,\ C_i)$ where $V_i \subseteq V$,
• S is a finite set of *states*,
• $\delta : S \times \Sigma \to 2^S$ is a *transition function*,
• $S_0 \subseteq S$ is a non-empty set of *initial states*,
• $F \subseteq S$ is a non-empty set of *final states*.

An MCA is a generalization of an ordinary finite-state automaton in which the alphabets are a finite set of patterns defined over a set of typed variables. Given a binding $\mathcal{B}$ for the variables $V = \{v_1, \ldots, v_k\}$, the finite-state automaton obtained by substituting $\mathcal{B}(v_i)$ for v_i for all $1 \le i \le k$ in $\mathcal{A}$ is denoted by $\mathcal{B}(\mathcal{A})$. Notice that $\mathcal{B}(\mathcal{A})$ is a simple finite-state automaton. We explain this using an example. Consider the MCA $\mathcal{A}$ shown in Figure 11 with $V = \{A, B, C, D\}$.

[3]We use one-way matching which is simpler than full unification. Note that the instruction sequence does not contain any variables. We instantiate variables in the pattern so that they match the corresponding terms in the instruction sequence.

The automata obtained from $\mathcal{A}$ corresponding to the bindings $\mathcal{B}_1$ and $\mathcal{B}_2$ are shown in Figure 11. The uninterpreted variables in the MCA were introduced to handle obfuscation transformations based on register reassignment. The malicious code automaton corresponding to the code fragment shown in Figure 7 (from the Chernobyl virus) is depicted in Figure 12.



```
mov     esi, ecx
mov     eax, 0d601h
pop     edx
pop     ecx
```

$\mathcal{B}_1 = \{$ [A, esi],
[B, ecx],
[C, eax],
[D, edx] $\}$

```
mov     esi, eax
mov     ebx, 0d601h
pop     ecx
pop     eax
```

$\mathcal{B}_2 = \{$ [A, esi],
[B, eax],
[C, ebx],
[D, ecx] $\}$

Figure 11: Malicious code automaton for a Chernobyl virus code fragment, and instantiations with different register assignments, shown with their respective bindings.

7.2 Detector Operation

The detector takes as its inputs the annotated CFG P_Σ of a program procedure P and a malicious code automaton MCA $\mathcal{A} = (V, \Sigma, S, \delta, S_0, F)$. Note that the set of patterns Σ is used both to construct the annotated CFG and as the alphabet of the malicious code automaton. Intuitively, the detector determines whether there exists a malicious pattern that occurs in $\mathcal{A}$ and P_Σ. We formalize this intuitive notion. The annotated CFG P_Σ is 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. Our detector determines whether the following language is empty:

$$L(P_\Sigma) \cap \left(\bigcup_{\mathcal{B} \in \mathcal{B}_{All}} L(\mathcal{B}(\mathcal{A})) \right)$$

In the expression given above, $L(P_\Sigma)$ is the language corresponding to the annotated CFG and $\mathcal{B}_{All}$ is the set of all bindings to the variables in the set V. In other words, the detector determines whether there exists a binding $\mathcal{B}$ such that the intersection of the languages P_Σ and $\mathcal{B}(\mathcal{A})$ is non-empty.

Our detection algorithm is very similar to the classic algorithm for determining whether the intersection of two regular languages is non-empty [22]. However, due to the presence of variables, we must perform unification during the algorithm. Our algorithm (Figure 13) combines the classic algorithm for computing the intersection of two regular languages with unification. We have implemented the algorithm as a data-flow analysis.

• For each node n of the annotated CFG P_A we associate pre and post lists L_n^{pre} and L_n^{post} respectively. Each element of a list is a pair $[s, \mathcal{B}]$, where s is the state of the MCA $\mathcal{A}$ and $\mathcal{B}$ is the binding of variables. Intuitively, if $[s, \mathcal{B}] \in L_n^{pre}$, then it is possible for $\mathcal{A}$ with the binding $\mathcal{B}$ (i.e. for $\mathcal{B}(\mathcal{A})$) to be in state s just before node n.

