Technische Universität
München

Fakultät für Informatik

Forschungs- und Lehreinheit Informatik VII

Model Checking Malicious Code

Diplomarbeit

Johannes Kinder

Themensteller:	Prof. Dr. Helmut Veith
Betreuer:	Dr. Stefan Katzenbeisser
Abgabedatum:	17. Mai 2005

Erklärung: Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 17.05.2005

Abstract

Recent years have seen a dramatic increase of security incidents on the Internet related to e-mail worms. These particular pieces of malicious code are often developed by mischievous teenagers and are not very skillfully engineered, but still spread globally in a matter of minutes and cause a large amount of economic damage.

Conventional anti-virus products nowadays still rely on static pattern matching. They do not detect a new worm variant as long as the binary representation is sufficiently different, even if the functionality of the worm has not significantly changed. As a result, e-mail worms remain undetected during the phase of their initial outbreak, successfully reaching hundreds of thousands of mailboxes.

In this thesis, a novel contribution to the field of semantic malware detection is presented. This approach employs model checking, a well proven formal verification method, to find behavioral patterns inside an executable. In particular, the new temporal specification logic CTPL is introduced, which allows a succinct representation of program behavior on assembler level. A specialized model checking algorithm for CTPL allows efficient validation of disassembled executables against malicious code specifications.

In the course of this thesis, the CTPL model checking algorithm has been implemented in the Java programming language. Two exemplary specifications of malicious behavior are created and tested with the prototype on a set of current e-mail worms as well as on benign programs. The positive results prove CTPL model checking to be a very promising approach for the sophisticated detection of viruses and worms.

Acknowledgements

First of all, I would like to express my gratitude towards Prof. Dr. Helmut Veith for giving me the opportunity to work on an exciting subject, for his continuing enthusiasm and support of this thesis, and his patience in giving me advice on all topics of logics and model checking. His exceptional dedication to the theoretical computer science group created a very pleasant working environment for everyone.

I also want to thank my supervisor, Dr. Stefan Katzenbeisser, for guiding me through this thesis, for contributing fresh perspectives and inspiring ideas, and of course for a great deal of proofreading and advice on the twists of the English language. Furthermore, I would like to thank Ikarus Software and Dr. Christopher Kruegel for providing me with a large set of malware samples, which allowed me to test and improve my specifications.

Finally, I wish to extend my thanks to my parents for supporting me throughout my studies and to my friends for further proofreading and their words of encouragement.

Johannes Kinder
May, 2005

Table of Contents

List of Figures

Chapter 1

Introduction

The Internet of today connects a vast number of household- and business-owned personal computers mostly running Microsoft Windows operating systems on x86-compatible architectures. Recent security incidents have shown that this monoculture is a very attractive target for maliciously aligned individuals developing worms—programs that spread autonomously over network connections requiring little or no user interaction.

1.1 Malware

Apart from worms that exploit vulnerabilities in network services, such as *Sasser*, *SQLSlammer*, and *Code Red*, the most successful and widespread pests have been e-mail worms, which in most cases simply rely on users opening attachments to e-mails out of curiosity. Replicating with this rather unsophisticated method, various versions of *NetSky*, *MyDoom*, and *Bagle* have been dominating the malware statistics for over a year (Figure 1.1). MyDoom alone is estimated to have caused a total economic damage of around 3 billion US Dollars during the phase of its initial outbreak [mi204].

The damage caused by these worms is completely unproportional to the amount of creativity and skill used for programming them: In contrast to the 'classic' viruses of the pre-Internet era that spread by infecting executable files, creating an e-mail worm that infects hundreds of thousands of computers does not require knowledge of system programming or even assembly language. MyDoom and NetSky, for example, are written in Visual C++, and do not appear to be very skillfully engineered, containing obvious bugs

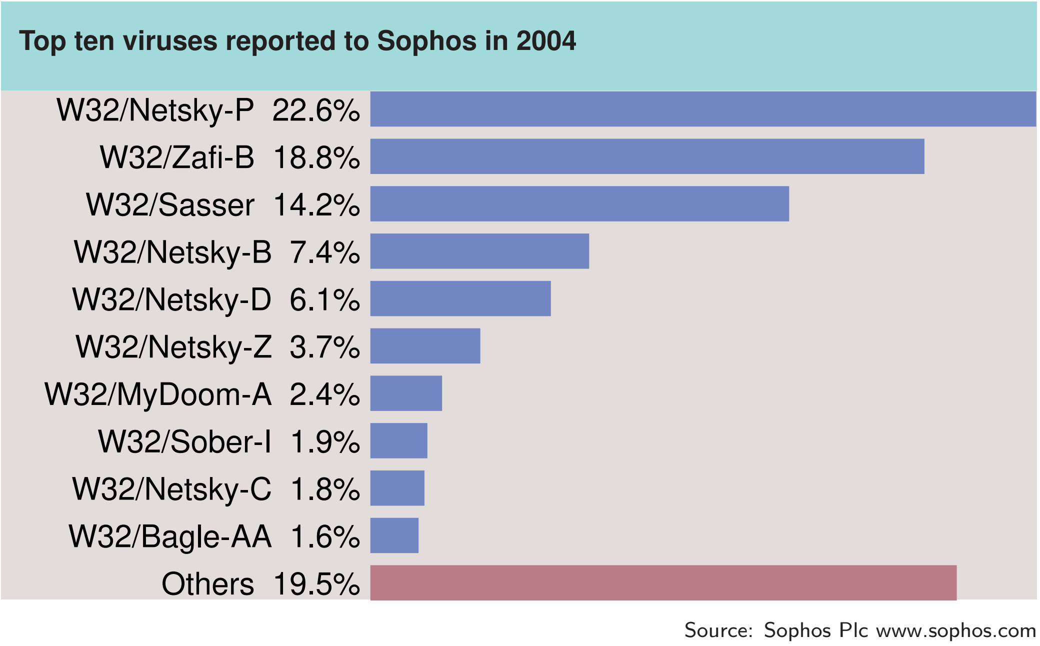


Figure 1.1: Most prolific viruses in 2004.

in some versions. During the last years it became evident that shortly after a new virus is released into the wild, several modified versions of the virus appear. These variants do not need to originate from the same author—typically, the source code of a worm is spread via Internet forums or even within the worm itself. That way, the level of programming skill required for triggering the next worm epidemic is very low. As a result of these developments, we see new worm derivatives appearing on the Internet almost every day. While these new versions differ only slightly from the original in terms of functionality, the resulting binary file can change significantly, depending on the compiler used and its optimization settings; this problem is further aggravated by the use of *executable packers* such as UPX [OM] or FSG [Xtr].

1.2 Detection Methods

1.2.1 Signature Matching

Current anti-virus products use rather inflexible but efficient static signature matching as their principal detection method. Static signature matching employs a database containing characteristic binary code sequences of known malware and matches these sequences against executables. This database of virus signatures has to be kept up-to-date in order to be able to detect viruses that emerge after the release of the anti-virus software. Keeping the false positives rate low is vital to the productive use of virus scanners; as a result, signatures are chosen very narrowly so that each of them matches exactly one single version of a specific malware. Consequently, a signature designed for one version of a virus or worm will usually not match against derivatives thereof. Moreover, the modifications do not even have to be substantial—Christodorescu and Jha showed that solely inserting a `nop` instruction into the binary code might render the virus undetectable for commercially available virus scanners [CJ04]. This problem is addressed by the anti-virus community through releasing updates to their signature databases as quickly as possible, usually daily or in some cases even several times a day. However, there will always be a significant window of vulnerability between the release of a mutated worm and the database updates, during which the malware remains undetected by conventional virus scanners. It would thus be highly desirable to have a scanner that reliably detects a virus or worm together with a large class of its potential derivatives.

1.2.2 Dynamic Analysis

Dynamic analysis is a relatively new approach to malware detection and solves some of the problems of static signature matching by using a more general behavioral approach. The core component of a dynamic scanner is the *sandbox* [Nor03], a virtual machine that fully simulates an environment for the executable to be checked. The sandbox also includes files with bait e-mail addresses and simulates a SMTP-server to provide a potential worm with a realistic computation environment. While the executable runs on this virtual machine, the scanner looks for suspicious interaction with the artificial environment. For example, if the tested program sends emails with executable attachments to addresses taken from the bait files, this behavior will qualify as malicious.

However, while this is certainly an interesting approach to malware detection, it can easily be foiled as it faces a fundamental problem: Pure observation of the behavior of an executable will always be limited to a certain time span, and nothing can be predicted about future actions of the program. For example, a program might set a timer to start its malicious actions five minutes after program start; or, even simpler, a malicious software could decide randomly when to run its payload. Evidently, a detection method has to be sought that considers all possible operational sequences in the program.

1.2.3 Semantic Analysis

Sometimes also referred to as 'static analysis', semantic analysis methods try to match semantic program behavior against general malicious actions. However, contrary to dynamic analysis, this is achieved without actually running the code, but rather by analyzing the possible program flow inside an executable. Therefore, semantic analysis has the potential of solving the problems of both signature matching and dynamic analysis:

- Malware definitions can be more general, matching a large class of viruses or worms.
- All possible program paths are checked, not only a limited time span.

Semantic analysis methods can be based on model checking [CGL99], a well proven formal verification method that is widely used for verifying properties such as fairness or liveness in hard- and software. Detecting viruses by semantic analysis is a new and still scarcely researched area, but modern advances in computing power make it a feasible and promising aim. In this thesis, a new semantic analysis method is introduced that checks executables against malware specifications defined in an extension to the temporal logic CTL, called CTPL (introduced in Section 3.2). CTPL allows powerful and succinct specifications of malicious program behavior.

1.3 Previous Work

The shortcomings of classic malware detection techniques are well known. Sophisticated semantic analysis methods for virus detection have been declared the goal of anti-virus research several times. While some approaches to this goal have been discussed in the literature, a decisive breakthrough has yet to be achieved.

Bergeron et. al. [BDD$^+$01] extract the control flow graph of an executable and reduce it to a subgraph whose nodes represent selected system calls. This subgraph is then checked against a description of suspicious behavior, which is defined as specific sequences of system calls. However, they do not perform any kind of data flow analysis and are not able to find dependencies between these calls. In particular, they give an example of a suspicious system call sequence that sends data over a network connection after reading from a file; however they cannot tell whether the data actually originated from the file due to the nature of their method.

Singh and Lakhotia [SL03] use a control flow graph reduced to basic blocks; a program can be partitioned into basic blocks by cutting the code sequence behind all instructions that influence the control flow (i.e. jump instructions) and before every jump target, thus creating a graph in which each node represents a set of strictly sequential instructions (a basic block). Together with annotations to these basic blocks supplied by the IDAPro disassembler they translate the graph into input to the model checker SPIN to check it against a formula in linear temporal logic (LTL), which specifies viral behavior. However, in [LS03] they express serious doubt about the feasibility of this method and generally of malicious code detection by formal analysis.

Christodorescu and Jha [CJ03] concentrate on fighting common virus obfuscation techniques such as register renaming or the insertion of dead code (e.g. `nop` or `mov eax,eax`) and unconditional jumps into the virus body. To this end, they transform the disassembled program code into an automaton where abstracted assembler instructions constitute transitions between states. Another automaton is built from the code of the virus to be detected in a similar way, but with an additional loop in every state that allows any number of irrelevant instructions (dead code) to occur. For resistance against register renaming, registers in the virus automaton are replaced by unresolved symbols. In the detection phase, they check whether the languages specified by the malicious code automaton and the program automaton have a non empty intersection with respect to the possible variable bindings. Using their approach, they can decide whether the executable contains a version of the virus obfuscated by means described above. However, they do not allow meaningful code between individual states and thus are not able to detect versions that contain extended functionality besides any obfuscation transformations.

1.4 Motivation

Theoretical results [Coh87] show that it will never be possible to create a universal malware detector, as this problem is undecidable. Creation and detection of malicious code is thus destined to be a race of arms, where each camp has to counter the other's newly developed methods. However, at this time, the balance has shifted strongly in the direction of malware writers. Current virus detection is still rather primitive and far from utilizing the full potential of modern methods of program analysis. With conventional detection methods, the anti-virus industry has not been able to adapt to the pace of worms spreading around the globe in a matter of minutes.

As a result of the failure to implement new methods of malware detection, the skill level required nowadays for releasing a new worm is highly unproportional to its effect: The most successful e-mail worms of today are not particularly well engineered, and still manage to initially remain undetected as long as their binary code differs enough to foil signature based scanning. These e-mail worms, oftentimes released by teenagers with no more than high school computer science knowledge, contain absolutely no novel propagation or stealth techniques. They are nearly indistinguishable in terms of functionality, varying only in payload and the text that is sent in the e-mails, yet outbreaks of these worms cannot be contained.

The goal has to be to significantly raise the level of effort needed to create successful malware, since signature matching has already reached its limits. Malware detection by semantic methods, however, seems to be the most promising approach to achieve measurable progress in fighting malicious code.

Chapter 2

Technical Analysis

Before focusing on the theoretical and formal aspects of malicious code detection by model checking, we will take a look at the technical foundation of the approach.

2.1 Reverse Engineering

The term *reverse engineering* describes the process of analyzing a hard- or software product in order to gain insight about the inner functionality. It is most often heard of in the context of companies trying to reduce development costs by utilizing code belonging to their competition, or of software pirates cracking copy protections or license limitations of programs. However, it is also a way to understand the mechanisms of malicious software, and the prerequisite for automated semantic virus detection tools.

Standard executables on a 32 bit Windows system are encoded in the Portable Executable (PE) format. PE files contain a header with various information about the file, such as the entry address or relocation information, one or more code sections with the actual binary program code, and multiple data sections that hold the import tables, resources (e.g. icons), or other data such as static variables and constants. Pietrek [Pie02] provides a detailed view on the PE format.

The executable code inside a PE-file is a continuous byte stream, that, after being loaded into system memory, is started by calling the entry address. Binary instructions vary in length, depending on parameter number and type. Hence, after reading the byte located at the address contained in the instruction pointer (IP), the CPU has to decide at runtime whether and how many more bytes have to be loaded to complete the instruction. However, to be able to perform a static semantic analysis, it is important to have a code representation that is a sequence of instructions as opposed to a sequence of bytes. Consequently, a *disassembler* has to statically transform the byte sequence into an assembler program. During this process, it resolves potential ambiguities by a best effort method. There are two main strategies for disassembling a binary executable [LD03]:

- The *linear sweep* disassembler starts at the entry point and decodes the instructions strictly sequentially, assuming that every instruction is aligned to the next. This rather simple algorithm is very efficient (it requires only linear time), but it is not able to detect inlined data in the code segment. As long as the data is skipped at runtime by a jump instruction, inlining data is a perfectly legal practice that can be encountered in an executable when the compiler inserted alignment bytes or generated a jump table from a `case` statement.

- The evident solution to this problem is interpreting control flow changing instructions during disassembly: The *recursive traversal* algorithm follows the control flow and resolves the target address whenever a jump instruction is encountered and starts disassembling another control flow branch from there.

In case of indirect jumps (e.g. jumps to addresses calculated in registers), things become more difficult, as the resulting address cannot be statically resolved. Common ways to solve the problem include processing of all sections marked as code even if they are unreferenced. Because these portions of code are not referenced, the entry points may not be obvious; therefore, the disassembler looks for common code patterns in the binary and determines in multiple passes from which entry point the binary decodes to meaningful assembler code.

Probably the worst problem for disassemblers is self-modifying code that is produced or changed at runtime, and executed afterwards. While modification of code segments might be forbidden on certain systems, the instruction encoding always allows to jump right in the middle of an earlier binary instruction. In this case the same bytes are executed again, but this time they are decoded into completely different instructions. Figure 2.1 illustrates an

Disassembled binary:

offset	binary	assembler
`0000:`	`66 33 C0`	`xor eax, eax`
`0003:`	`66 B9 B8 05 00 CB`	`mov ecx, 0CB0005B8h`
`0009:`	`66 F7 E1`	`mul ecx`
`000C:`	`66 85 C0`	`test eax, eax`
`000F:`	`74 F4`	`jz 05h`

Execution trace:

offset	binary	assembler
`0000:`	`66 33 C0`	`xor eax, eax`
`0003:`	`66 B9 B8 05 00 CB`	`mov ecx, 0CB0005B8h`
`0009:`	`66 F7 E1`	`mul ecx`
`000C:`	`66 85 C0`	`test eax, eax`
`000F:`	`74 F4`	`jz 05h`
`0005:`	`B8 05 00`	`mov ax, 05h`
`0008:`	`CB`	`retf`

Figure 2.1: Ambiguous assembler code.

example of such code—the final `jz` jumps to the third byte of the `mov` instruction in line 2, which reveals the instructions `mov` and `retf` that were not visible initially. However, a very high amount of skill is required for creating working code with one of these methods, and handcrafting larger malicious programs one byte at a time is not feasible in reality.

These problems make the problem of disassembling executables not simply a matter of mapping bytes one to one to instructions, but a generally undecidable problem, similar to program equivalence. A disassembler has to put up with these difficulties and try to produce an output that is as close as possible to an actual assembler source file that would compile to the same executable again. Moreover, it aids analysis by also processing information other than the code sections, notably the import section which contains all system and library calls used by the program. For improving human readability and supporting automated analysis, calls to functions imported from known libraries may be resolved to reveal the underlying name instead of showing just the ordinal identification number.

