Proactive Detection of Computer Worms Using Model Checking

Johannes Kinder, Stefan Katzenbeisser, *Member, IEEE,* Christian Schallhart, and Helmut Veith

Abstract—Although recent estimates are speaking of 200,000 different viruses, worms, and Trojan horses, the majority of them are variants of previously existing malware. As these variants mostly differ in their binary representation rather than their functionality, they can be recognized by analyzing the program behavior, even though they are not covered by the signature databases of current anti-virus tools. Proactive malware detectors mitigate this risk by detection procedures which use a single signature to detect whole classes of functionally related malware without signature updates. It is evident that the quality of proactive detection procedures depends on their ability to analyze the semantics of the binary.

In this paper, we propose the use of model checking—a well established software verification technique—for proactive malware detection. We describe a tool which extracts an annotated control flow graph from the binary and automatically verifies it against a formal malware specification. To this end, we introduce the new specification language CTPL, which balances the high expressive power needed for malware signatures with efficient model checking algorithms. Our experiments demonstrate that our technique indeed is able to recognize variants of existing malware with a low risk of false positives.

Index Terms—Invasive Software, Model Checking.

1 Introduction

The Internet connects a vast number of personal computers, most of which run Microsoft Windows operating systems on x86-compatible architectures. Recent global security incidents have shown that this monoculture is a very attractive target for e-mail worms, self-replicating malicious programs which rely on users opening e-mail attachments out of curiosity. Spreading with this rather unsophisticated method, various versions of *NetSky*, *MyDoom*, and *Bagle* have dominated the malware statistics of 2004 and are still regularly seen in top 10 threat lists in 2008. *MyDoom* alone caused a total economic damage of around 3 billion US Dollars during the phase of its initial outbreak [2].

Despite the high economic damage, it is relatively easy to develop e-mail worms: In contrast to the 'classic' viruses of the pre-Internet era, which spread by infecting executable files, e-mail worms that infect hundreds of thousands of computers can be created without knowledge of system programming or assembly language. Today's e-mail worms are commonly written in high-level programming languages: *MyDoom* and *NetSky*, for example, have been created using Microsoft Visual C++. The source code of these malicious programs is often spread via Internet forums, making it accessible to the broad public including the infamous *script kiddies*. Since high-level code can be easily altered and recompiled, several modified versions of the worm (called *variants*) typically appear in the wild shortly after the initial release of the worm. In addition to variants created by different individuals, the original author usually releases his own improved and extended versions of the worm in short order. For example, *NetSky* exists in more than 30 versions, of which up to three were released during one single day. While worm variants differ only slightly from the original in terms of functionality, the executable file can differ significantly, depending on the compiler and its optimization settings.

Due to these differences in the binary representation, current anti-virus products usually fail to detect new malware variants unless an update is supplied [3]. Their principal detection method relies on a large database of *virus signatures*, binary strings of known viral code. A file is assumed to be malicious if it contains one of the signature strings. In order to minimize false positives of the detector, signatures are chosen very narrowly so that one signature matches exactly one specific worm. Consequently, a signature designed for one version of a virus or worm will usually not match against future variants thereof. Anti-virus vendors address this problem by 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 time span between the release of a new worm (variant) and the next database update, called 'window of vulnerability', during which the new malware remains undetected by conventional scanners.

Removing or shortening this window of vulnerability, i.e., enhancing malware detectors in a way that they

- *J. Kinder, S. Katzenbeisser, and H. Veith are with Technische Universität Darmstadt, Fachbereich Informatik, Hochschulstr. 10, D-64289 Darmstadt, Germany. E-mail: {kinder, veith}@cs.tu-darmstadt.de, skatzenbeisser@acm.org.*
- *C. Schallhart is with Technische Universität München, Fakultät für Informatik, Boltzmannstr. 3, D-85748 Garching, Germany. E-mail: schallha@in.tum.de*

Manuscript received 8 Feb. 2007; revised 2 Jun. 2008; accepted 5 Nov. 2008. A preliminary version of this paper appeared at the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'05) [1].

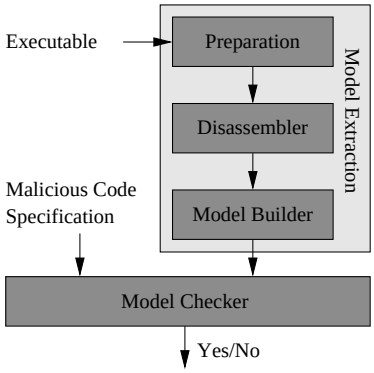


Fig. 1. Architecture of the proposed worm detector.

are capable of detecting worms which were not known at the time of the last signature update, is one of the central problems considered in anti-virus research. Even though theoretical results [4] show that it is impossible to construct a universal malware detector (i.e., a detector that is capable of detecting every foreseeable malware), malicious code that performs similar actions as known malware can be identified by a *proactive* malware detector. Such a detector does not use the mere syntactic representation of a worm as a detection criterion, but rather checks for the presence of malicious *behavior* in suspicious binaries. This approach allows to reliably detect a whole class of malicious programs that share a common functionality, even if their precise syntactic representations are not known beforehand. Rather than using syntactic malware signatures, proactive detectors use signatures that match known malicious *functionality*. In particular, this approach lends itself for the detection of worm variants.