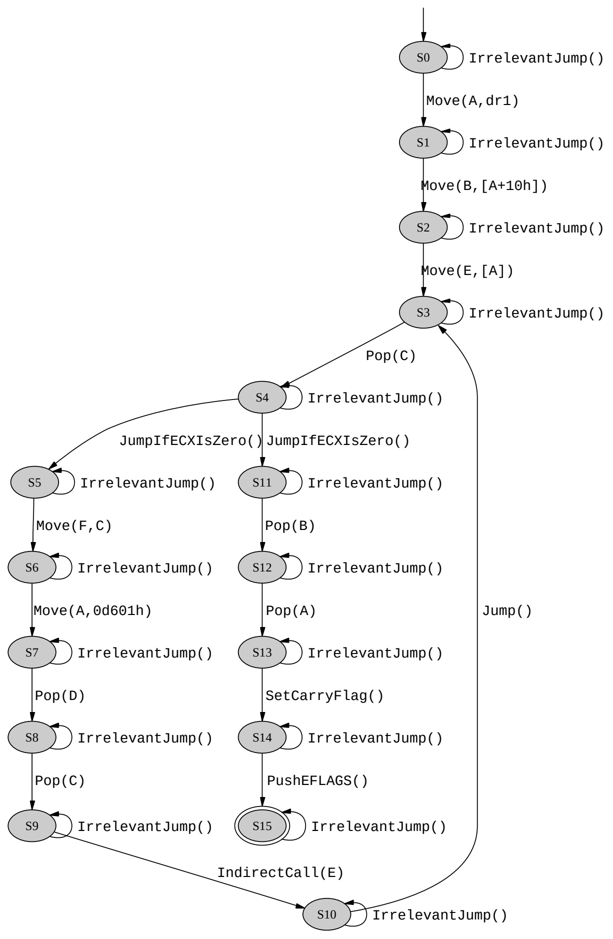


Figure 12: Malicious code automaton corresponding to code fragment from Figure 7.

• **Initial condition:** Initially, both lists associated with all nodes except the start node n_0 are empty. The pre list associated with the start node is the list of all pairs $[s, \emptyset]$, where s is an initial state of the MCA $\mathcal{A}$, and the post list associated with the start node is empty.

• **The do-until loop:** The **do-until** loop updates the pre and post lists of all the nodes. At the end of the loop, the worklist WS contains the set of nodes whose pre or post information has changed. The loop executes until the pre and post information associated with the nodes does not change, and a fixed point is reached. The join operation that computes L_i^{pre} takes the list of state-binding pairs from all of the L_j^{post} sets for program points preceding i and copies them to L_i^{pre} only if there are no repeated states. In case of repeated states, the conflicting pairs are merged into a single pair only if the bindings are compatible. If the bindings are incompatible, both pairs are thrown out.

• **Diagnostic feedback:** Suppose our algorithm returns a non-empty set, meaning a malicious pattern is common to the annotated CFG P_Σ and MCA $\mathcal{A}$. In this case, we return the sequence of instructions in the executable corresponding to the malicious pattern. This is achieved by keeping an additional structure with the algorithm. Every time the post list for a node n is updated by taking a transition in $\mathcal{A}$ (see the statement 14 in Figure 13), we store the predecessor of the added state, i.e., if $[\delta(s, \Gamma), \mathcal{B}_s \cup \mathcal{B}]$ is added to L_n^{post}, then we add an edge from s to $\delta(s, \Gamma)$ (along with the binding $\mathcal{B}_s \cup \mathcal{B}$) in the associated structure. Suppose we detect that L_n^{post} contains a state $[s, \mathcal{B}_s]$, where s is a final state of the MCA $\mathcal{A}$. Then we traceback the associated structure from s until we reach an initial state of $\mathcal{A}$ (storing the instructions occurring along the way).

```
Input: A list of patterns $\Sigma = \{P_1, \ldots, P_r\}$, a malicious code automaton $\mathcal{A} = (V, \Sigma, S, \delta, S_0, F)$,
and an annotated CFG $P_\Sigma =< N, E >$
Output: $true$ if the program is likely infected, $false$ otherwise
MaliciousCodeChecking($\Sigma, \mathcal{A}, P_\Sigma$)
(1)     $L^{pre}_{n_0} \leftarrow \{ [s, \emptyset] \mid s \in S_0 \}$, where $n_0 \in N$ is the entry node of $P_\Sigma$
(2)     foreach $n \in N$ do $L^{pre}_n \leftarrow \emptyset$
(3)     foreach $n \in N$ do $L^{post}_n \leftarrow \emptyset$
(4)     $WS \leftarrow \emptyset$
(5)     do
(6)         $WS_{old} \leftarrow WS$
(7)         $WS \leftarrow \emptyset$
(8)         foreach $n \in N$                                   // update pre information
(9)             if $L^{pre}_n \neq \bigcup_{m \in Previous(n)} L^{post}_m$
(10)                $L^{pre}_n \leftarrow \bigcup_{m \in Previous(n)} L^{post}_m$
(11)                $WS \leftarrow WS \cup \{n\}$
(12)        foreach $n \in N$                                   // update post information
(13)            $NewL^{post}_n \leftarrow \emptyset$
(14)            foreach $[s, \mathcal{B}_s] \in L^{pre}_n$
(15)                foreach $[\Gamma, \mathcal{B}] \in Annotation(n)$             // follow a transition
(16)                        $\wedge\ Compatible(\mathcal{B}_s, \mathcal{B})$
(17)                    add $[\ \delta(s, \Gamma),\ \mathcal{B}_s \cup \mathcal{B}\ ]$ to $NewL^{post}_n$
(18)            if $L^{post}_n \neq NewL^{post}_n$
(19)                $L^{post}_n \leftarrow NewL^{post}_n$
(20)                $WS \leftarrow WS \cup \{n\}$
(21)    until $WS = \emptyset$
(22)    return $\exists\ n \in N\ .\ \exists\ [s, \mathcal{B}_s] \in L^{post}_n\ .\ s \in F$
```