Despite the theoretical limitations, current disassemblers are able to correctly reverse engineer nearly all programs that have been created using common compilers (after unpacking, see next section). In 2004, this included at least 80% of all currently prevalent worms: Of all worms listed in Figure 1.1, only Bagle has been programmed in assembly language.

A completely different problem is the reverse engineering of programs using a just-in-time compiler or some kind of interpreter language (such as Java or Perl). These qualify as self-modifying code in the way that code is created at runtime, and are thus not susceptible to static disassembly. However, as their source code has to be available to the just-in-time compiler, they will be vulnerable to analysis methods specially tailored towards their implementation language.

2.2 Protection Schemes

Since reverse engineering is a threat to the intellectual property of commercial software, there has been significant research in protecting executable code against such analysis. For obvious reasons, vendors of commercial copy protection libraries, such as dongle systems or CD-based copy protection schemes, have a particularly high interest in supporting development in that area. As a result, a wide spectrum of both commercial and open source tools exists that can be used to make debugging and reverse engineering of software a non-trivial task. However the use of such tools is not limited to protecting commercial software—in fact most malicious code encountered in the wild nowadays is at least protected by the use of an *executable packer*.

2.2.1 Executable Packers

The protection of most pieces of malware is limited to the use of one of the freely available executable packers such as `UPX` [OM] or `FSG` [Xtr]. The open source tool UPX seems to be particularly popular with malware writers, and is very commonly found wrapping worm variants. A list of the packers used by common malware can be found in Table 2.1. As the name suggests, their classical purpose is simply the reduction of an executable's size. Possible uses are to save bandwidth when a file is published on the Internet, or to use less space on a limited data storage, such as a USB stick. For example, the Windows installer of the Mozilla Firefox browser and the setup program of

Malware family	**Versions**	**Executable Packer**
Bugbear	a, b, e	UPX 0.x/1.x
	h	FSG 1.33
Dumaru	c, g, h	UPX 0.x/1.x
	y, z	FSG 1.33
Klez	a, e, g, h	None
MyDoom	a, f, g, h, j, k, n, o, r, s, t, u, w, x, af, ag	UPX 0.x/1.x
	m	FSG 2.0
NetSky	b, l, m, o, w	UPX 0.x/1.x
	c	ASPack 2.12
	p, r	FSG 1.0
	f, h, y	PE Pack 1.0
	z, aa, ab, ac	PECompact 2.x
	d, e, q	PEtite 2.2
	j, k, n, x	tElock

Table 2.1: Executable packers used with malware.

the Cygwin environment are both packed with UPX. They compress—and in most cases also encrypt—an executable and add an extraction routine to the compressed file. Everytime the packed executable is run, this routine decompresses and decrypts the original binary into system memory and cedes control to it afterwards.

An obvious attack to this kind of protection is simply allowing the program to load, and dumping an image of the process memory with a tool such as `ProcDump` after the decompression part is finished. The dumped executable might require some additional fixing of header structures, but the code itself is visible in its original form and susceptible to reverse engineering and static analysis. This simple method works for most kinds of executable packers and encryptions, as the unpacking function typically extracts the complete program right at start, and does not interfere with later computations.

The drawback of this method is that the executable must be loaded, which might not be acceptable in all cases as it cannot always be guaranteed that the program is terminated before any malicious functionality is executed. Besides using a virtual machine to avoid potential damage, it is also possible to unpack the executable by creating a separate tool out of the information gained from the unpacking routine included in the program. There are already many unpackers for the common packing utilities, such as `UPX`, `FSG`, `ASPack` or `Petite`. `UPX` even offers a decompression command by itself, however the packed file can be altered in such a way that `UPX` does not recognize it anymore.

2.2.2 Code Obfuscation

More sophisticated techniques for protecting code against reverse engineering are called *code obfuscation*. According to Collberg, Thomborson, and Low [CTL97], an obfuscator transforms a program into an obfuscated program that displays the same observable behavior but is illegible. They measure the quality of an obfuscator by its *potency*, *resilience*, and *cost*. In their definition, potency is the amount of subjective complexity added to a program, thus making it harder for humans to comprehend the functionality. Resilience describes the robustness of the obfuscation against automated deobfuscation methods. Finally, cost refers to the magnitude of additional time and space consumption caused by the transformation.

Various possibilities for obfuscating code have been identified in the literature [CJ03, CJ04, LD03]. In the scope of this work, two general kinds of obfuscation methods are of particular interest: First, those obfuscating transformations that foil the usual signature based malware detection techniques, and second those that thwart the disassembly of an executable, which is the prime prerequisite for employing semantic detection methods. Techniques belonging to the first group are:

- *Dead code insertion.* This method inserts instructions or sequences of instructions without effect on the machine state at random points in the program. Examples for dead code on x86 architectures are the `nop` instruction or statements such as `mov eax,eax`. Dead code insertion changes the binary footprint of a piece of malicious software, yielding false negatives in traditional anti-virus products.

- *Code reordering.* A sequence of binary code can be fractured into several pieces and put together in a random order by connecting subsequent instructions in the original code through unconditional jumps. As long as addresses used in the code are rewritten during the process, the program semantics is not affected even though the resulting binary executable is very different.

- *Register substitution.* The x86 architecture features several general purpose registers, which are fully permutable, with some exceptions such as loop control or mixed 8/16/32 bit math operations. For example, by changing all occurrences of `ebx` to `edx`, code can be transformed into a semantically equivalent but binary different program.

- *Instruction substitution.* In large instruction sets such as those of the x86 processor family, various different instructions can be used to perform equivalent operations [Int04]. The following three assembler code snippets illustrate this fact—they all push a constant value of 0xBEEF onto the stack, but each one of them uses different instructions to achieve that goal, resulting in different binary representations:

	`mov ax,0xBEEF`	`mov [esp],0xBEEF`
`push 0xBEEF`	`push ax`	`sub esp,0x02`

The second group of obfuscations fight the successful disassembly of a binary executable. To outsmart a linear sweep disassembler (see Section 2.1), it is sufficient to insert an illegal instruction into the code and precede it by a jump that skips it during normal execution. Recursive traversal is able to cope with such problems, however its more ‘intelligent’ approach also allows for possible attacks:

- *Misusing control flow statements.* Advanced disassemblers rely on the fact that procedures are called by `call` and return with a `ret` statement. However, these are only unconditional jumps using the stack to save and restore the program counter and may as such be misused to execute jumps to arbitrary targets. In particular, any `jmp` x may be replaced by `push` x followed by `ret`.

- *Unreachable conditional branches.* By using conditional jumps depending on a condition which always evaluates to a constant value already known at programming time, a disassembler can be tricked into exploring a branch that would never be executed at runtime and thus can contain illegal instructions causing the disassembler to report an error.

- *Branch functions.* In this method, jump instructions in the program are replaced by a call to an intermediate function, the branch function, whose sole functionality is to forward control to the correct target address for this jump. This can be done by calculating the target address from an argument passed to the branch function, or by using a lookup table.

While there is evidence that perfect obfuscation of program behavior is theoretically unachievable [BGI+01], a disassembler will still be unable to produce correct output for a thoroughly obfuscated program, thus making static analysis extremely difficult or even impossible.

2.2.3 Solutions

Malware detectors relying on signature matching may be foiled by the obfuscation techniques in the first part of the last section. However, semantic methods, such as the approach described in this thesis, are mostly resistant against these transformations. Obfuscators based on instruction substitution may at a first glance seem to be an exception, but their power is dependent on static tables of equivalent assembly code. The obfuscator uses such tables to choose randomly among equivalent instructions for every instruction it translates. However this is a static process, and conversely these tables might as well be used by a detector to generate abstracted, non-ambiguous pseudo-assembly code from a binary.

Strong obfuscation methods as in the second part, though, practically enable a skilled author of malicious code to prohibit successful disassembly and thus semantic analysis of his code. However, the practical impact may be rather small: On the one hand, the vast majority of malware plaguing the networks at the time of writing is *not* very well engineered. It is evident that the authors usually lack the skill to incorporate strong obfuscation methods that go beyond the use of simple packing tools. On the other hand, while the actual functionality of obfuscated code might be resistant to disassembly and analysis, the sheer fact that it is strongly obfuscated is often a reliable indicator for an untrustworthy piece of software.

2.3 Malware Analysis

In this section we take a closer look at some of the most important worm families of the last two years, which will be used as exemplary targets for the malware detection technique proposed in later chapters. Each one consists of several versions spawned from the same codebase, either by the author updating and improving his worm code, or by other virus writers who gained access to the source.

2.3.1 NetSky

NetSky serves as an excellent example of a typical e-mail worm, and as such will be analyzed in detail. From February 2004 until his arrest three months later in early May, the author of NetSky, an 18-year-old German student, released 24 versions of the worm, each of which needed new signatures to be detected by virus scanners. The quick succession of new variants combined with their fast proliferation clearly showed the deficiencies of non pro-active virus scanners, that failed to contain the outbreak. According to the virus wildlist [Wil], sightings of nearly every version of NetSky are still being reported after one year. This makes NetSky probably the most 'successful' worm of 2004.

Despite of its success, NetSky is neither particularly innovative nor does it take too much care to hide itself from detection. Protection is limited to the use of executable packers; the author seems to have changed the compression tools a lot in order to minimize the similarities of the binary code in the different versions. NetSky is written in Microsoft Visual C++ and most versions rely on social engineering for propagation: Sending mails with executable attachments designed to arouse the interest of the victims and thus tricking them into executing the worm. NetSky variants a, b, and c also copy themselves into shared folders of popular peer to peer file sharing programs, using various names suggesting pornographic content or 'cracks' to popular commercial software. In addition, the worm tries to terminate processes matching names from a hard coded list of anti-virus monitoring software and also removes instances of other malware such as Bagle variants and MyDoom. The author seems to have seen himself in a competition against other worm authors over the top ranks in the virus threat lists, or,

```
char[] mailAddresses;
boolean[] mailFlag;
int addressCount;

int WinMain(int hInst, int hPreInst, char* args, int nCmdShow)
{
  int inetFlags;

  InitRandom(GetTickCount());
  CreateMutex(0, 0, "AdmSkynetJklS003");
  if (GetLastError() == ERROR_ALREADY_EXISTS) return 0;

  if (strlen(args) <= 0)
    MessageBoxA(0, "The file could not be opened!",
                "Error", MB_ICONERROR);
  mailAdresses[0] = "skynet@skynet.de";
  addressCount = 1;
  Infect();       /* Copies worm to Windows directory, modifies
                     registry to autostart the worm and
                     removes autostart entries of anti-virus
                     shields and other worms.                */
  ScanDrives();   /* Copies worm to shared folders and harvests
                     e-mail addresses. The total number is
                     stored in addressCount                  */
  for(i=0; i<addressCount; i++) {
    while (!InternetGetConnectedState(&inetFlags, 0)))
      Sleep(50);
    if (!mailFlag[i])
      if (SendMail(mailAdresses[i]))  /* sendMail generates
           a mail with a spoofed sender address chosen randomly
           from the addressList, tries to send it to the
           specified address and returns 1 on success        */
        mailFlag[i] = 1;
    Sleep(50);
  }
}
```

Figure 2.2: Reconstructed `WinMain()` function of NetSky.b.

as he states in strings embedded in some of the versions, because he wanted his creation, in some twisted way, to be seen as a virus removal tool [Ciu04].

Figure 2.2 shows the `WinMain` function of NetSky.b that was reconstructed from the disassembly and thus should resemble the original C++ code. After initialization of the pseudo random generator, it checks via a system mutex whether the same NetSky version is already active on this machine, to allow only one running instance at the same time. To discourage possible suspicion by the user expecting an e-mail attachment to open, the worm displays an error message. Most victims will not take special notice of this incident, as undisplayable content is a rather common error they are accustomed to. NetSky continues by copying itself to the Windows directory—an action, which is common to a large range of worms and will be the basis for the malicious behavior specification created in Section 3.2.6—and modifying the registry to have itself started after every system boot as well as removing auto-start entries of anti-virus software and a set of other viruses.

After successful infection, the worm starts recursing through all directories on all fixed disks looking for text files, which it subsequently scans for e-mail addresses. Along the way, the worm also copies itself under random names into every folder found that contains either the string 'share' or 'sharing'. Finally it enters an infinite loop in which NetSky waits until a live Internet connection is found and iteratively sends itself to the mail addresses found, remembering successful deliveries to avoid multiple mailings during one instance—on the next system startup, however, the whole process is repeated. Regardless of the addresses found, the list of emails always starts with a hard coded address pointing to a freemail account which belongs to the author. This way he receives an email to this account for every newly infected machine, thus being able to monitor the outbreak of NetSky. However these accounts were always closed down rather quickly after a thorough analysis of the worm by security firms, so the addresses changed with every new version of the worm.

Later versions of NetSky are a little more advanced and use multiple threads to harvest addresses and send mails, and also contain a payload that emits sound from the system speaker during a certain time period. Some versions may also include a separate backdoor that is extracted and installed when the worm is run, or launch a denial service attack on several websites on certain dates.

2.3.2 MyDoom

Soon after its appearance in January 2004, MyDoom.a broke all existing records becoming the fastest spreading worm of all time. During its initial proliferation phase, MyDoom was allegedly responsible for up to 20% of all mail traffic on the Internet. MyDoom is very similar to NetSky in terms of functionality: It spreads primarily via e-mail attachments but also copies itself to the shared folder of the Kazaa file sharing program. It contains a simple payload that launches a denial of service attack at a specific time. Most versions are packed with UPX, and MyDoom.a additionally employs a simple ROT13 encoding to hide the registry keys and other sensitive strings it uses [Sza04b].

MyDoom.a installed a backdoor on each victim machine, which of course created a tempting target for other malware writers due to the sheer number of infected hosts. Soon a new generation of malware emerged that replicated by exploiting this new widespread vulnerability—for example Doomjuice (allegedly from the same author as MyDoom), Vesser, Welchia, or some versions of Agobot [Sza04a]. Doomjuice.a also dropped the source code of MyDoom.a on all infected hosts, which effectively made the source freely available to everyone interested. As a result, MyDoom has seen a large number of different versions, and is at the time of writing obviously still being developed, as new MyDoom versions are still released from time to time. The most recent version, MyDoom.bh, was discovered on March 21, 2005.

2.3.3 Klez

Klez was named the most prolific malware of 2002 by several anti-virus companies, having topped the infections charts for 15 consecutive months. Aside from relying on the usual social engineering concept, it tries to exploit a security hole in certain Outlook express versions that causes the attachment to be automatically executed upon reading the mail. Starting with version Klez.e, it also copies itself to network shares, and replaces executable files with itself, preserving the original file under a different name (*companion infection*). These versions also contain a particularly destructive payload that overwrites data files such as source code files, spreadsheets, and MP3 files found on the hard disk. Klez also drops the (seriously bugged) Elkern virus on infected systems, a classical file infector that spreads independently from the worm. Both pieces of malware are written by the same author, and newer Klez variants also carry improved versions of Elkern.

The motivation of the Klez author seem just as naive as those of the NetSky creator two years later. He tries to gain a positive image by terminating the processes of some other malware and even asks for employment in hidden messages embedded in the worm, disregarding the amount of damage he caused by releasing the worm. Besides terminating the processes of Sircam, Nimda, and the like, Klez also looks for running anti-virus programs and kills their processes. At that time, anti-virus software was not necessarily immune against this type of attack, which might be one of the reasons why Klez.h infections were top ranking even one year after updated signatures had been released.

2.3.4 Dumaru

The Dumaru family is a very heterogeneous pool of related worms and backdoors, that has had its high times in late 2003 and the first quarter of 2004. The worm code has obviously been available to a number of authors, as functionality and targets vary over the versions. Large portions of the code seem to have been taken from other malware. For example, the worm contains the NTFS-streams [Esp00] based file infection routine of the experimental W2K/Stream virus; however, the borrowed code has not been particularly carefully integrated, and is severely bugged, effectively reducing propagation by file infection to a minimum [Fer04].

Some variants drop a Trojan horse that logs itself into an IRC server and listens to remote control requests, while others feature an integrated IRC backdoor or a simple TCP interface that allows executing files or performing common pranks like opening the CD-ROM tray or playing sound files. Apparently all versions also try to harvest sensitive information such as bank account credentials by logging key strokes inside windows whose titles match a string from a list of names of login windows of several popular banking and money transfer services. The harvested information is stored and sent in specific time intervals to a hard coded e-mail address. Some of the later Dumaru versions do not even contain propagation code anymore, but concentrate on the backdoor functionality instead.

This particular piece of malware leaves an overall impression of criminal intentions on the side of the different authors because of all the strategies to profit from their creation, in contrast to the obviously more naive and excited style of the NetSky author.

Chapter 3

Model Checking

The malicious code detection technique described in this thesis is based on the concept of model checking, which has classically been used as an efficient method to verify the concordance of systems to specifications, such as proving the correctness of protocols in concurrent environments. Section 3.1 serves as introduction to the general area of model checking, before the concept is adapted to suit the problem of malicious code detection in Sections 3.2 and 3.3.

3.1 Introduction

3.1.1 Preliminaries

In model checking, a system is described by a *Kripke structure*, which is essentially a labeled directed graph consisting of states and transitions between states.

Definition 3.1. *A Kripke structure M is a triple $\langle S, R, L\rangle$, where S is a set of states, $R \subseteq S \times S$ is a total transition relation, and $L : S \to 2^P$ is a labeling function that associates a set of propositions (elements of P, where P is the set of all propositions) to each state.*

The labeling reflects the truth values of the atomic propositions with regard to the system state. The labeling causes propositions to represent different truth values depending on the state in which they are evaluated.

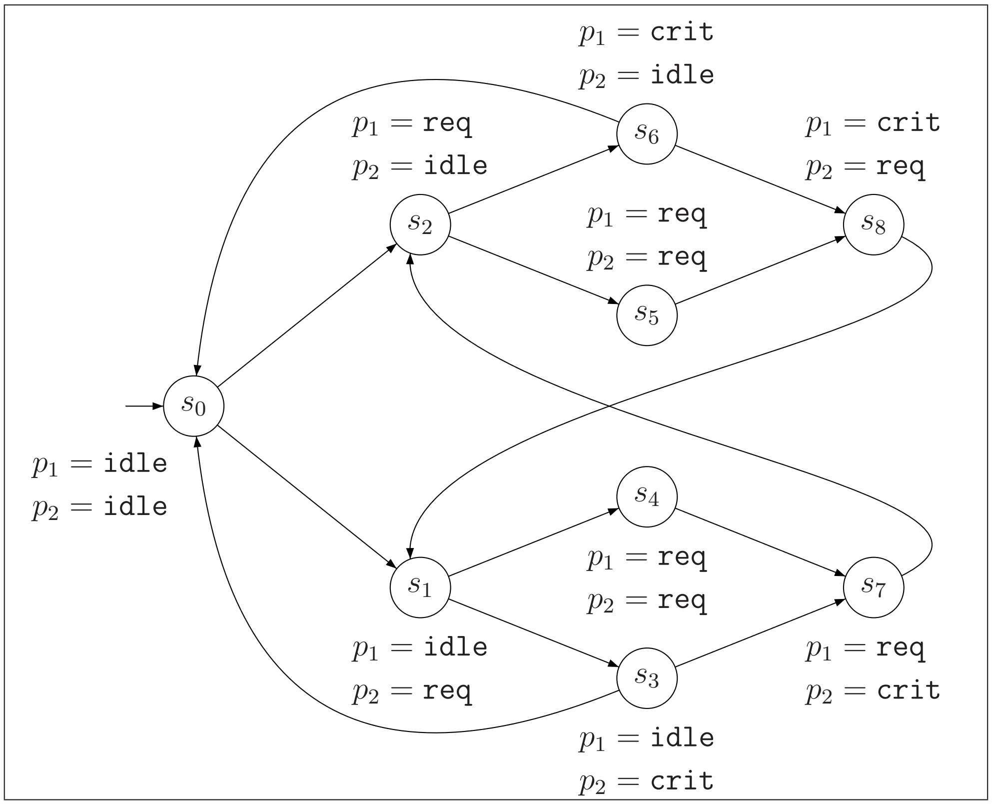


Figure 3.1: Kripke structure of a fair mutex protocol.

Definition 3.2. *A proposition p holds in s, if and only if p is contained in the set of labels of a state s of the Kripke structure M:*

$$M, s \models p \Leftrightarrow p \in L(s).$$

For reasoning about systems, the paths through a Kripke structure are of particular interest, as they denote the possible sequences of valid state transitions during execution of the program or protocol specified in the model.

Definition 3.3. *A path $\pi = s_0 s_1 s_2 \ldots$ in M is an infinite sequence of states $s_i \in S$ with $(s_i, s_{i+1}) \in R$ for each $i \geq 0$.*

As an abbreviation, we denote with π^i the state at position i in a path π, in such a way that π^0 is the first state in π. Π_s denotes the set of all possible paths in M starting in state s.

A Kripke structure that represents a fair mutex protocol between two processes p_1 and p_2 can be found in Figure 3.1. The propositions in this example are of the form '$p_1 = \texttt{idle}$', '$p_1 = \texttt{req}$', or '$p_1 = \texttt{crit}$', which represent that process 1 is idle, requests to enter its critical section, or is inside its critical section, respectively. Every state is labeled with two propositions that reflect the status of both processes. The mutex has been designed such that whenever a process requests to enter its critical section ('$p_i = \texttt{req}$'), it is eventually allowed to do so ('$p_i = \texttt{crit}$'). This property is called *fairness* and is generally desirable for concurrent systems. We can use temporal logics such as the one presented in the next section to formulate properties of this kind.

3.1.2 Computation Tree Logic

For reasoning about Kripke structures, we need a logic that allows to specify temporal properties of Kripke structures by statements such as '*there will eventually be a state in which the proposition p holds*'. This is possible with temporal logics such as LTL [Pnu81] or CTL [CE81]. CTL (Computation Tree Logic) is a branching time logic that allows to quantify over several paths originating from a state, while linear time logics only consider one path at once. Besides the standard propositional logic operators $\wedge$, $\vee$, and $\neg$, CTL offers six special temporal operators, **A**, **E**, **X**, **F**, **G**, and **U**. **A** and **E** are path quantifiers that intuitively express '*for all paths*' and '*there is a path*', respectively. **X**, **F**, **G**, and **U** are linear-time operators that can be used to specify properties along one path π.

Formally, the semantics of the individual temporal operators can be defined as follows:

- $\mathbf{A}\,\psi$ is true in a state s, written $M, s \models \mathbf{A}\,\psi$, if ψ is true for all paths in Π_s.
- In contrast, $M, s \models \mathbf{E}\,\psi$ holds if there exists a path in Π_s where ψ holds.

The linear time operators express properties of one specific path π:

- $\mathbf{X}\,\psi$ is true on a path π if ψ holds in state π^1, i.e. the *next* state from the path's initial state.
- $\mathbf{F}\,\psi$ is true if there exists a state somewhere along π where ψ holds (ψ *finally* holds).

- $\mathbf{G}\,\psi$ is true if ψ holds in all states of π (ψ holds *globally*).
- $\psi_1\,\mathbf{U}\,\psi_2$ is true if ψ_1 holds in all states on the path π *until* a state in which ψ_2 holds.

In CTL, path and linear-time operators can occur only pairwise in the combinations **AX**, **EX**, **AU**, **EU**, **AF**, **EF**, **AG**, **EG**; without this restriction, the logic is called CTL*, which is a superset of both CTL and LTL.

Coming back to the mutex example of last section (Figure 3.1), we are now able to formulate the fairness property in CTL as:

$$\begin{array}{rl} & \mathbf{AG}((p_1 = \texttt{req}) \rightarrow \mathbf{AF}(p_1 = \texttt{crit})) \\ \wedge & \mathbf{AG}((p_2 = \texttt{req}) \rightarrow \mathbf{AF}(p_2 = \texttt{crit})) \end{array}$$

The formula expresses that the following holds in every state on all paths: Whenever a process requests to enter its critical region, all subsequent paths will finally reach a state where this process does enter its critical region.

For CTL there exists a number of useful equivalences, which in particular allow to define an adequate set of temporal operators from which all other combinations of temporal operators can be derived, thus decreasing the number of operators needed to be implemented in a CTL model checker. These equivalences are (the proofs can be found in [HR00]):

$$\begin{array}{rcl} \neg\mathbf{AF}\psi & \equiv & \mathbf{EG}\neg\psi \\ \neg\mathbf{EF}\psi & \equiv & \mathbf{AG}\neg\psi \\ \neg\mathbf{AX}\psi & \equiv & \mathbf{EX}\neg\psi \\ \mathbf{AF}\psi & \equiv & \mathbf{A}[\top\,\mathbf{U}\,\psi] \\ \mathbf{EF}\psi & \equiv & \mathbf{E}[\top\,\mathbf{U}\,\psi] \\ \mathbf{A}[\psi_1\,\mathbf{U}\,\psi_2] & \equiv & \neg(\mathbf{E}[\neg\psi_2\,\mathbf{U}\,(\neg\psi_2 \wedge \neg\psi_1)] \vee \mathbf{EG}\neg\psi_2) \end{array}$$

Adequate sets of operators are e.g. **AU**, **EU** and **EX** or **AF**, **EU**, and **EX**. In particular, a set of temporal connectives is adequate if and only if it contains **EU**, at least either of **AX** or **EX**, and at least one of **EG**, **AF**, and **AU** [Mar01].

3.1.3 Model Checking Assembler Code

Model checking, as described in this section, allows to validate abstract systems defined by a Kripke structure against a formula defined in a specification

```
label1:  cmp ebx,[bp-4]
         jz j
         dec ebx
         jmp c
label2:  mov eax,[bp+8]
```

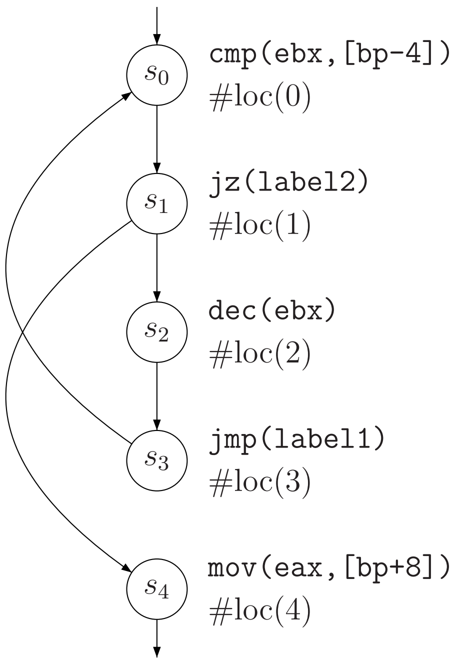


Figure 3.2: Executable code sequence and corresponding Kripke structure.

logic such as CTL. The Kripke structure is usually obtained by the model checker through an abstract notation of state transition systems. However, such an abstraction is not available when checking machine code.

The basic idea used in this thesis for applying model checking to malicious code detection is to transform a disassembled executable into a Kripke structure by creating a state for every instruction, and to label this state with two propositions: one representing the instruction (opcode and parameters), and one identifying the location (#loc(L)), which makes every instruction distinguishable from other, equal instructions. These states are then connected by edges:

- Every state that is labeled with an unconditional jump (i.e. `jmp`) is connected only to its jump target.
- States labeled with a conditional jump (such as `jz, jbe`) are connected to both the state representing their immediate successor in the disassembled procedure and their jump target.

- Return statements do not have a successor and are connected to themselves in a loop transition.
- Indirect jumps cannot be statically resolved and will become dead ends if they are unconditional. These dead ends are given a loop transition to avoid leaf nodes in the Kripke structure.
- Every other state is connected to its immediate successor.

Figure 3.2 demonstrates how a snippet of assembler code can be transformed into a Kripke structure. The Kripke structure generated this way can then be checked against a specification. In standard CTL, a formula specifying that the system function `DeleteFile` is called at some point in the future would have the form

$$\mathbf{EF}(\texttt{call DeleteFileA}).$$

While it would be possible to create specifications in pure CTL, there are significant drawbacks which will be examined further in the next section.

3.2 Computation Tree Predicate Logic

3.2.1 Why a new logic?

In this section we introduce the new branching time logic CTPL (Computation Tree Predicate Logic) as an extension to CTL. Despite being equally expressive as CTL, CTPL substantially simplifies the defining of specifications for model checking assembler code. The need for extending CTL becomes evident when considering the problem of detecting recurring behavioral patterns in different families and versions of malware. In a flexible specification logic, it has to be possible to formulate statements such as '*The value `2Fh` is assigned to some register, and the contents of that register is later pushed onto the stack*'. As the number of possible register assignments is finite, this can be formulated in CTL by simply enumerating all possible combinations in one large expression. The above example would result in this formula:

$$\begin{aligned}
&\mathbf{EF}(\texttt{mov eax,2Fh} \land \mathbf{AF}(\texttt{push eax})) \lor \\
&\mathbf{EF}(\texttt{mov ebx,2Fh} \land \mathbf{AF}(\texttt{push ebx})) \lor \\
&\mathbf{EF}(\texttt{mov ecx,2Fh} \land \mathbf{AF}(\texttt{push ecx})) \lor \\
&\ldots
\end{aligned}$$

Plain CTL formulas that model potentially malicious behavior will always be very large if these specifications are general enough to allow for register substitution (as explained in Section 2.2.2). In CTL, this is only possible by explicitly mentioning each possible register assignment. While this is already impractical in case of register substitution, it becomes outright infeasible when using memory variables in specifications. Using CTL, this would require creating a formula containing every possible memory address combination, or at least adapting the formula to each specific Kripke structure before model checking by generating clauses with all variable names found in the disassembly. Either way, it becomes obvious that it is certainly not realistic to specify malicious behavior for assembler programs in an instruction oriented way using standard CTL.

In order to account for these difficulties we introduce the temporal logic CTPL. CTPL is an extension to CTL, tailored towards specification of code patterns in a general and natural way. Even though CTPL is not more expressive than CTL, it allows specialized model checking algorithms to handle all possible combinations of register and memory variable assignments succinctly by introducing quantified variables in a formula.

3.2.2 Predicates

The transformation from assembler code to a Kripke structure M yields propositions of the form '`push eax`'. In the new logic, we want to be able to differentiate the instruction (`push`) from its parameters (`eax`) in the labeling of M. To this end, we allow propositions to be *predicates*:

Definition 3.4. *A predicate describes a property of a state with respect to zero or more parameters.*

Unlike first-order logic, where a predicate of arity 0 will become a Boolean constant, predicates may have zero parameters, as their truth value depends on the state in which they are evaluated. In particular, predicates of arity 0 resemble the propositions defined in Section 3.1.2.

The parameters of a predicate can be *constants*, which are elements of the infinite set $\mathcal{I}$:

Definition 3.5. *The set of all constants $\mathcal{I}$ is the infinite set of all arbitrary alphanumeric strings and numbers.*

In the context of modeling assembler programs as Kripke structures, $\mathcal{I}$ corresponds to all integer values, all memory addresses, all possible local variables, all register names, and combinations thereof (e.g. for indexed addressing). We now define a finite subset $\mathcal{U}$ of $\mathcal{I}$ that will be part of the extended Kripke structure introduced below:

Definition 3.6. *A universe $\mathcal{U} \subset \mathcal{I}$ is the finite set of constants available as parameters for predicates in labels of states in a certain Kripke structure.*

Again in the context of assembler programs, the universe describes the set of those constants, memory addresses, and registers in use by the actual program that is being analyzed. With the universe $\mathcal{U}$, we now define the set P' of all predicates over elements of $\mathcal{U}$.

Definition 3.7. *P' denotes the set of all predicates $p(c_1, \ldots, c_n)$ of arbitrary arity $n \geq 0$ with all possible combinations of parameters $c_i \in \mathcal{U}$, where $1 \leq i \leq n$.*

We are now able to give the extended definition of a Kripke structure M which allows predicates in labels:

Definition 3.8. *An extended Kripke structure M is a tuple $\langle S, R, L, \mathcal{U} \rangle$ with the set of states S, the total transition relation $R \subseteq S \times S$, the labeling function $L : S \to 2^{P'}$, and the universe $\mathcal{U}$.*

From now on we will refer to this extended definition when using Kripke structures.

3.2.3 Syntax

The specification logic CTPL is built as an extension to CTL, and as such allows all syntax elements known from CTL (see Section 3.1.2). In addition, we allow propositions to be *predicates* as defined above. Aside from constants, parameters of predicates in CTPL formulas can also be variables. We generalize variables and constants to be *terms*:

Definition 3.9. *A term t is either a variable or a constant, i.e. $t \in (\mathcal{I} \cup V)$, where V is the set of all variables.*

For easier identification, from now a r, x or y will refer to a variable, while t refers to a general term (either constant or variable). Constants may be numbers or words or combinations of both; where the actual value of a constant is not given, it will be represented by the symbol c.

Definition 3.10. *The syntax of CTPL is defined inductively:*

- $\top$ *and* $\bot$ *are CTPL formulas.*
- *If* p *is a predicate of the arity* $n \geq 0$ *and* $t_1, \ldots, t_n$ *are terms, then* $p(t_1, \ldots, t_n)$ *is a CTPL formula.*
- *If* ψ *is a CTPL formula, then* $\neg\psi$, $\mathbf{AX}\,\psi$, $\mathbf{AF}\,\psi$, $\mathbf{AG}\,\psi$, $\mathbf{EX}\,\psi$, $\mathbf{EF}\,\psi$, *and* $\mathbf{EG}\,\psi$ *are CTPL formulas.*
- *If* ψ_1 *and* ψ_2 *are CTPL formulas, then* $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\mathbf{E}[\psi_1\,\mathbf{U}\,\psi_2]$, *and* $\mathbf{A}[\psi_1\,\mathbf{U}\,\psi_2]$ *are CTPL formulas.*
- *If* ψ *is a CTPL formula and* x *is a variable, then* $\forall x\, \psi$ *and* $\exists x\, \psi$ *are CTPL formulas.*

3.2.4 Semantics

In order to define the semantics for CTPL, we first introduce a set of preliminary definitions for later use. These are related to similar concepts in first-order logic.

As CTPL allows variables, we need a mapping that collects assignments to free variables, called the environment $\mathcal{B}$, similar as in the semantics definition of first-order logic [HR00]. The environment contains pairs of variable names and constants (from the universe $\mathcal{U}$) and can be implemented as a lookup table.