Figure 13: Algorithm to check a program model against a malicious code specification.

8 Experimental Data

The three major goals of our experiments were to measure the execution time of our tool and find the false positive and negative rates. Our testing proceeded as follows:

• First, we constructed ten obfuscated versions of the four viruses. Let $V_{i,k}$ (for $1 \le i \le 4$ and $1 \le k \le 10$) denote the k-th version of the i-th virus. The obfuscated versions were created by varying the obfuscation parameters, e.g., number of nops and inserted jumps. For the i-th virus, $V_{i,1}$ denoted the "vanilla" or the unobfuscated version of the virus.

• Let M_1, M_2, M_3 and M_4 be the malicious code automata corresponding to the four viruses.

Testing environment: The testing environment consisted of a Microsoft Windows 2000 machine. The hardware configuration included an AMD Athlon 1 GHz processor and 1 GB of RAM. We used CodeSurfer version 1.5 patchlevel 0 and IDA Pro version 4.1.7.600.

Testing on malicious code: We will describe the testing with respect to the first virus. The testing for the other viruses is analogous. First, we ran SAFE on the 10 versions of the first virus $V_{1,1}, \ldots, V_{1,10}$ with malicious code automaton M_1. This experiment gave us the false negative rate, i.e., the pattern corresponding to M_1 should be detected in all versions of the virus.

Next, we executed SAFE on the versions of the viruses $V_{i,k}$ with the malicious code automaton M_j (where $i \neq j$). This helped us find the false positive rate of SAFE.

We found that SAFE's false positive and negative rate were 0. We also measured the execution times for each run. Since IDA Pro and CodeSurfer were not implemented by us, we did not measure the execution times for these components. We report the average and standard deviation of the execution times in Tables 5 and 6.

Testing on benign code: We considered a suite of benign programs (see Section 8.1 for descriptions). For each benign program, we executed SAFE on the malicious code automaton corresponding to the four viruses. Our detector reported "negative" in each case, i.e., the false positive rate is 0. The average and variance of the execution times are reported in Table 7. As can be seen from the results, for certain cases the execution times are unacceptably large. We will address performance enhancements to SAFE in the future.

	Annotator		Detector	
	avg.	(std. dev.)	avg.	(std. dev.)
Chernobyl	1.444 s	(0.497 s)	0.535 s	(0.043 s)
z0mbie-6.b	4.600 s	(2.059 s)	1.149 s	(0.041 s)
f0sf0r0	4.900 s	(2.844 s)	0.923 s	(0.192 s)
Hare	9.142 s	(1.551 s)	1.604 s	(0.104 s)

Table 5: SAFE performance when checking obfuscated viruses for false negatives.