Definition 3.11. *The environment* $\mathcal{B}$ *is a partial function that maps free variables to constants from the universe* $\mathcal{U}$*, and by definition every constant to itself.* $\mathcal{B}[x \mapsto c]$ *denotes the environment that maps the variable* x *to the constant* c *and every other variable* y *to* $\mathcal{B}(y)$*.* $\mathcal{B}(t)$ *represents the value mapped to the term* t *in the environment* $\mathcal{B}$*.*

We now define a modeling relation that takes the environment into account when evaluating the formula:

Definition 3.12. *$M, s \models_{\mathcal{B}} \varphi$ denotes that the formula φ holds at state s in the Kripke structure M with respect to the environment $\mathcal{B}$.*

The semantics of CTPL is shown in Figure 3.3; in most parts, this definition is similar to the one of CTL, modified only to respect the environment $\mathcal{B}$. Rule 1 initializes the environment by implicitly existentially quantifying any free variables. Rule 6 defines the quantifier $\forall$ in the way that $M, s \models_{\mathcal{B}} \forall x\, \psi$ holds if ψ holds with respect to all environments that extend $\mathcal{B}$ by a mapping of x to an element of the universe $\mathcal{U}$. Rule 7 handles $\exists$ analogously. Rule 2 defines the semantics of CTPL predicates: A predicate $p(t_1, \ldots t_n)$ over the terms $t_1, \ldots, t_n$ (variables or constants) evaluates to true in a state s with respect to a binding $\mathcal{B}$ if and only if s is labeled with $p(\mathcal{B}(t_1), \ldots, \mathcal{B}(t_n))$, where $\mathcal{B}(t_i)$ expresses the constant value of the term t_i in the environment $\mathcal{B}$.

3.2.5 Equivalences

Two formulas ψ and φ are semantically equivalent if for every M and every s we have $M, s \models \psi \Leftrightarrow M, s \models \varphi$. All equivalences for CTL given in Section 3.1.2 also hold in CTPL. In addition, the variable quantifiers provide for some more equivalent expressions that can be useful when creating specifications.

Theorem 3.13. *If $\psi(x)$ is a formula containing the free variable x, and φ a formula that does not contain x, then the following equivalences hold:*

$$
\begin{array}{lrcl}
\text{(i)} & \mathbf{AG}(\forall x\, \psi(x)) & \equiv & \forall x(\mathbf{AG}\, \psi(x)). \\
\text{(ii)} & \mathbf{AX}(\forall x\, \psi(x)) & \equiv & \forall x(\mathbf{AX}\, \psi(x)). \\
\text{(iii)} & \mathbf{EF}(\exists x\, \psi(x)) & \equiv & \exists x(\mathbf{EF}\, \psi(x)). \\
\text{(iv)} & \mathbf{EX}(\exists x\, \psi(x)) & \equiv & \exists x(\mathbf{EX}\, \psi(x)). \\
\text{(v)} & \mathbf{E}[\psi_1\, \mathbf{U}(\exists x\, \psi_2(x))] & \equiv & \exists x \mathbf{E}[\psi_1\, \mathbf{U}\, \psi_2(x)]. \\
\text{(vi)} & \mathbf{A}[(\forall x\, \psi_1(x))\, \mathbf{U}\, \psi_2] & \equiv & \forall x \mathbf{A}[\psi_1(x)\, \mathbf{U}\, \psi_2].
\end{array}
$$

Proof. These equivalences can be proved in a straightforward manner by using the semantics definitions of Figure 3.3. The notation $\psi([x \backslash t])$ will represent the formula ψ in which all occurrences of x have been syntactically replaced by t.

1.	$M, s \models \psi$	$\Leftrightarrow$	There is a $\mathcal{B}$ such that $M, s \models_{\mathcal{B}} \psi$.
2.	$M, s \models_{\mathcal{B}} p(t_1, \ldots, t_n)$	$\Leftrightarrow$	$p(\mathcal{B}(t_1), \ldots, \mathcal{B}(t_n)) \in L(s)$.
3.	$M, s \models_{\mathcal{B}} \neg\psi$	$\Leftrightarrow$	$M, s \models_{\mathcal{B}} \psi$ does not hold.
4.	$M, s \models_{\mathcal{B}} \psi_1 \vee \psi_2$	$\Leftrightarrow$	$M, s \models_{\mathcal{B}} \psi_1$ or $M, s \models_{\mathcal{B}} \psi_2$.
5.	$M, s \models_{\mathcal{B}} \psi_1 \wedge \psi_2$	$\Leftrightarrow$	$M, s \models_{\mathcal{B}} \psi_1$ and $M, s \models_{\mathcal{B}} \psi_2$.
6.	$M, s \models_{\mathcal{B}} \forall x\, \psi$	$\Leftrightarrow$	For all $c \in \mathcal{U}$, $M, s \models_{\mathcal{B}[x \mapsto c]} \psi$.
7.	$M, s \models_{\mathcal{B}} \exists x\, \psi$	$\Leftrightarrow$	For some $c \in \mathcal{U}$, $M, s \models_{\mathcal{B}[x \mapsto c]} \psi$.
8.	$M, s \models_{\mathcal{B}} \mathbf{EF}\, \psi$	$\Leftrightarrow$	There is a path $\pi \in \Pi_s$ containing a state $s_i \in \pi$ such that $M, s_i \models_{\mathcal{B}} \psi$.
9.	$M, s \models_{\mathcal{B}} \mathbf{EG}\, \psi$	$\Leftrightarrow$	There is a path $\pi \in \Pi_s$ such that $M, s_i \models_{\mathcal{B}} \psi$ for all states $s_i \in \pi$.
10.	$M, s \models_{\mathcal{B}} \mathbf{EX}\, \psi$	$\Leftrightarrow$	There is a state s_1 such that of $(s, s_1) \in R$ and $M, s_1 \models_{\mathcal{B}} \psi$.
11.	$M, s \models_{\mathcal{B}} \mathbf{E}\, [\psi_1 \mathbf{U} \psi_2]$	$\Leftrightarrow$	For a path $\pi = (s_0, s_1, \ldots)$ where $s = s_0$ there is a $k \geq 0$ such that $M, s_i \models_{\mathcal{B}} \psi_1$ for all $i < k$ and $M, s_j \models_{\mathcal{B}} \psi_2$ for all $j \geq k$.
12.	$M, s \models_{\mathcal{B}} \mathbf{AF}\, \psi$	$\Leftrightarrow$	Every path π from s contains a state $s_i \in \pi$ such that $M, s_i \models_{\mathcal{B}} \psi$.
13.	$M, s \models_{\mathcal{B}} \mathbf{AG}\, \psi$	$\Leftrightarrow$	On every path π from s, there holds $M, s_i \models_{\mathcal{B}} \psi$ in all states $s_i \in \pi$.
14.	$M, s \models_{\mathcal{B}} \mathbf{AX}\, \psi$	$\Leftrightarrow$	For all s_1 such that $(s, s_1) \in R$, holds $M, s_1 \models_{\mathcal{B}} \psi$.
15.	$M, s \models_{\mathcal{B}} \mathbf{A}\, [\psi_1\, \mathbf{U}\, \psi_2]$	$\Leftrightarrow$	For all paths $\pi = (s_0, s_1, \ldots)$ where $s = s_0$ there is a $k \geq 0$ such that $M, s_i \models_{\mathcal{B}} \psi_1$ for all $i < k$ and $M, s_j \models_{\mathcal{B}} \psi_2$ for all $j \geq k$.

Figure 3.3: Semantics of CTPL.

(i) Besides the semantics definition we only need the knowledge that quantifiers of the same kind commute, i.e. that $\forall x \forall y\, \psi \equiv \forall y \forall x\, \psi$:

$$\begin{aligned} M, s \models \mathbf{AG}(\forall x\, \psi(x)) &\Leftrightarrow \forall \pi \in \Pi_s, \forall i \in \mathbb{N},\ M, \pi^i \models \forall x\, \psi(x) \\ &\Leftrightarrow \forall \pi \in \Pi_s, \forall i \in \mathbb{N}, \forall t \in \mathcal{U},\ M, \pi^i \models \psi([x \backslash t]) \\ &\Leftrightarrow \forall t \in \mathcal{U}, \forall \pi \in \Pi_s, \forall i \in \mathbb{N},\ M, \pi^i \models \psi([x \backslash t]) \\ &\Leftrightarrow \forall t \in \mathcal{U},\ M, s \models \mathbf{AG}(\psi([x \backslash t])) \\ &\Leftrightarrow M, s \models \forall x(\mathbf{AG}\, \psi(x)). \end{aligned}$$

(ii) Proving this equivalence is largely analogous to the above approach:

$$\begin{aligned} M, s \models \mathbf{AX}(\forall x\, \psi(x)) &\Leftrightarrow \forall \pi \in \Pi_s,\ M, \pi^1 \models \forall x\, \psi(x) \\ &\Leftrightarrow \forall \pi \in \Pi_s, \forall t \in \mathcal{U},\ M, \pi^1 \models \psi([x \backslash t]) \\ &\Leftrightarrow \forall t \in \mathcal{U}, \forall \pi \in \Pi_s,\ M, \pi^1 \models \psi([x \backslash t]) \\ &\Leftrightarrow \forall t \in \mathcal{U},\ M, s \models \mathbf{AX}(\psi([x \backslash t])) \\ &\Leftrightarrow M, s \models \forall x(\mathbf{AX}\, \psi(x)). \end{aligned}$$

(iii) Since $\mathbf{EF}\, \psi \equiv \neg \mathbf{AG}\, \neg \psi$ and $\exists x\, \psi(x) \equiv \neg \forall x\, \neg \psi(x)$, we can prove this equivalence by using (i):

$$\begin{aligned} \mathbf{EF}(\exists x\, \psi(x)) &\equiv \neg \mathbf{AG}(\neg(\neg \forall x\, \neg \psi(x))) \\ &\equiv \neg \mathbf{AG}(\forall x(\neg \psi(x))) \\ &\overset{\text{(i)}}{\equiv} \neg \forall x\, \mathbf{AG}(\neg \psi(x)) \\ &\equiv \exists x\, \neg \mathbf{AG}(\neg \psi(x)) \\ &\equiv \exists x\, \mathbf{EF}\, \psi(x) \end{aligned}$$

(iv) For this proof, we use $\mathbf{EX}\, \psi \equiv \neg \mathbf{AX}\, \neg \psi$ and (ii):

$$\begin{aligned} \mathbf{EX}(\exists x\, \psi(x)) &\equiv \neg \mathbf{AX}(\neg(\neg \forall x\, \neg \psi(x))) \\ &\equiv \neg \mathbf{AX}(\forall x(\neg \psi(x))) \\ &\overset{\text{(ii)}}{\equiv} \neg \forall x\, \mathbf{AX}(\neg \psi(x)) \\ &\equiv \exists x\, \neg \mathbf{AX}(\neg \psi(x)) \\ &\equiv \exists x\, \mathbf{EX} \psi(x) \end{aligned}$$

(v) This equivalence can again be proved using the semantics definition only:

$$
\begin{aligned}
&M, s \models \mathbf{E}[\psi_1 \,\mathbf{U}\, (\exists x\, \psi_2(x))] \Leftrightarrow \\
&\quad \Leftrightarrow \exists \pi \in \Pi_s, \exists i \in \mathbb{N}(M, \pi^i \models \exists x\, \psi_2(x) \wedge \forall j < i,\ M, \pi^j \models \psi_1) \\
&\quad \Leftrightarrow \exists \pi \in \Pi_s, \exists i \in \mathbb{N}(\exists t \in \mathcal{U}(M, \pi^i \models\ \psi_2([x \backslash t])) \wedge \forall j < i,\ M, \pi^j \models \psi_1) \\
&\quad \Leftrightarrow \exists \pi \in \Pi_s, \exists i \in \mathbb{N}, \exists t \in \mathcal{U}(M, \pi^i \models\ \psi_2([x \backslash t]) \wedge \forall j < i,\ M, \pi^j \models \psi_1) \\
&\quad \Leftrightarrow \exists t \in \mathcal{U}, \exists \pi \in \Pi_s, \exists i \in \mathbb{N}(M, \pi^i \models\ \psi_2([x \backslash t]) \wedge \forall j < i,\ M, \pi^j \models \psi_1) \\
&\quad \Leftrightarrow \exists t \in \mathcal{U},\ M, s \models \mathbf{E}[\psi_1 \,\mathbf{U}\, (\exists x\, \psi_2([x \backslash t]))] \\
&\quad \Leftrightarrow M, s \models \exists x\, \mathbf{E}[\psi_1 \,\mathbf{U}\, \psi_2(x)]
\end{aligned}
$$

(vi) Here we use (v) and the CTL equivalence

$$\mathbf{A}[\psi_1 \,\mathbf{U}\, \psi_2] \equiv \neg(\mathbf{E}[\neg\psi_2 \,\mathbf{U}\, (\neg\psi_2 \wedge \neg\psi_1)] \vee \mathbf{EG}\, \neg\psi_2)$$

$$
\begin{aligned}
\mathbf{A}[(\forall x\, \psi_1(x)) \,\mathbf{U}\, \psi_2] \quad &\equiv \quad \neg(\mathbf{E}[\neg\psi_2 \,\mathbf{U}\, (\neg\psi_2 \wedge \neg\forall x\, \psi_1(x))] \vee \mathbf{EG}\, \neg\psi_2) \\
&\equiv \quad \neg(\mathbf{E}[\neg\psi_2 \,\mathbf{U}\, (\neg\psi_2 \wedge \exists x\, \neg\psi_1(x))] \vee \mathbf{EG}\, \neg\psi_2) \\
&\equiv \quad \neg(\mathbf{E}[\neg\psi_2 \,\mathbf{U}\, \exists x\, (\neg\psi_2 \wedge \neg\psi_1(x))] \vee \mathbf{EG}\, \neg\psi_2) \\
&\overset{(v)}{\equiv} \quad \neg(\exists x\, \mathbf{E}[\neg\psi_2 \,\mathbf{U}\, (\neg\psi_2 \wedge \neg\psi_1(x))] \vee \mathbf{EG}\, \neg\psi_2) \\
&\equiv \quad \forall x\, \neg(\mathbf{E}[\neg\psi_2 \,\mathbf{U}\, (\neg\psi_2 \wedge \neg\psi_1(x))] \vee \mathbf{EG}\, \neg\psi_2) \\
&\equiv \quad \forall x\, \mathbf{A}[\psi_1(x)) \,\mathbf{U}\, \psi_2]
\end{aligned}
$$

□

3.2.6 Modeling Program Behavior

CTPL has been designed to allow for much flexibility in specifying program behavior. Using the translation from an assembler program to a Kripke structure described in Section 3.1.3, we are able to specify behavior of an executable such as '*An execution path exists such that at some point some register is set to zero, and in the next instruction pushed onto the stack*' by the succinct CTPL formula

$$\exists r\, \mathbf{EF}(\mathtt{mov}(r, \mathtt{0}) \wedge \mathbf{EX}\, \mathtt{push}(r)).$$

Here, r is a variable, existentially quantified by $\exists$, and `0` is a constant, while `mov` and `push` are both predicates. By replacing **EX** with **EF**, we can specify a code sequence where other instructions are allowed to occur between mov and push, thus transforming the sentence into '*An execution path exists such that at some point some register is set to zero, and from there a path exists such that this register is finally pushed onto the stack*', written in CTPL as:

$$\exists r\, \mathbf{EF}(\mathtt{mov}(r, \mathtt{0}) \wedge \mathbf{EF}\, \mathtt{push}(r)).$$

Note that this specification does not prevent the presence of instructions between mov and push that modify the contents of the register r. To ensure that the value of 0 is still present in r it is necessary, however, to disallow any change to the register. Of course there are many instructions that affect the contents of a register, but for simplicity reasons, we assume for a moment that a register is only changed by a mov instruction. We need to find a path on which the register r is not changed between the first and the second instruction, so we will want to use the construction $\mathbf{E}\neg\exists t\, \mathtt{mov}(r, t))\, \mathbf{U} \ldots$ in the CTPL formula. The register needs to be protected starting with the instruction immediately following the assignment of 0 to r, which is specified by prefixing the **EU** construction with **EX**:

$$\exists r\, \mathbf{EF}(\mathtt{mov}(r, \mathtt{0}) \wedge \mathbf{EX}\, \mathbf{E}(\neg\exists t\, \mathtt{mov}(r, t))\, \mathbf{U}\, \mathtt{push}(r)).$$

In specifications it is generally desirable to ensure the integrity of values between two specified instructions that are not necessarily consecutive. We can always generate a specification similar to the above that prohibits certain instructions from occurring.

Of particular interest for specifying malicious behavior in executables are calls to the system API, as they are necessary to perform any I/O operation, be it network or file access. On the assembler level, a system call to the Windows API will be represented as a call instruction. Immediately before this call, one or more push instructions will be present, pushing the parameters of the system call onto the stack. The stack layout for the parameters of a system call at the moment the call is executed can be seen in Figure 3.4; for illustration purposes, we use the WinAPI call GetModuleFileName. The push instructions either have constant parameters, or they are preceded by other instructions that compute the parameter value dynamically. CTPL can be used to specify the behavior of such code fragments independently of the actual instruction scheduling produced by the compiler. Such a specification enforces only the correct computation of the parameter values and the

C Declaration:

```
DWORD GetModuleFileName(
   HMODULE hModule,
   LPTSTR lpFilename,
   DWORD nSize
);
```

Assembler Code:

```
      xor ebx, ebx
      push 80h
      lea eax, lpFileName
      push eax
      push ebx
  →   call GetModuleFileName
```

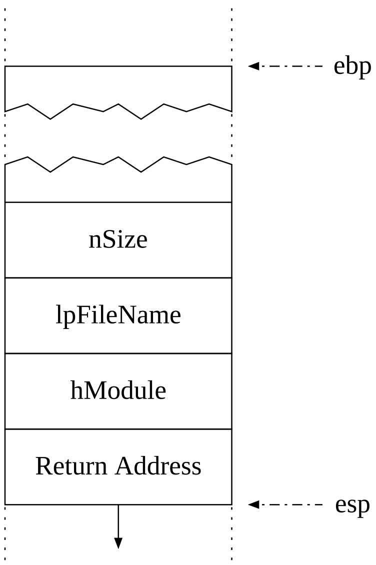


Figure 3.4: Local stack frame at the moment of a system call.

correct stack layout at the time of the function call. This can be achieved in CTPL by the conjunction of several different subformulas. One subformula represents the order in which the function parameters are pushed onto the stack, finishing with the system call itself, while the other subformulas specify the computation of the individual parameter values. In order to correctly anchor these subformulas in each other, we use the special location predicate $\#\mathrm{loc}(L)$ mentioned in Section 3.1.3, which is true only in the L-th node of the Kripke structure.

Using this predicate, a specification for a correct call to a function `fn` that takes two parameters, where the second parameter is set to zero, can be written as:

$$
\begin{aligned}
\exists L \exists r_1 (\quad & \mathbf{EF}(\texttt{mov}(r_1, 0) \wedge \mathbf{EF}\#\mathrm{loc}(L)) \wedge \\
& \exists r_2 \mathbf{EF}(\texttt{push}(r_2) \wedge \mathbf{EF}(\texttt{push}(r_1) \wedge \#\mathrm{loc}(L) \wedge \mathbf{EF}(\texttt{call}(\texttt{fn})))) \\
)
\end{aligned}
$$

The first line of the formula expresses that there exists a `mov` instruction in the code that clears some register r_1, before there will be some state after this instruction, which corresponds to the location L specified in the next

`mov`	`edi, [ebp+arg_0]`	
`xor`	`ebx, ebx`	Clear register `ebx` for later use.
`push`	`edi`	
	⋮	
`lea`	`eax, [ebp+ExFileName]`	Load the address of the string buffer.
`push`	`104h`	Push string buffer size.
`push`	`eax`	Push the string buffer address.
`push`	`ebx`	Set the first argument to NULL.
`call`	`ds:GetModuleFileName`	Call `GetModuleFileName`.
`lea`	`eax, [ebp+FileName]`	Load the address of the destination filename.
`push`	`ebx`	Set the third argument to zero.
`push`	`eax`	Push the address of the destination filename.
`lea`	`eax, [ebp+ExFileName]`	Fetch source filename address.
`push`	`eax`	Push the address as first argument.
`call`	`ds:CopyFileA`	Call `CopyFile`.

Figure 3.5: Code fragment of the infection routine of *Klez.h*.

line. The second line specifies a sequence of `push` instructions that precede a `call` to function `fn`. In particular, it asserts that

- eventually a register r_2 will be pushed onto the stack,
- at some later point register r_1 (which is the same register as in the first line of the formula) is also pushed onto the stack,
- and finally a call to function `fn` is executed.

For simplicity, the subformula that would ensure integrity of r_1 between the `mov` and its corresponding `push` instruction has been omitted, as well as the subformulas that would assert that the stack is not altered between the `push` instructions and the final `call`. Note that the formula does not specify a temporal relationship of line one and two based on operators. The only temporal link between the two subformulas is the location L of the instruction that pushes register r_1. We will refer to such a location as *anchor* in the future.

Figure 3.5 shows a part of the disassembled infection routine of the worm *Klez.h* that exhibits a behavior typical to e-mail worms. The code uses the Windows API call `GetModuleFileName` to determine the filename of the executable it was loaded from, and afterwards uses a second system call `CopyFile` to copy this file to another location, which is usually a Windows system directory or a shared folder. The Windows API function `GetModuleFileName` can be used to retrieve the filename of the executable belonging to a specific process module and takes three parameters:

1. `hModule`: A numerical handle to the process module whose name is requested.

2. `lpFilename`: A pointer to a string buffer designated to hold the returned filename.

3. `nSize`: The size of the string buffer.

If `hModule` is zero (NULL), the filename of the calling process is returned, which is the case in this example and enforced in the corresponding specification created later on. The system call `CopyFile` takes three parameters:

1. `lpExistingFilename`: A pointer to a string holding the name of the source file.

2. `lpNewFilename`: A pointer to a string containing the target filename.

3. `bFailIfExists`: A Boolean flag that determines whether the target file should be overwritten if it already exists.

The fragment of assembler code in Figure 3.5 displays the invocation of those two system calls, along with the necessary parameter initializations (relevant lines are explained in the figure).

This typical proliferation behavior of a worm is specified in the CTPL formula in Figure 3.6. The structure of the formula is consistent with the process of specifying system calls described above. It matches code that calls `GetModuleFileName` with a zero handle to retrieve its own filename, and later uses the result as a parameter to the system call `CopyFile`. It is divided into three main subformulas that are put into conjunction with each other. Temporal dependencies between different clauses are expressed by the use of location predicates. The first line of the formula quantifies the variables common to all three of the main subformulas:

$$
\begin{array}{l}
\exists L_m \exists L_c \exists v_{File}(\\
\quad \exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0(\\
\quad\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \\
\quad\quad\quad \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_0, t) \vee \mathtt{lea}(r_0, t)))\mathbf{U}\#\mathrm{loc}(L_0)) \\
\quad\quad \wedge \mathbf{EF}(\mathtt{mov}(r_1, \mathtt{0}) \\
\quad\quad\quad \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_1, t) \vee \mathtt{lea}(r_1, t)))\mathbf{U}\#\mathrm{loc}(L_1)) \\
\quad\quad \wedge \mathbf{EF}(\mathtt{push}(c_0) \\
\quad\quad\quad \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
\quad\quad\quad\quad \mathbf{U}(\mathtt{push}(r_0) \wedge \#\mathrm{loc}(L_0) \\
\quad\quad\quad\quad\quad \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
\quad\quad\quad\quad\quad\quad \mathbf{U}(\mathtt{push}(r_1) \wedge \#\mathrm{loc}(L_1) \\
\quad\quad\quad\quad\quad\quad\quad \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathbf{U}(\mathtt{call}(\mathtt{GetModuleFileName}) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge \#\mathrm{loc}(L_m)) \\
\quad\quad\quad\quad\quad\quad) \\
\quad\quad\quad\quad) \\
\quad\quad) \\
\quad) \\
\quad \wedge \exists r_0 \exists L_0(\\
\quad\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \\
\quad\quad\quad \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_0, t) \vee \mathtt{lea}(r_0, t)))\mathbf{U}\#\mathrm{loc}(L_0)) \\
\quad\quad \wedge \mathbf{EF}(\mathtt{push}(r_0) \wedge \#\mathrm{loc}(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
\quad\quad\quad \mathbf{U}(\mathtt{call}(\mathtt{CopyFileA}) \wedge \#\mathrm{loc}(L_c))) \\
\quad) \\
\quad \wedge \mathbf{EF}(\#\mathrm{loc}(L_m) \wedge \mathbf{EF}\#\mathrm{loc}(L_c)) \\
)
\end{array}
$$

Figure 3.6: CTPL formula for code creating copies of its own executable.

- L_m, the location of the `GetModuleFileName` call,
- L_c, the location of the `CopyFile` call, and
- v_{File}, the string buffer which holds the filename of the process.

The first and largest subformula, from lines 2 to 18 represents the call to `GetModuleFileName`. It is itself split into three conjugated subformulas, which share the following local variables:

- r_0, in which the address of the string buffer for the executable filename is loaded,
- r_1, the register holding the NULL value passed as parameter `hModule`,
- L_0, the location in which the pointer to the string buffer r_0 is pushed,
- L_1, the location where r_1 is pushed, and
- c_0, the size of the string buffer (the correct initialization of this constant does not need to be checked and is assumed).

Line 3 starts the subformula that specifies that the string buffer pointer is stored in r_0. There has to be a path afterwards on which this register is not altered by `mov` or `lea` instructions until the location L_0 is reached, which is assured using the data integrity construction described earlier. Line 5 asserts that register r_1 is set to zero and again enforces that this register remains unchanged until it is pushed onto the stack at location L_1. Lines 7 to 12 specify the order in which the individual parameters are written to the stack, and enforce that no stack operations are performed in-between that would break the correct parameter layout. The `push` instructions are conjugated with location predicates to mark the locations L_1 and L_2, which anchor the former two subformulas regarding parameter initialization. This subformula is concluded by the actual system call to `GetModuleFileName` in line 13, bound to location L_m.

The second system call to `CopyFile` is specified in lines 19 to 24 in a similar way. This time, only two local variables are used:

- r_0, some register r_0 (not necessarily the same register as in the first subformula!) that is assigned the address of the string buffer containing the executable filename, and

- L_0, the location in which this pointer is pushed onto the stack.

The structure is completely analogous to the first system call; the first two lines assert loading of the buffer address to r_0, and the third and fourth line specify r_0 as first parameter to `CopyFile`. This time, other parameters are not given, because we are not interested in the values of neither destination filename nor overwrite flag. The last line in the specification, line 25, enforces the correct ordering of the two system calls, using their locations. `GetModuleFileName` has to be called before `CopyFile`, i.e. the location L_m must occur before L_c.

Using constructions similar to the one described for this whole CTPL formula, it is possible to specify a wide range of malicious behavior. Another example for a CTPL formula describing malicious behavior can be found in Section 4.3.

3.3 Model Checking CTPL

3.3.1 Outline of the Algorithm

The efficiency of the classical explicit model checking algorithm for CTL [CE81] is based on the fact that it uses a form of dynamic programming. CTPL is derived from CTL—accordingly, the algorithm to check whether a Kripke structure M is a model of a CTPL formula φ is an extension to the model checking algorithm for CTL. The algorithm also iterates over the set of subformulas of φ, and visits the states of the Kripke structure as often as the classical algorithm. But as CTPL allows quantified variables, the algorithm has to keep track of those variable bindings that make the individual subformulas evaluate to true. The number of bindings might become exponentially large in the worst case, increasing the complexity of the algorithm. The complexity of CTPL model checking will be analyzed in detail in Section 3.4, where it will be shown that the model checking problem for CTPL is **PSPACE**-complete. However, **PSPACE**-completeness tells little about the practical performance on real world formulas and models. Experiments show that in particular the exponential growth of binding sets is not a performance bottleneck.

An outline of the model checking algorithm is given in Figure 3.7. It takes a formula φ and a Kripke Structure M as input. We assume that φ is passed

Algorithm ModelCheckCTPL:
Input: a Kripke structure M and a CTPL formula φ (as parse tree)
Output: set of states in M which satisfy φ

for all subformulas φ' of formula φ in ascending order of size
  case φ' **of**
    $\bot$: label no states;
    $p(t_1, \ldots, t_n)$: $\texttt{LABEL}_{\texttt{p}}(\varphi')$;
    $\exists x\ (\psi)$: $\texttt{LABEL}_{\exists}(\varphi')$;
    $\neg\psi$: $\texttt{LABEL}_{\neg}(\varphi')$;
    $\psi_1 \wedge \psi_2$: $\texttt{LABEL}_{\wedge}(\varphi')$;
    $\psi_1 \vee \psi_2$: $\texttt{LABEL}_{\vee}(\varphi')$;
    $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$: $\texttt{LABEL}_{\texttt{EU}}(\varphi')$;
    $\mathbf{EX}\psi$: $\texttt{LABEL}_{\texttt{EX}}(\varphi')$;
    $\mathbf{AF}\ \psi$: $\texttt{LABEL}_{\texttt{AF}}(\varphi')$;
output all $s \in S$ where $\exists C\ (s, \varphi, C) \in L$;

Figure 3.7: The CTPL Model Checking Algorithm.

in the form of a parse tree, so we can traverse it in a bottom-up fashion from the smallest subformulas, which are the predicates used, up to the complete formula φ. In every step of the iteration over the subformulas φ' of φ, all states in which φ' holds are labeled with φ'. Because the subformulas are ordered by size, all subformulas smaller than the current φ' are guaranteed to have been processed in previous steps and may be used in this step. $M \models \varphi$ holds if the initial state s_0 of M is finally labeled with φ.

The algorithm needs to process only predicates and the CTPL operators $\bot$, $\neg$, $\wedge$, $\exists$, **EX**, **EU**, and **AF**, as every formula containing other CTPL connectives can be transformed to use only operators of this subset (see Section 3.1.2). For efficiency reasons, both $\wedge$ and $\vee$ are handled directly even though one implementation would suffice for completeness.

3.3.2 Variable Bindings

The reason model checking of CTPL formulas is different from CTL model checking is the fact that the truth value of an expression depends on the assignment of variables. Bindings are generated at the moment the algorithm matches a predicate in the formula against a predicate in a label of the Kripke structure; in terms of first-order logic, bindings are the most general unifier for those two predicates. As the algorithm traverses the CTPL formula in a bottom-up manner, it has to remember these bindings in order to deduct the truth of a larger formula out of the truth values of its subformulas. If a subformula holds only with respect to certain bindings, these bindings have to be propagated up to every larger formula that includes it. If two subformulas ψ_1 and ψ_2 are combined with a binary operator, the bindings have to be consistent with each other: The variables that are bound to a certain value in ψ_1 must not be assigned otherwise in ψ_2. The bindings are collected and managed in the following ways:

- A single assignment of a constant to a variable is called *atomic assignment* a and is of the form $(v = c)$ or $(v \neq c)$, where $v \in V, c \in \mathcal{U}$. Here, V is the set of all variables. Because $\mathcal{U}$ is finite, $(v \neq c)$ is an abbreviation for
$$\bigvee_{u \in \mathcal{U}, u \neq c} (v = u).$$
We will denote the set of all atomic assignments by A.

- A *binding* $\mathcal{B}$ is a conjunction of one or more such atomic assignments:
$$\mathcal{B} = (a_1 \wedge \cdots \wedge a_n) \quad \text{with } a_1 \ldots a_n \in A,$$
or, in clause notation, $\mathcal{B} \subseteq A$. Note that this notation is consistent with the definition of the environment $\mathcal{B}$ in the semantics of CTPL (Figure 3.3). The empty binding that does not contain any assignments will be denoted by $\top$, as it formulates an empty condition that is always true.

- As several bindings may exist that satisfy a specific formula, bindings need to be collected in a set of legal bindings $C \subseteq 2^A$. This set of satisfying bindings may be seen as a disjunction of individual bindings:
$$C = (\mathcal{B}_1 \vee \ldots \vee \mathcal{B}_n).$$

Sets of bindings are disjunctions of conjunctions of atoms, which can be represented as Boolean formulas in disjunctive normal form (DNF). Thus, whenever sets of bindings have to be merged during the algorithm, this can be done by Boolean operations. The algorithm again transforms the resulting

Boolean formula in DNF, and simplifies atoms as necessary according to the following natural semantics:

$$\begin{aligned} \neg(v = c) &\equiv (v \neq c), \\ (v = c_1) \wedge (v = c_2) &\equiv \perp, \\ (v = c) \wedge (v \neq c) &\equiv \perp, \\ (v = c_1) \wedge (v \neq c_2) &\equiv (v = c_1), \end{aligned}$$

where $v \in V$, $c_1, c_2 \in \mathcal{U}$, and $c_1 \neq c_2$. We will call a binding containing $\perp$ contradictory; such bindings can be eliminated from its enclosing binding set as they are unsatisfiable. A binding set that does not contain any bindings is also unsatisfiable and can be simplified to $\perp$. Furthermore, a binding set containing the empty binding $\top$ makes its associated CTPL formula hold unconditionally and can be simplified to $\top$ itself.

To account for the possibility of multiple satisfying binding sets, the labeling L of a Kripke structure is extended to be a relation of the form $L \subseteq (S \times \Phi \times 2^{(2^A)})$, where S is the set of states, Φ is the set of all CTPL formulas, and $2^{(2^A)}$ is the power set of all possible bindings. In particular, a tuple (s, φ', C) is stored in L, if the subformula φ' holds in state s with respect to all variable bindings $\mathcal{B} \in C$. The extended labeling relation is initialized by assigning $\top$ as set of satisfying bindings C to all initial labels of the Kripke structure.

3.3.3 The Algorithm in Detail

This section will demonstrate the processing of the individual operators in the subroutines referenced in the outline of the algorithm in Figure 3.7.

Predicates. The procedure in Figure 3.8 handles the smallest CTPL formulas, the predicates. In particular, it adds the tuple $(s, p(t_1, \ldots, t_n), \{\mathcal{B}\})$ to the labeling relation L for every state s in which the predicate $p(t_1, \ldots, t_n)$ holds with respect to the variable binding $\mathcal{B}$. Note that the Kripke structure is already labeled with predicates over constants (instructions and locations, see Section 3.1.3) before the model checking algorithm is started. These initial labels are all associated to the binding set $\top$ to denote that they hold unconditionally. Predicates hold in states if they match these initial labels with respect to their associated variable bindings.

1 **procedure** $\mathtt{LABEL}_\mathtt{p}(\varphi')$ // $\varphi' = p(t_1, \ldots, t_n)$
2 stateIteration: **for all** $s \in S$
3 **if** $\exists c_1, \ldots, c_n$ with $(s, p(c_1, \ldots, c_n), \top) \in L$ **then**
4 $\mathcal{B} := \top$;
5 **for** $i := 1$ **to** n
6 **if** t_i is a variable **then** $\mathcal{B} := \mathcal{B} \wedge (t_i = c_i)$;
7 **else if** $t_i \neq c_i$ **then continue** stateIteration;
8 **if** $\mathcal{B} \not\equiv \bot$ **then** $L := L \cup (s, \varphi', \{\mathcal{B}\})$;

Figure 3.8: Subroutine $\mathtt{LABEL}_\mathtt{p}$, handling predicates.