	Annotator		Detector	
	avg.	(std. dev.)	avg.	(std. dev.)
z0mbie-6.b	3.400 s	(1.428 s)	1.400 s	(0.420 s)
f0sf0r0	4.900 s	(1.136 s)	0.840 s	(0.082 s)
Hare	1.000 s	(0.000 s)	0.220 s	(0.019 s)

Table 6: SAFE performance when checking obfuscated viruses for false positives against the Chernobyl/CIH virus.

8.1 Descriptions of the Benign Executables

tiffdither.exe is a command line utility in the *cygwin* toolkit version 1.3.70, a UNIX environment developed by Red Hat, for Windows.
winmine.exe is the Microsoft Windows 2000 Minesweeper game, version 5.0.2135.1.
spyxx.exe is a Microsoft Visual Studio 6.0 Spy++ utility, that allows the querying of properties and monitoring of messages of Windows applications. The executable we tested was marked as version 6.0.8168.0.
QuickTimePlayer.exe is part of the Apple QuickTime media player, version 5.0.2.15.

	Executable	`.text`	Procedure	Annotator		Detector	
	size	size	count	avg.	(std. dev.)	avg.	(std. dev.)
tiffdither.exe	9,216 B	6,656 B	29	6.333 s	(0.471 s)	1.030 s	(0.043 s)
winmine.exe	96,528 B	12,120 B	85	15.667 s	(1.700 s)	2.283 s	(0.131 s)
spyxx.exe	499,768 B	307,200 B	1,765	193.667 s	(11.557 s)	30.917 s	(6.625 s)
QuickTimePlayer.exe	1,043,968 B	499,712 B	4,767	799.333 s	(5.437 s)	160.580 s	(4.455 s)

Table 7: SAFE performance in seconds when checking clean programs against the Chernobyl/CIH virus.

9 Conclusion and Future Work

We presented a unique view of malicious code detection as a obfuscation-deobfuscation game. We used this viewpoint to explore obfuscation attacks on commercial virus scanners, and found that three popular virus scanners were susceptible to these attacks. We presented a static analysis framework for detecting malicious code patterns in executables. Based upon our framework, we have implemented SAFE, a static analyzer for executables that detects malicious patterns in executables and is resilient to common obfuscation transformations.

For future work, we will investigate the use of theorem provers during the construction of the annotated CFG. For instance, SLAM [2] uses the theorem prover Simplify [16] for predicate abstraction of C programs. Our detection algorithm is context insensitive and does not track the calling context of the executable. We will investigate the use of the PDS formalism, which would make our algorithm context sensitive. However, the existing PDS formalism does not allow uninterpreted variables, so it will have to be extended to be used in our context.

Acknowledgments:
We would like to thank Thomas Reps and Jonathon Giffin for providing us with invaluable comments on earlier drafts of the paper.

References

[1] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *2002 IEEE Symposium on Security and Privacy (Oakland'02)*, pages 143–159, May 2002.

[2] T. Ball and S.K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN'01)*, volume 2057 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Advances in Cryptology (CRYPTO'01)*, volume 2139 of *Lecture Notes in Computer Science*, pages 1 – 18. Springer-Verlag, August 2001.

[4] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2), 1996.

[5] CERT Coordination Center. Denial of service attacks, June 2001. `http://www.cert.org/tech_tips/denial_of_service.html` (Last accessed: 22 Aug. 2003).

[6] S. Chandra and T.W. Reps. Physical type checking for C. In *ACM SIGPLAN - SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 66 – 75. ACM Press, September 1999.

[7] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *9th ACM Conference on Computer and Communications Security (CCS'02)*. ACM Press, November 2002.

[8] B.V. Chess. Improving computer security using extending static checking. In *2002 IEEE Symposium on Security and Privacy (Oakland'02)*, pages 160–173, May 2002.

[9] D.M. Chess and S.R. White. An undetectable computer virus. In *Proceedings of Virus Bulletin Conference*, 2000.

[10] F. Cohen. Computer viruses: Theory and experiments. *Computers and Security*, 6:22 – 35, 1987.

[11] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July 1997.

[12] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, San Diego, California, USA, January 1998. ACM Press.

[13] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 439–448. ACM Press, 2000.

[14] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'78)*, pages 84 – 96. ACM Press, January 1978.