The subroutine iterates over all states: It checks for every state s whether it has been initially labeled with an instance $p(c_1, \ldots, c_n)$ of the predicate p (line 3). Whenever a predicate of the same kind has been found, the procedure calculates the most general unifier over the parameters: It instantiates an empty binding $\mathcal{B}$ and iterates over all parameters. During this iteration, it adds the atomic assignment $(t_i = c_i)$ to $\mathcal{B}$ for every parameter t_i of the predicate to be checked that is a variable (line 6). For each t_i that is a constant, it requires that the exact same constant is also present at the same place in the existing label, i.e. that $t_i = c_i$. If this is not the case, the state is not labeled and the iteration is resumed (line 7).

After all parameters have been processed, this instance of the predicate is associated with the generated binding $\mathcal{B}$ and added to the labeling relation, as long as $\mathcal{B}$ does not contain a contradiction. A contradiction would cause $\mathcal{B}$ to equal $\bot$, and could emerge when the same variable had to be bound to different values to unify the two instances of p.

Existential Quantifiers. Existential quantifiers are processed in the subroutine depicted in Figure 3.9. Because subformulas are iterated from smallest to largest, the algorithm can safely assume that all states where ψ holds are already labeled with ψ. The subroutine searches for those states and copies the labels of ψ to the new label φ', except for those atomic assignments that contain x.

An existential quantifier limits the scope of the quantified variable to the enclosed subformula. In the algorithm, assignments of this variable have to be removed from the set of bindings at the time the subformula headlined

1 **procedure** `LABEL`$_\exists(\varphi')$ $\quad // \varphi' = \exists x\,(\psi)$
2 $\quad$ **for all** $s \in S$
3 $\quad\quad$ **if** $\exists C$ with $(s, \psi, C) \in L$ **then**
4 $\quad\quad\quad$ $C' := \bot$;
5 $\quad\quad\quad$ **for all** $\mathcal{B} \in C$
6 $\quad\quad\quad\quad$ $\mathcal{B}_0 := \top$;
7 $\quad\quad\quad\quad$ **for all** $a \in \mathcal{B}$
8 $\quad\quad\quad\quad\quad$ **if** $x \notin a$ **then** $\mathcal{B}_0 := \mathcal{B}_0 \wedge a$;
9 $\quad\quad\quad\quad$ $C' := C' \vee \mathcal{B}_0$;
10 $\quad\quad\quad$ $L := L \cup (s, \varphi', C')$;

Figure 3.9: Subroutine `LABEL`$_\exists$, handling existential quantifiers.

by the quantifier is processed. To justify this behavior, consider the formula $\mathbf{AF}(\exists x\, p(x))$; it specifies that on every path there will be a state in which some x exists such that $p(x)$ holds. This is fundamentally different from $\exists x\, \mathbf{AF}\, p(x)$, which would express that there is one specific x, such that on all paths there will be states in which $p(x)$ holds.

As the algorithm works from bottom up, it would start for both formulas by labeling all states containing a predicate p with $p(x)$ and associating the label to a binding that assigned the variable x to some constant value. In the latter case, the next subformula to be checked would be $\mathbf{AF}\, p(x)$. Bindings in which x had been assigned different values would cause contradictions, which would prevent states from being labeled with this subformula. In the former case, however, $\exists x\, p(x)$ would be evaluated first. This would cause all assignments of x to any constants to be purged, thus making all $p(x)$ labels compatible.

The subroutine iterates over all states and processes those labeled with ψ. It uses two loops to enumerate all atomic assignments a (line 7) in all bindings (line 5), and creates a new set of bindings by copying all atomic assignments except those containing x (line 8). Finally, the new set of bindings is associated with the subformula φ' and added to the labeling (line 10).

Negations. Figure 3.10 shows the subroutine for processing subformulas headed by a negation. It iterates all states: If ψ does not hold in a state s, i.e. s is not labeled with the enclosed subformula ψ, then $\neg\psi$ holds in s regardless of variable bindings, and a label $(s, \neg\psi, \top)$ is added (line 5). If s

```
1 procedure LABEL¬(φ′)    // φ′ = ¬ψ
2   for all s ∈ S
3     if ∃C with (s, ψ, C) ∈ L then
4       if ¬C ≢ ⊥ then L := L ∪ (s, φ′, ¬C);
5     else L := L ∪ (s, φ′, ⊤);
```

Figure 3.10: Subroutine LABEL$_\neg$, handling negations.

```
1 procedure LABEL∧(φ′)    // φ′ = ψ1 ∧ ψ2
2   for all s ∈ S
3     if ∃C1 with (s, ψ1, C1) ∈ L and ∃C2 with (s, ψ2, C2) ∈ L then
4       if C1 ∧ C2 ≢ ⊥ then L := L ∪ (s, φ′, C1 ∧ C2);
```

Figure 3.11: Subroutine LABEL$_\wedge$, handling conjunctions.

does have a label ψ, then $\neg\psi$ still holds in s under all variable bindings that cause ψ to evaluate to false in s.

Consequently, a new set of bindings is created by negating the Boolean formula representing the set of bindings for ψ. This negated formula is then again brought into DNF, which creates new bindings and eventually causes equality assignments in the binding to be changed to inequality and vice versa. For example, if the set of bindings C is

$$((x \neq c) \wedge (x \neq d) \wedge (y = e)) \vee ((y = c)),$$

the negated binding set $\neg C$ becomes

$$((x = c) \wedge (y \neq c)) \vee ((x = d) \wedge (y \neq c)) \vee ((y \neq e) \wedge (y \neq c)).$$

However, if C contains the empty binding, i.e. $C = \top$, the new set equals $\perp$ and is not added (line 4).

Conjunctions. The subroutine for conjunctions is depicted in Figure 3.11. It iterates all states, labeling those in which both subformulas ψ_1 and ψ_2 hold. The binding set C' for the new label φ', however, has to suit both subformulas in this state. This is achieved by creating C' as the Boolean conjunction of the two individual binding sets, $C_1 \wedge C_2$; afterwards, C' is transformed to DNF again. If C' is not contradictory, i.e. the resulting DNF is not simplified to $\perp$, then finally the new label is added (line 4).

1 **procedure** $\texttt{LABEL}_{\vee}(\varphi')$ // $\varphi' = \psi_1 \vee \psi_2$
2 **for all** $s \in S$
3 **if** $\exists C_1$ with $(s, \psi_1, C_1) \in L$ **then** $C' := C_1$ **else** $C' := \bot$;
4 **if** $\exists C_2$ with $(s, \psi_2, C_2) \in L$ **then** $C' := C' \vee C_2$;
5 **if** $C' \not\equiv \bot$ **then** $L := L \cup (s, \varphi', C')$;

Figure 3.12: Subroutine $\texttt{LABEL}_{\vee}$, handling disjunctions.

procedure $\texttt{LABEL}_{\texttt{EU}}(\varphi')$ // $\varphi' = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$
for all $s \in S$
if $\exists C_2$ with $(s, \psi_2, C_2) \in L$ **then** $L := L \cup (s, \varphi', C_2)$;
while L has changed **do**
for all states s **if** $\exists C_s$ with $(s, \varphi', C_s) \in L$ **then**
for all $(p, s) \in R$ *// for all parents of s*
if $\exists C_1$ with $(p, \psi_1, C_1) \in L$ **then**
$C' := C_s \wedge C_1$;
if $C' \not\equiv \bot$ **then**
if $\exists C_p$ with $(p, \varphi', C_p) \in L$ **then**
$L := L \cup (p, \varphi', C' \vee C_p)$;
else $L := L \cup (p, \varphi', C')$;

Figure 3.13: Subroutine $\texttt{LABEL}_{\texttt{EU}}$, handling **EU**.

Disjunctions. Disjunctions are processed by the pseudo code in Figure 3.12. If a state s is labeled with one of the subformulas ψ_1 and ψ_2, it copies the binding set of this subformula to the new label φ'. If s is labeled with both formulas, the new binding set is created as the union of the existing ones, as any of the variable bindings in both sets will cause $\psi_1 \vee \psi_2$ to evaluate to true in s.

In particular, if s is labeled with ψ_1, the algorithm copies C_1, the set of bindings of ψ_1 in s, to a new binding set C'. Else, C' is initialized with $\bot$ (line 3). Next, if ψ_2 holds in s with respect to the binding set C_2, these bindings are added to C', i.e. connected by Boolean 'or' (line 4). In case C' was $\bot$, the result of this operation is just C_2. Finally, if it is not still equal to $\bot$, the new binding is associated with φ' and added to the labeling relation.

```
procedure LABEL_EX(φ')    // φ' = EX ψ
  for all s ∈ S
      if ∃C_s with (s, ψ, C_s) ∈ L then for all (p, s) ∈ R
          if ∃C_p with (p, φ', C_p) ∈ L then L := L ∪ (p, φ', C_s ∨ C_p);
          else L := L ∪ (p, φ', C_s);
```

Figure 3.14: Subroutine $\texttt{LABEL}_{\texttt{EX}}$, handling **EX**.

EU. Figure 3.13 shows the labeling algorithm for a subformula φ' of the form $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$. The basic idea is that φ' trivially holds in every state in which ψ_2 holds. Beginning from there, a fixed point iteration labels every state with φ' in which ψ_1 holds and which has a predecessor labeled with φ'.

To implement this, the algorithm first labels all states with φ' that are labeled with ψ_2 (line 2). After this initial step, the algorithm enters the main **while** loop, a fixed point iteration which terminates as soon as the labeling does not change anymore (line 4).

Inside this loop, all states s already labeled with φ' are iterated (line 5). Because $\varphi' = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ holds in these states, φ' will also hold in every predecessor state in which ψ_1 holds. Consequently, every predecessor state p that is already labeled with ψ_1 is labeled with φ' (lines 7–11). In order to satisfy both subformulas, the binding set C' of the new label φ' in p has to take into account both the set C_1 associated with ψ_1 in the label of p, and the set C_s coming from the child state's label φ'.

To this end, C' is created as the conjunction of C_1 and C_s (line 8). If C' is not contradictory, the subroutine checks whether p has already been labeled with φ' because of another successor in another iteration step (line 10), and unions C' with the existing set of bindings in this case (line 11). If the label is new, φ' is added as a new label to p with respect to the binding C' (line 12).

EX. The section of the CTPL model checking algorithm responsible for validating **EX** expressions is shown in Figure 3.14. It iterates over the parents p of those states s labeled with ψ and labels them with $\varphi' = \mathbf{EX}\,\psi$, as ψ will hold in a successor state of p, namely s. A state can have multiple successors that are labeled with ψ, thus the union of all individual binding sets has to

1 **procedure** $\texttt{LABEL}_{\texttt{AF}}(\varphi')$ // $\varphi' = \mathbf{AF}\,\psi$
2 **for all** $s \in S$
3 **if** $\exists C$ with $(s, \psi, C) \in L$ **then** $L := L \cup (s, \varphi', C)$;
4 **while** L has changed **do**
5 stateIteration: **for all** states s
6 $C' := \top$;
7 **for all** $(s, c) \in R$ *// for all children of* s
8 **if** $\exists C_c$ with $(c, \varphi', C_c) \in L$ **then** $C' := C' \wedge C_c$;
9 **else continue** stateIteration;
10 **if** $C' \equiv \bot$ **then continue** stateIteration;
11 $L := L \cup (s, \varphi', C')$;

Figure 3.15: Subroutine $\texttt{LABEL}_{\texttt{AF}}$, handling **AF**.

be associated to the label φ' (line 4), because φ' will hold with respect to each set of bindings.

AF. The temporal operator **AF** is, analogously to **EU**, implemented by a fixed point algorithm, which is given in Figure 3.15. In line 3, all states with a label ψ are initially labeled with φ', as $\mathbf{AF}\,\psi$ is sure to also hold where ψ already does. Afterwards, the **while** loop labels—in each step—every state with φ' of which *all* successors are currently labeled with φ'.

The set of bindings generated for this new label has to take the binding sets of all successor states into account: In the pseudo code, this is accomplished by successively adding binding sets of child states of s labeled with φ' into a conjunction with a binding set that is initially $\top$ (line 8). If any child state is not labeled with φ' (line 9), or if the iteratively generated binding set is contradictory (line 10), the next s is processed. The loop continues until a fixed point is reached, i.e., until the labeling relation remains unchanged for one iteration.

3.4 Complexity of CTPL Model Checking

A model checker for plain CTL can be implemented as an efficient bottom up algorithm (see Section 3.1.2) and thus requires only polynomial time, while the more powerful CTL* logic is already **PSPACE**-complete [CES86].

As the logic CTPL, which was introduced in Section 3.2, allows succinct definitions of specifications that would take exponential space when written as a CTL formula, the model checking problem for CTPL can be expected to be of higher complexity. More precisely, in the following paragraphs it will be proved that model checking CTPL is, like CTL*, **PSPACE**-complete.

Lemma 3.14. *The model checking problem for CTPL is* **PSPACE**-*hard.*

Proof. For proving **PSPACE**-hardness, it is sufficient to find a polynomial reduction from a **PSPACE**-hard problem to CTPL model checking. A classic problem known to be **PSPACE**-complete is the satisfiability of quantified Boolean formulas (QBF) [SM73]. A QBF is a propositional formula in which literals may be quantified by $\exists$ or $\forall$, e.g.

$$\forall a \exists b \, (a \vee b) \wedge (\neg a \vee \neg b).$$

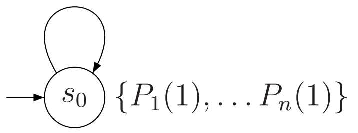
W.l.o.g. it suffices to consider closed formulas in prenex normal form, as free variables are implicitly existentially quantified. Let

$$q = Q_1 a_1 \ldots Q_n a_n \; k$$

be a closed QBF in prenex normal form with the quantifiers $Q_1 \ldots Q_n$, the variables $a_1 \ldots a_n$, and the propositional kernel k containing only the variables $a_1 \ldots a_n$. Furthermore, let φ be the CTPL formula

$$\varphi = Q_1 x_1 \ldots Q_n x_n \; k'$$

with the same quantifiers $Q_1 \ldots Q_n$ as in q, and the variables $x_1 \ldots x_n$. k' is obtained by replacing the variable a_i in k by a unique unary predicate $P_i(x_i)$ for every $1 \leq i \leq n$. Further, let M be the following Kripke structure:



We will now show that q is satisfiable if and only if φ is true in the Kripke structure M, i.e.

$$q \Leftrightarrow M, s_0 \models \varphi.$$

In QBF, the universe is predefined as consisting of *true* and *false*; accordingly, we choose the universe for M to be $\mathcal{U} = \{0, 1\}$. Because the initial state of

M is labeled with $P_1(1) \dots P_n(1)$, $M, s_0 \models P_i(1)$ holds (i.e. $P_i(1)$ evaluates to *true*) for every $1 \le i \le n$; correspondingly, also $M, s_0 \not\models P_i(0)$ holds (i.e. $P_i(0)$ evaluates to *false*) for every $1 \le i \le n$.

When omitting the temporal operators of CTPL, the evaluation rules in the semantics of CTPL and QBF are now equivalent, as rule 2 of the CTPL semantics definition,

$$M, s \models_{\mathcal{B}} p(t_1, \dots, t_n) \Leftrightarrow p(\mathcal{B}(t_1), \dots, \mathcal{B}(t_n)) \in L(s),$$

degenerates in the context of this specially crafted Kripke structure into

$$M, s \models_{\mathcal{B}} p(t) \Leftrightarrow \mathcal{B}(t) = 1,$$

which is equivalent to saying that a (QBF) variable evaluates to true when it is set to true in the environment. The remaining semantics for Boolean operators and quantifiers are identical, consequently we can now conclude that q is satisfiable if and only if $M, s_0 \models \varphi$.

The transformation for turning a QBF into a CTPL formula described above obviously uses only polynomial space, as the formula only grows by one predicate symbol per variable, which causes the required space to double in worst case. $\square$

Lemma 3.15. *The model checking problem for CTPL is in* **PSPACE**.

Proof. To prove membership in **PSPACE**, we have to find an algorithm that solves the model checking problem for CTPL using only a polynomial amount of space. Unfortunately, the algorithm given in Section 3.3.1, while being efficient in most practical cases, uses an exponential amount of space in the worst case, as the binding sets can become exponentially large on $\wedge$ and $\neg$ operators. This is due to the bottom-up approach of the algorithm that considers all satisfying combinations of bindings at once. However, it is possible to formulate a recursive top-down model checking algorithm for CTPL that tests possible variable bindings one by one and thus uses only polynomial space.

The algorithm given in Figure 3.16 passes the formula φ and the initial state of the Kripke structure M to the function `check` that recurses through the syntax tree of φ. In every recursion step, the topmost symbol of the current subtree determines which part of the switch-statement is executed. Because every formula can be rewritten using only polynomial space to an

Algorithm ModelCheckCTPLTopDown:
Input: A Kripke structure M and a closed CTPL formula φ
Output: Whether the starting state s_0 of M satisfies φ

```
output check(s₀, φ);

function check (state s, formula φ′) : Boolean
  case topOperator(φ′) of
     ⊥: return false;
     ψ1 ∧ ψ2: return check(s, ψ1) and check(s, ψ2);
     ¬ψ: return not check(s, ψ);
     EX ψ:
        for all (s, c) ∈ R
           if check(c, ψ) then return true;
        return false;
     E[ψ1 U ψ2]:
        mark(s, φ′);
        if check(s, ψ2) then return true;
        else if check(s, ψ1) then
           for all (s, c) ∈ R
              if (not isMarked(c, φ′)) and check(c, φ′) then
                 return true;
        return false;
     AF ψ:
        mark(s, φ′);
        if check(s, ψ) then return true;
        else for all (s, c) ∈ R
           if (not isMarked(c, φ′)) and (not check(c, φ′)) then
              return false;
        return true;
     ∃x ψ:
        for all t ∈ 𝒰
           if check(s, ψ[x\t]) then return true;
        return false;
     P(t):
        if (s, P(t)) ∈ L then return true;
        else return false;
```

Figure 3.16: Recursive model checking algorithm for CTPL.

equivalent one containing only predicates, $\perp$, $\neg$, $\wedge$, $\exists$, **EX**, **EU**, and **AF**, it is sufficient to handle only this subset of CTPL. We will prove by induction over the size of CTPL formulas that space cost is only polynomial in the input size (formula φ and model M).

As the function `check` is recursive, the size of the stack frame has to be taken into account: Every instance of the function occupies at least the space for its two parameters, which is $\mathcal{O}(|\varphi| + \log |S|)$ where S is the set of all states in M. The space for parameters can be reused for the return value, so it does not add to the space complexity. In particular, this allows to give the induction start, because checking of all atomic CTPL formulas, namely $\perp$ (line 5) and all predicates (line 31), does not use space other than its stack frame (for the Boolean return value).

The induction hypothesis is that the maximum space cost $\mathcal{C}(n)$ needed for checking any formula ψ of size $|\psi| \leq n$ is polynomial. What needs to be shown is that if an operator is added in front of a formula of size $|\psi| \leq n$, or if two formulas of size $\leq n$ are combined by an operator, the cost for checking the resulting formula is still polynomial. This is done separately for every operator:

- $\wedge$: The two checks are performed sequentially, so the required space is bounded by $\mathcal{O}(\mathcal{C}(n)+|\varphi|+\log |S|)$ which is polynomial by the induction hypothesis.

- $\neg$: Analogous to the above.

- **EX** ψ: All recursive calls are sequential, thus the space for every call can be reused. The relation R is already given in the model, so only a counter variable for the **forall** statement is needed, thus total space is

$$\mathcal{O}(\mathcal{C}(n) + |\varphi| + 2\log |S|) = \mathcal{O}(\mathcal{C}(n) + |\varphi| + \log |S|).$$

- $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$: This case is different in that it may call the `check` function with the same formula on child states. To avoid infinite recursion caused by loops in the model, every state is marked with the subformula it has been already checked with, so every state is checked at most once. Space used by sequential `check` calls inside a recursion instance (lines 14 and 15) can be reused, so it is needed only once. More influential are the maximum recursion depth of $|S|$ and the space needed per recursion instance for parameters and the **forall** iterator. Because marks of nested subformulas must not interfere with each other, they need to be unique for each subformula, causing an additional requirement of $|S| \cdot |\varphi|$. Thus the total space needed is

$$\mathcal{O}(\overbrace{|S| \cdot |\varphi|}^{\text{marks}} + \overbrace{|S| \cdot (|\varphi| + 2\log|S|)}^{\text{stack}} + \overbrace{\mathcal{C}(n)}^{\text{call}})$$

which can be simplified to

$$\mathcal{O}(|S| \cdot (|\varphi| + \log|S|) + \mathcal{C}(n)),$$

which is obviously polynomial.

- **AF** ψ: The proof for **AF** is analogous to the one for **EU**.
- $\exists x\ \psi$: The universe iterator will need $\mathcal{O}(\log|\mathcal{U}|)$ space, the individual `check` calls may reuse the same space, thus counting only once, and substituting the variables in their parameters does not take extra space. Consequently, the total is the polynomial

$$\mathcal{O}(\mathcal{C}(n) + |\varphi| + \log|S| + \log|\mathcal{U}|).$$

Regardless which operator is added to a CTPL formula ψ, the space needed for checking it with the algorithm in 3.16 is always polynomial in the size of model and formula. □

From Lemma 3.14 and Lemma 3.15 we get:

Theorem 3.16. *CTPL model checking is* **PSPACE**-*complete.*

Chapter 4

Implementation and Results

The CTPL model checking algorithm described earlier has been implemented as a Java application called *Mocca* ('**Mo**del **C**he**c**king **A**ssembler'), which will be presented in this chapter together with the results produced during experimental runs with different specifications on several malware specimens.

4.1 Toolchain

Mocca expects a plain text assembler file as input to construct the internal model representation, so there is some amount of preprocessing necessary when a new executable is to be checked. As a first step, it has to be determined whether the potentially malicious program is encoded with some kind of executable packer (see Section 2.2.1). There are tools available that are able to perform this task automatically and reliably, such as PEiD [JQsx]. Knowing which packer was used to protect the program, the corresponding unpacking tool can be chosen to correctly unpack the executable. The unpacking routine is usually completely separated from the underlying program, due to the fact that programs are packed by an external tool after their compilation. More complex obfuscation methods, which change the control flow of the code itself (as mentioned in Section 2.2.2), would typically have to be integrated into the program already during development.

After unpacking, the resulting plain binary can be passed to a disassembler. The prototype toolchain uses Datarescue's IDAPro for this task, which generates the plain text assembler file used as input to the Mocca model checker. Mocca creates the Kripke structure of the executable by parsing the

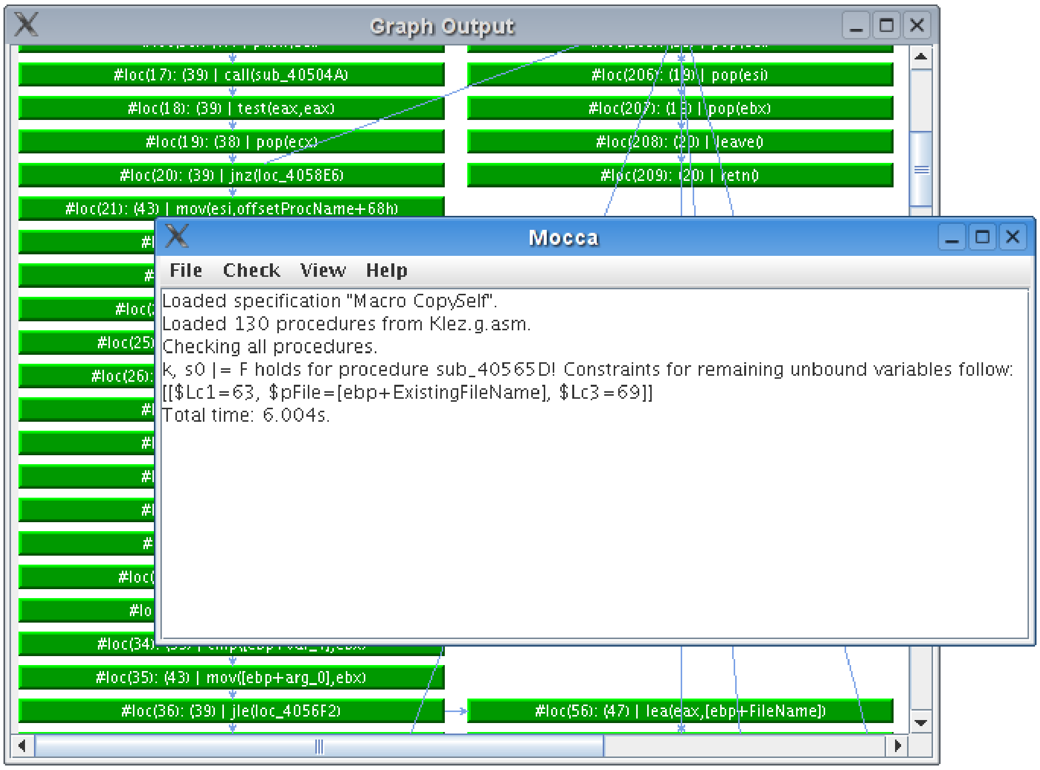


Figure 4.1: Screenshot of the Mocca GUI.

assembler file. During parsing, it is able to perform some simple syntactical substitutions to disambiguate the assembler code; currently, instructions of the form `xor` r, r are replaced by `mov` r, `0`. Finally, Mocca checks the Kripke structure against a malicious code specification file, which is a plain text file of the format described in Section 4.2.1.

Mocca has both a graphical (screenshot in Figure 4.1) and a command line interface. Both allow to load an assembler file and a specification, and check either all procedures or an individual one. In the graphical version, the Kripke structure of the matching procedure can also be inspected with all states labeled with matching subformulas.

The complete sequence of steps in the toolchain of the prototype can be seen in Figure 4.2. The steps are currently performed manually, but are rather mechanic in nature and thus can be automated towards an integrated analysis environment.

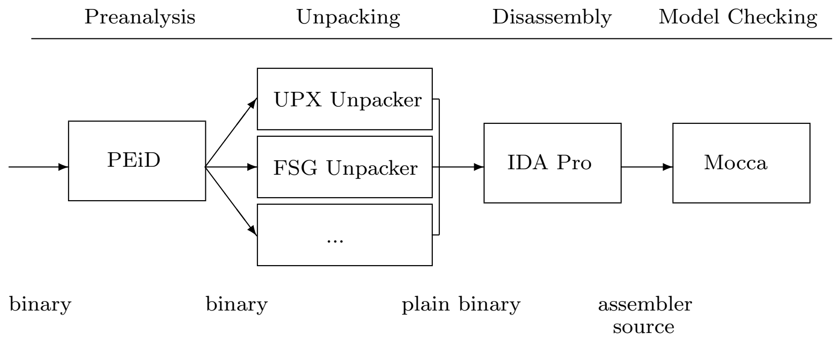


Figure 4.2: Toolchain for the prototype.

4.2 The Mocca Malware Detector

4.2.1 Specification File Format

Mocca reads specifications from text files, which consist of five sections. Every section is identified by a corresponding section header, a keyword enclosed in brackets; all lines of text read after the header are regarded as belonging to the section until the line where the next section header is encountered.

- `[version]`: A version tag in a single line identifying this specification file as belonging to a specific Mocca version.
- `[name]`: The name identifying the specification.
- `[description]`: Several lines of text describing the specification, for use in a graphical interface to give explanatory feedback to the user.
- `[clues]`: A list of the clues to look for in procedures, one per line. Clues are a simple concept to drastically decrease model checking times in larger assembler files. Malicious code specifications usually describe a specific connection between different system calls. If a procedure does not even contain these system calls, it can be safely skipped in the analysis. Every procedure is textually scanned for every clue in the list, and if one is missing, the procedure is excluded from model checking. In principle, a clue can be any kind of string that must occur in the disassembled procedure code. However, using system calls as clues has proved to be the most reasonable approach.

- `[formula]`: The actual malicious code specification as a CTPL formula in the format described in the next section.

All text in a line after a semicolon (';') is treated as a comment and is ignored; empty lines are skipped as well. An example input file, containing the specification used for detecting variants of NetSky, Klez and MyDoom, can be found in Figure 4.3.

4.2.2 Specification Language

The syntax for malicious code specification formulas used in Mocca is very similar to the CTPL syntax as defined in Section 3.2.3.

Operators. In its current state, Mocca supports the operators $\exists$, $\wedge$, $\vee$, $\neg$, **AF**, **EF**, **EX**, and **EU**, which are sufficient to express every possible CTPL statement, as all other operators can be transformed to combinations of operators from this set by using rewrite rules from Section 3.1.2. The following table reflects the Mocca syntax of the operators:

CTPL Operator	Mocca syntax
$\psi_1 \wedge \psi_2$	ψ_1 `&` ψ_2
$\psi_1 \vee \psi_2$	ψ_1 `|` ψ_2
$\neg\psi$	`-`ψ
$\mathbf{AF}\ \psi$	`AF` ψ
$\mathbf{EF}\ \psi$	`EF` ψ
$\mathbf{EX}\ \psi$	`EX` ψ
$\mathbf{E}[\psi_1\ \mathbf{U}\ \psi_2]$	`E` ψ_1 `U` ψ_2
$\exists x\ \psi$	`exists` $x\ \psi$

Predicates. For nearly all operators, the parameters are predicates; the only exception to this rule is the `exists` operator, which takes a variable as first parameter. Consequently, Mocca implicitly treats those symbols as predicates that ought to be predicates due to the parse tree.

Variables. Inside the parameter list of a predicate, symbols are either variables or constants. To avoid ambiguity, the specification language distinguishes between variables and constants. In Mocca specifications, every

```
                    CopySelfMacro.spec
                    ------------------

[version]
Mocca 0.1

[name]
CopySelf(Macro)

[description]
CopySelf specification using Macros.
An executable matching this specification retrieves
its own filename and uses this information to copy
itself to a different location.
Supports indirect and direct parameter initialization.

[clues]
call(CopyFileA)
call(GetModuleFileNameA)

[formula]
EF(%syscall(GetModuleFileNameA, $*, $pFile, 0) &
  E %noassign($pFile) U %syscall(CopyFileA, $pFile))

; The handle to the own file is stored in $pFile.
```

Figure 4.3: Input file of the ‘CopySelf’ specification.

variable is preceded by a dollar sign (`$`). Variables that are not explicitly quantified in the formula are assumed to be existentially quantified over the whole formula. For example, the CTPL formula

$$\exists x\, \mathbf{EF}(p(x) \wedge \mathbf{E}[\neg q(x)\, \mathbf{U}\, \exists y\, (q(y) \wedge \mathbf{AF} p(y)))])$$

can, by exploiting the implicit existential quantification of the free variable x, be written in Mocca syntax as

```
EF(p($x) & E -q($x) U exists $y (q($y) & AF p($y)))
```

Constants. Every symbol that occurs inside of a predicate that is not preceded by a dollar sign is treated as a constant. Predicates in the formula that contain constants as parameters can only hold in a state if these constants exactly match the values at the respective places in the predicate labeling the state.

Wildcards. A common occurrence in malicious code specifications are variables used as wildcards that match every parameter to a certain instruction. Their actual value is of no interest, as they are not needed in other parts of the formula.

Consider the following example: To specify that there is a path on which nothing is pushed onto the stack before a pop instruction is reached, we would usually write

$$\mathbf{E}(\neg\exists x\, \texttt{push}(x))\ \mathbf{U}\ (\exists y\, \texttt{pop}(y)).$$

To simplify specifications and improve their human readability by reducing the abundance of `exists` statements in the formula, the special wildcard `$*` is supported in Mocca, which corresponds to a variable that is locally existentially quantified around the enclosing predicate. Accordingly, every predicate $p(\ldots, \texttt{\$*}, \ldots)$ containing the wildcard `$*` is implicitly expanded to $\exists x\, p(\ldots, x, \ldots)$. The above formula can thus be written succinctly as

```
E (-push($*)) U pop($*)
```

It is noteworthy that using the wildcard is *not* equal to using a free singular variable, which would be implicitly quantified over the whole formula while it is this way only quantified over the local predicate.

Location. The special location predicate described in Section 3.1.3 is referenced by the notation `#loc(`*location*`)`. Mocca ensures uniqueness of labels inside a procedure by labeling every state in the extracted control flow graph with its line number relative to the procedure start. Variables may be used to refer to specific locations, as in `#loc($L1)`.

Macros. CTPL allows succinct specifications for assembler programs, but still examples such as the formula in Figure 3.6 show that these specifications can grow beyond easy legibility, and might be somewhat error prone due to the large number of nested expressions.

Mocca's specification language alleviates this problem by the introduction of several macros that encapsulate commonly used specification patterns. As an illustrative example, the formula depicted in Figure 3.6 can be written by the use of macros in the much more simple and natural form

```
EF(
  %syscall(GetModuleFileNameA, $*, $pFile, 0) &
    E %noassign($pFile) U %syscall(CopyFileA, $pFile)
)
```

Macros supported by the current version of the prototype are:

- `%nostack`. This macro expands to

  `(-push($*) & -pop($*))`,

  which is a statement to ensure that the stack is not altered in a state. This macro is most commonly used together with the **EU** operator to assure stack integrity, as in

  `E %nostack U call(DeleteFileA)`

- `%noassign(`*variable*`)`. This macro specifies that no value is assigned to the given variable in this state. In the prototype implementation, this expands to `(-mov(`*variable*`, $*) & -lea(`*variable*`, $*))`. On the x86 architecture, there are of course many more operations than `mov` and `lea` that assign a value to a register or memory operand. However, during practical testing this rather narrow definition proved to be sufficient.