[15] DataRescue sa/nv. IDA Pro – interactive disassembler. `http://www.datarescue.com/idabase/` (Last accessed: 3 Feb. 2003).

[16] D. Detlefs, G. Nelson, and J. Saxe. The simplify theorem prover. `http://research.compaq.com/SRC/esc/simplify.html`.

[17] U. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *2000 IEEE Symposium on Security and Privacy (Oakland'00)*, pages 246–255, May 2000.

[18] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer-Verlag, July 2000.

[19] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, 1996.

[20] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium (Security'02)*. USENIX Association, August 2002.

[21] GrammaTech Inc. CodeSurfer – code analysis and understanding tool. `http://www.grammatech.com/products/codesurfer/index.html` (Last accessed: 3 Feb. 2003).

[22] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2001.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, January 1990.

[24] T. Jensen, D.L. Metayer, and T. Thorn. Verification of control flow based security properties. In *1999 IEEE Symposium on Security and Privacy (Oakland'99)*, May 1999.

[25] E. Kaspersky. *Virus List Encyclopaedia*, chapter Ways of Infection: Viruses without an Entry Point. Kaspersky Labs, 2002. `http://www.viruslist.com/eng/viruslistbooks.asp?id=32&key=0000100007000020000100003` (Last accessed: 3 Feb. 2003).

[26] Kaspersky Labs. `http://www.kasperskylabs.com` (Last accessed: 3 Feb. 2003).

[27] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):323 – 337, December 1992.

[28] R.W. Lo, K.N. Levitt, and R.A. Olsson. MCF: A malicious code filter. *Computers & Society*, 14(6):541–566, 1995.

[29] G. McGraw and G. Morrisett. Attacking malicious code: report to the Infosec research council. *IEEE Software*, 17(5):33 – 41, September/October 2000.

[30] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the Sapphire/Slammer worm. Technical report, The Cooperative Association for Internet Data Analysis (CAIDA), February 2003. `http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html` (Last accessed: 3 Feb. 2003).

[31] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based Typed Assembly Language. In Xavier Leroy and Atsushi Ohori, editors, *1998 Workshop on Types in Compilation*, volume 1473 of *Lecture Notes in Computer Science*, pages 28 – 52. Springer-Verlag, March 1998.

[32] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, pages 85 – 97. ACM Press, January 1998.

[33] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[34] E.M. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the 8th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'81)*, pages 219 – 230. ACM Press, January 1981.

[35] C. Nachenberg. Polymorphic virus detection module. *United States Patent # 5,696,822*, December 9, 1997.

[36] C. Nachenberg. Polymorphic virus detection module. *United States Patent # 5,826,013*, October 20, 1998.

[37] S. Owre, S. Rajan, J. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, August 1996.

[38] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 49–61. ACM Press, January 1995.

[39] M. Samamura. *Expanded Threat List and Virus Encyclopaedia*, chapter W95.CIH. Symantec Antivirus Research Center, 1998. `http://securityresponse.symantec.com/avcenter/venc/data/cih.html` (Last accessed: 3 Feb. 2003).

[40] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium (Security'02)*, pages 149 – 167. USENIX Association, August 2002.

[41] P. Ször and P. Ferrie. Hunting for metamorphic. In *Proceedings of Virus Bulletin Conference*, pages 123 – 144, September 2001.

[42] TESO. Burneye ELF encryption program. `https://teso.scene.at` (Last accessed: 3 Feb. 2003).

[43] D. Wagner and D. Dean. Intrusion detection via static analysis. In *2001 IEEE Symposium on Security and Privacy (Oakland'01)*, May 2001.

[44] R. Wang. Flash in the pan? *Virus Bulletin*, July 1998. Virus Analysis Library.

[45] Z. Xu. *Safety-Checking of Machine Code*. PhD thesis, University of Wisconsin, Madison, 2000.

[46] z0mbie. Automated reverse engineering: Mistfall engine. `http://z0mbie.host.sk/autorev.txt` (Last accessed: 3 Feb. 2003).

[47] z0mbie. RPME mutation engine. `http://z0mbie.host.sk/rpme.zip` (Last accessed: 3 Feb. 2003).

[48] z0mbie. z0mbie's homepage. `http://z0mbie.host.sk` (Last accessed: 3 Feb. 2003).