This macro is also useful in the context of the **EU** operator in situations when it must be ensured that a register is not altered in the timespan between the state in which it is assigned a value and the state in which it is used in another instruction. For example, a formula that specifies that some register is assigned a constant value and pushed on the stack at a later time, while enforcing that the contents of the register do not change is given by:

```
EF mov($r, FFh) & E %noassign($r) U push($r)
```

- `%syscall(`*function, param1, param2, ...*`)`. System calls usually follow the pattern explained in Section 3.2.6: Parameters are pushed onto the stack either directly or indirectly by assigning them to a register which is subsequently pushed—code of the latter form is commonly produced by compilers when the same constant is used multiple times; finally the call is executed. The `%syscall` macro generates a CTPL formula that models exactly this behavior, and allows for direct or indirect parameter initialization. To this end, it uses multiple location predicates to tie the different initialization branches together and to enable choice between the individual ways of initialization. For picking the correct initialization instructions, it relies on a simple type system:
  - Parameters that are a variable and start with the letter *i* are treated as immediate parameters that need no initialization branch. Thus, they are represented by a simple `push(`*variable*`)` in the main system call branch of the formula. The wildcard `$*` is handled analogously and translated to `push($*)`.
  - Every parameter that is a variable and starts with the letter *p* is treated as a memory address and is thus expected to be initialized by using `lea` to load the memory address into some register, which is then pushed onto the stack.
  - For parameters that fit in neither category, the expanded formula will specify both the possibility to be passed directly (as immediate parameters) or indirectly using an intermediate register and an initial `mov($r,`*param*`)`.

  This macro is different from the aforementioned in that it is not simply expanded by substituting a portion of text at the same position in the formula; the macro string in the formula is merely replaced by the final call instruction, paired with a location predicate used as anchor for the initialization branches, resulting to

`call(`*function*`) & #loc(Lc`*n*`)`

where n is a number unique to every macro invocation. Regarding the use of this macro in a CTPL formula, this also implies that the macro is anchored in the state of its call instruction. This is important, as it leads to natural semantics of statements such as

`E %noassign($pFile) U %syscall(CopyFileA, $pFile)`,

and allows for partially parallel parameter initialization of subsequent system calls, i.e. to reuse the intermediate registers filled with values needed by both calls.

Other formula branches are put into conjunction with the CTPL formula. There is one branch defining the correct sequence of push instructions, ending with a reference to the location predicate of the call state. Every push instruction in this branch is paired with a location predicate of its own, serving as anchor for the individual parameter initialization subformulas, which constitute the remaining branches. For performance reasons, location predicates are always paired with an instruction, even if this causes some redundancy in the formula.

- `%sysfunc(`*variable, function, param1, param2, ...*`)`. The `%sysfunc` macro specifies a system call, too, but takes return values into account. The arguments of the macro resemble those of `%syscall`, with the addition of the parameter *variable* that denotes the place in which the return value of the system call is to be stored.

  Windows API functions return 32bit values by storing them in the `eax` register. If the value is to be saved for later use, there must be an instruction in which `eax` is moved to some other location (which is specified in *variable*). This instruction also has to be the anchor state for this macro because in the case of any earlier anchor state, `%noassign` could not be used on *variable* after a `%sysfunc`; the assignment of the return value in `eax` to the variable would contradict the consistency requirement. To this end, the macro string is replaced with `mov(`*variable*`, eax) & #loc(Lc`n`)`, where again n is unique for each macro occurrence. The actual call branch is appended at the end of the formula, specified to prohibit the return value in `eax` from being overwritten by either usual assignment or another call:

```
EF(%syscall(function) & EX(E (%noassign(eax) & -call($*))
          U (mov(variable, eax) & #loc($Ln))))
```

This expansion references the `%syscall` macro defined above, as the call itself is completely analogous; macros are resolved recursively to allow this simplification.

The following example illustrates the behavior and power of macro expansion:

```
EF(
  %sysfunc($ibuf, LockResource) &
  EX(E %noassign($ibuf)
    U %syscall(WriteFile, $ibuf, $*)
  )
)
```

This formula specifies that the return value of the system call `LockResource` is used as an immediate parameter to `WriteFile`, and assures that it is not altered between the two calls. After expansion of the `%syscall` and `%sysfunc` macros, the formula has transformed to this expression:

```
EF(
  mov($ibuf, eax) & #loc($Lc1)
  & EX(E %noassign($ibuf)
    U (call(WriteFile) & #loc($Lc3)))
)
& EF(
  call(LockResource)
  & EX(E (%noassign(eax) & -call($*))
    U (mov($ibuf, eax) & #loc($Lc1))
  )
)
```

```
& EF(
  push($ibuf) & EX(E %nostack
    U (push($*) & EX(E %nostack
      U (call(WriteFile) & #loc($Lc3)))
    )
  )
)
```

The first one of the three large **EF** clauses has been generated from the original formula by replacing the macro strings with their immediate substitutes given in their definitions above. The second clause is derived from the `%sysfunc` macro, and anchored via location `#loc(Lc1)` in the first step of the first clause, while the third clause takes care of parameters for the `WriteFile` call and is anchored via `#loc(Lc3)` at the last step of the first clause. In the first clause, the `mov` and `call` instructions could have been omitted as they are already defined in the other two clauses, and a reference to the location would have been sufficient. However, stand-alone location predicates degrade the performance of the model checker as they cause many subformulas to match in states where they are not needed. Connecting them to the instruction in a conjunction prohibits propagation of the location labels to larger subformulas.

4.3 Experimental Results

4.3.1 Testing Environment

For testing purposes, Mocca has been installed on an AMD Athlon 1800+ machine with 768MB of RAM. The Java virtual machine was configured to allocate 100MB of RAM initially and at maximum 512MB to meet memory demands when checking complex and large procedures.

To evaluate both the performance and detection accuracy, the prototype has been tested on malicious code and benign programs. In particular, the test suite of worms consisted of:

- Various versions of all worms described in Section 2.3, namely
  - *Dumaru*, versions *a* and *b*,

  - *Klez*, versions *a*, *e*, *g*, *h*,
  - *MyDoom*, versions *a*, *i*, *m*, and *aa*, and
  - *NetSky*, versions *b*, *c*, *d*, *e*, and *p*.

- *Badtrans.a*. The e-mail worm Badtrans originated in April 2001. Besides the usual propagation routines, it also drops an executable on the system that logs keystrokes of the users.

- *Bugbear*, versions *a* and *e*. Bugbear (discovered in September 2002) is yet another e-mail worm that drops a keylogger on the infected systems and uses different threads to open a backdoor and stop anti-virus processes.

- *Nimda*, versions *a* and *e*. Nimda does not only propagate via e-mail, but also infects files both locally and on network shares, and tries to exploit vulnerabilities of IIS web servers. File infection works by overwriting the target file with a generated worm executable where the original file is contained inside the worm as a data resource. Each time the worm starts, it extracts this resource to a file and launches it.

- *Swen.a*. The author of the e-mail worm Swen, which emerged in September 2003, put remarkable effort into the social engineering quality of the e-mails. He even went so far as to mimic the visual behavior of Microsoft patches once run on the system. Furthermore, e-mails sent by Swen also try to exploit a vulnerability in unpatched Microsoft Outlook versions.

Each of the malicious programs was prepared for analysis by disassembling the machine code with IDAPro and writing the disassembly to a plain text file. If an executable was initially packed, the file was extracted prior to this step with a suitable tool.

Keeping the false positives rate low is vital for a malware detector; thus it was important to validate that the specifications do not match benign code sequences. To this end, the following programs were disassembled and afterwards analyzed by the prototype:

- `notepad.exe`: The standard Microsoft Windows text editor.

- `cvs.exe`: Release 2.0.51 of the Free Software Foundation's Concurrent Versions System for Win32.
- `j2re-1_4_2_06-windows-i586-p.exe`: The windows installer of the Java Runtime Environment 1.4.6. Setup programs are of particular interest, as they copy files to different locations. This is a behavior similar to the first specification in Figure 4.3, and might thus yield false positives.
- `winamp276_full.exe`: The installer of the Winamp media player, version 2.76.
- `setup.exe`: Setup program of the Cygwin runtime environment for UNIX programs on Windows platforms. The executable is originally UPX-packed, but has been unpacked for testing.

Most of these tested programs perform a large number of file manipulations. Due to the nature of the specifications, programs that use system calls related to file operations could be expected to be the most prone to generate false positives, and have thus been preferred when selecting the test subjects.

4.3.2 Specifications

All executables were tested against two specifications. The first one was described in Section 3.2.6, and specifies that an executable retrieves its own filename and copies itself to a different location. The corresponding input file can be seen in Figure 4.3.

The other specification describes a program that first opens a file and later executes it. This behavior might not seem malicious at a first glance, but it is exactly the behavior of Trojan droppers, which create a file containing a Trojan horse somewhere on the hard drive and execute it. While there might be benign programs that also match this specification, e.g. integrated development environments, it is still a strong indicator for malware if it is found in an executable that is not expected to exhibit such behavior.

The somewhat lengthy specification is given in Figure 4.4. The formula owes its complexity mostly to two facts: First, both the system call `CreateFile` and the C library function `fopen` can be used to open a file. Second, the signature of the `CreateProcess` system call (given in Figure 4.5),

```
exists $r0 (
  EF(lea($r0, $pfname) & EX(E %noassign($r0)
    U (push($r0) & EX(E %nostack
      U ((call(CreateFileA) | call(fopen)) & #loc($Lopen))
; Both CreateFile and fopen may be used to open the file
  )))))
& exists $Lp1 exists $Lp2 (
  EF(push($*) & #loc($Lp2) & EX(E %nostack
    U (push($*) & #loc($Lp1) & EX(E %nostack
      U (call(CreateProcessA) & #loc($Lproc))
  ))))
& (exists $r0 (
  ; CreateProcess parameters variant 1
  EF(lea($r0, $pfname) & EX(E %noassign($r0)
    U (push($r0) & #loc($Lp1)))))
  | exists $r0 (
  ; CreateProcess parameters variant 2
  EF(lea($r0, $pfname) & EX(E %noassign($r0)
    U (push($r0) & #loc($Lp2)))))
  & (EF(push(0) & #loc($Lp1))
     | exists $r1 (
       EF(mov($r1,0) & EX(E %noassign($r1)
         U (push($r1) & #loc($Lp1))
  ))))))
& EF (call($*) & #loc($Lopen) & EF(call($*) & #loc($Lproc)))
```

Figure 4.4: Formula of the ‘ExecOpenedFile’ specification.

```
BOOL CreateProcess(
    LPCTSTR lpApplicationName,
    LPTSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);
```

Figure 4.5: Signature of the CreateProcess WinAPI function.

which is used to execute a file, makes it necessary to formulate alternatives for the parameter ordering. `CreateProcess` takes a total of ten parameters; in particular, the path and name of the file to be executed can be either passed as the first parameter *lpApplicationName*, or, if *lpApplicationName* is set to zero, as the first token of the command line string passed as second parameter, *lpCommandLine*.

In this case, the need to consider alternatives in the invocations makes it more efficient—and actually smaller in the expanded form—to code the calls manually and reuse as many subformulas as possible than to take advantage of system call macros.

4.3.3 Results

Testing results are displayed in Table 4.1. To give a measure for the amount of code in a program, it lists the number of procedures contained in each disassembled executable. Every program was tested against both specifications, and the time was taken individually. The depicted time reflects the processing time of Mocca, namely parsing of the assembler file, constructing the control flow graph, and model checking. Time used for unpacking (by various tools) or disassembly (by IDAPro) is not included, as it was not influenced by the prototype.

Tested Program	**Proc. count**	**CopySelf**		**ExecOpened**		**Result**
		Time	Match	Time	Match	
J2RE Installer	719	50531	n	19824	n	+
Notepad	74	31	n	16	n	+
CVS	1057	937	n	5890	n	+
Winamp Installer	50	53641	n	58469	n	+
Cygwin Setup	2031	860	n	1281	n	+
Badtrans.a	36	37141	n	102016	y	+
Bugbear.a	226	1078	y	9297	y	+
Bugbear.e	199	1719	n	1640	n	−
Dumaru.a	45	3797	y	12	n	+
Dumaru.b	78	3687	y	78	n	+
Klez.a	73	2250	y	282	n	+
Klez.e	130	5891	y	262	n	+
Klez.g	130	5829	y	313	n	+
Klez.h	133	6031	y	343	n	+
MyDoom.a	92	2781	y	2719	n	+
MyDoom.i	116	2281	y	31	n	+
MyDoom.m	97	2203	y	1593	n	+
MyDoom.aa	98	2047	y	31	n	+
NetSky.b	30	563	y	63	n	+
NetSky.c	31	587	y	15	n	+
NetSky.d	27	1937	y	16	n	+
NetSky.e	31	2437	y	<1	n	+
NetSky.p	5	641	y	<1	n	+
Nimda.a	87	62	n	3437	y	+
Nimda.e	87	47	n	4938	y	+
Swen.a	74	121469	y	3172	n	+

Table 4.1: Results from testing various programs against both specifications. A '+' in the results column denotes that an executable has been correctly categorized, a '−' the opposite. Time is given in ms.

The figure shows that Mocca was able to correctly categorize all but one executable, producing no false positives on the benign code. Analysis times on large or complex programs are rather high (up to 2 minutes), but most worms could be detected in less than 6 seconds. Analysis times below 100 milliseconds generally indicate that almost all procedures were skipped due to not matching one of the listed clues. However, the implementation is a prototype and far from being optimized; improved data structures such as OBDDs will allow more efficient operations on binding sets and can be expected to further lower checking times.

The original model for designing the 'CopySelf' specification (Figure 4.3) was derived from analyzing the source code of the worms NetSky.b and MyDoom.a. As the results show, this one specification does not only match these two families of worms, but even a whole class of functionally related malware.

Conclusion

Methods for detecting malicious code have not significantly improved over the last decades. The Internet has multiplied the replication speed of malware, and at the same time allows curious teenagers to get hold of worm source code and virus toolkits. These facts result in a large number of worm outbreaks, each wave surpassing the previous one in economic damage and infection count.

This thesis introduced a novel approach towards a new generation of malware detection. To allow for more general malware specifications that are not limited to a specific variant of a virus or worm, the new temporal logic CTPL was introduced. CTPL allows to write succinct specifications that capture the typical behavior of a virus or worm. In particular, two specifications were created that match a wide variety of current e-mail worms. To further simplify the design of specifications, several macros were implemented that encapsulate common specification patterns. The feasibility of the approach was demonstrated by implementing a prototype of a CTPL based model checker that validates behavior of assembler programs against malicious code specifications.

The experimental results show that CTPL model checking is a promising approach for robust detection of whole classes of functionally similar worms and viruses. There are several opportunities for future research to improve the efficiency of this technique: By abstracting x86 assembler code into a non-ambiguous form, specifications can be made smaller and more general through elimination of equivalent alternatives. Moreover, the use of efficient data structures such as OBDDs to hold the Boolean formulas representing satisfying variable assignments can be expected to further improve the overall performance of the model checker.

It is a widely accepted fact that semantic analysis is the future of malware detection. As new techniques such as the one described in this thesis are being developed, there is hope that semantic methods will be the tool to once again shift the balance of powers on the Internet towards a more secure and reliable future of computer networks.

Bibliography

[BDD+01] J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y.Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *Symposium on Requirements Engineering for Information Security*, March 2001.

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

[CE81] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CGL99] E. Clarke, O. Grumberg, and D. Long. *Model Checking*. MIT Press, 1999.

[Ciu04] M. Ciubotariu. Netsky: conflict starter? *Virus Bulletin*, May 2004.

[CJ03] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186. USENIX Association, August 2003.

[CJ04] M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'04)*, 2004.

[Coh87] F. Cohen. Computer viruses: theory and experiments. *Computers and Security*, 6(1):22–35, February 1987.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997.

[Esp00] D. Esposito. A programmer's perspective on NTFS 2000. *MSDN Online*, March 2000.

[Fer04] P. Ferrie. How Dumaru? the W32/Dumaru family. *Virus Bulletin*, March 2004.

[HR00] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Int04] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, 2004.

[JQsx] Jibz, Qwerton, snaker, and xineohP. PEiD. `http://peid.has.it/`. (Last accessed: 18 Apr. 2005).

[LD03] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM Conference on Computer and Communications Security (CCS 2003)*, October 2003.

[LS03] A. Lakhotia and P. Singh. Challenges in getting 'formal' with viruses. *Virus Bulletin*, September 2003.

[Mar01] Alan Martin. Adequate sets of temporal connectives in CTL. *Electronic Notes in Theoretical Computer Science*, 52(1), 2001.

[mi204] mi2g. MyDoom causes $3 billion in damages as SCO offers $250k reward. January 2004. `http://www.mi2g.com/cgi/mi2g/press/280104.php`.

[Nor03] Norman ASA. Norman sandbox whitepaper. Technical report, 2003.

[OM] M. Oberhumer and L. Molnár. Ultimate Packer for eXecutables. `http://upx.sourceforge.net/`. (Last accessed: 14 Apr. 2005).

[Pie02] M. Pietrek. An in-depth look into the Win32 portable executable file format. *MSDN Magazine*, February 2002.

[Pnu81] A. Pnueli. A temporal logic of programs. *Theoretical Computer Science*, 13:45–60, 1981.

[SL03] P. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *4th IEEE Information Assurance Workshop*, June 2003.

[SM73] L. Stockmeyer and A. Meyer. Word problems requiring exponential time(preliminary report). In *Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 1–9. ACM Press, 1973.

[Sza04a] G. Szappanos. Doomquest: Life after MyDoom. *Virus Bulletin*, April 2004.

[Sza04b] G. Szappanos. We're all doomed. *Virus Bulletin*, March 2004.

[Wil] The WildList Organization International. WildList. `http://www.wildlist.org/WildList`. (Last accessed: 30 Mar. 2005).

[Xtr] Xtreeme. Fast Small Good. `http://www.xtreeme.prv.pl/`. (Last accessed: 16 Dec. 2004).

Index