In this paper we outline a proactive semantic detection tool for computer worms which employs *model checking*, an algorithmic formal verification method widely used for verifying correctness properties of hard- and software. The architecture of the malware detector is depicted in Figure 1. The core component of the detector is a model checker that checks a model of a potentially malicious program against malicious code specifications, i.e., malware signatures specifying known malicious behavior, abstracted from implementation details. The program model is produced by a model extractor, which prepares a potentially malicious program for disassembly (e.g., by unpacking or decryption), passes the preprocessed code to a disassembler and extracts a simplified finite-state model from the disassembly. The specifications are written in the temporal logic CTPL, which is an extension of the classic branching-time logic CTL that lends itself well to the specification of code behavior. Using CTPL specifications, we were able to write signatures that match a large class of functionally related worms; for example, *one single CTPL malware signature* matches several variants of the *NetSky*, *MyDoom* and *Klez* worms without producing false positives.

The rest of the paper is structured as follows. In Sections 2 and 3 we review related work and the basic principles of model checking. Section 4 details the model extraction component of the detector. Our specification logic CTPL is introduced in Section 5; the capabilities of CTPL to specify malicious code behavior are demonstrated in Section 6. An efficient model checker for CTPL is introduced in Section 7; experimental results and limitations of the approach are discussed in Section 8. Finally, Section 9 concludes.

2 Related Work

Mitigating the above-mentioned shortcomings of traditional signature matching [3] has spurred academic research on improved methods for malware detection and analysis. Besides lightweight, statistical methods designed for use in Network Intrusion Detection Systems, where speed is critical [5], [6], there are a number of strategies which attempt to detect malicious *behavior*. These techniques follow two principal approaches, which are occasionally combined: *Dynamic Analysis* is performed on actual execution traces of the program under examination, while *Static Analysis* processes the whole program, i.e., all possible execution paths.

In Host-based Intrusion Detection Systems (HIDS), dynamic analysis is implemented as a live analysis program that runs in the background of a production system. Such a background monitor looks for suspicious sequences of system calls or for anomalies in the execution behavior of known system services [7]. While a HIDS provides decent protection at runtime, it cannot ensure that a program will not behave maliciously in the future. Alternatively, dynamic analysis can be performed by using *sandboxes*. A sandbox consists of a virtual machine and a simulated environment which are used to run suspicious executables for a limited time span. Depending on the sequence of actions, the simulated program is classified as malicious or benign [8], [9], [10], [11]. Sandbox-based systems suffer from the same restrictions as HIDS, but additionally monitor a program only over a short timeframe, making them blind towards malware that does not execute its payload right away.

To overcome this inherent problem of dynamic analysis, Moser et al. [12] adopted techniques from directed testing [13], [14] to explore multiple execution paths of a process. During execution of the process, they save memory snapshots at every conditional jump which depends on certain types of input data (e.g., system time, network data, etc.). Subsequently, they restore each snapshot, patch all data so that the opposite branch condition is met, and run the process again. The analysis tool can thus detect behavior which is triggered by certain dates or network commands. Still, timeouts are required in case of long computations or if a large number snapshots is created, which cause the tool to miss malicious behavior in many cases (the paper reports 42%).

Static analysis determines properties of all possible execution paths of a program. A malicious code detector

using static analysis classifies programs as malicious or benign according to their *potential* behavior. This is achieved by disassembling an executable and analyzing its control and data flow, without actually running the program. Bergeron et al. [15] proposed an early system which checks the control flow graph of a disassembled binary for suspicious system call sequences. The system does not allow to specify data dependencies, which are required to reliably identify malicious code (in particular the scheme is not able to express all specifications of Section 6).

Singh and Lakhotia [16] sketched a system that translates the control flow graph of a binary into input for the model checker SPIN. Viral specifications are written in the linear temporal logic LTL and encode sequences of system calls. Unfortunately, they did not report results on real test data, and in [17] they express serious doubt about the feasibility of this method and malicious code detection by formal methods in general.

Christodorescu and Jha [18], [19], [20] describe a malware detection algorithm which matches disassembled procedures against abstract malware templates. The templates are given as code snippets which describe the code fragment to be detected, but use abstract variable names and functions. The matching algorithm performs unification of template instructions with the actual code, and uses decision procedures to verify semantic equality between templates and the code under analysis. Similar to our approach, their matching algorithm also uses unification and is resilient against simple obfuscations [20]. The fundamental difference to the technique proposed in this paper is that we use an expressive branching time logic to specify malicious code, which allows more flexible specifications and helps to capture a wider range of similar malicious behavior found in different malware families.

3 Model Checking

Model checking [21], [22] is an algorithmic method to automatically verify the correctness of a finite state system with respect to a given specification. During the last years, model checking has become a standard tool in the hardware industry, and is increasingly used to verify the correctness of software. In model checking, a system can be a piece of hardware or software that is represented by a labeled state transition graph, a *Kripke structure*. The nodes of the Kripke structure are labeled with *atomic propositions* which represent properties of the system that hold in the respective states.

Formally, 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 assigns a set of propositions (elements of P) to each state. A proposition $p \in P$ holds in s, if and only if p is contained in the set of labels of the state s, i.e., $M, s \models p \Leftrightarrow p \in L(s)$.

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 suffix of a path π beginning with position i, in such a way that $\pi^0 = \pi$. Π_s denotes the set of all possible paths in M beginning with a state s.

Specifications verified by model checking are often expressed as formulas in a temporal logic such as CTL*, LTL, or CTL. Since the specification logic we will use later in this paper is based on CTL, we give a brief introduction to CTL. More detailed information can be found in the literature [23]. CTL extends propositional logic by six temporal operators which allow to specify the temporal behavior of a system: $\mathbf{A}, \mathbf{E}, \mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}$; $\mathbf{A}$ and $\mathbf{E}$ are path quantifiers that quantify over paths in a Kripke structure, whereas the others are linear-time operators that specify properties along a given path π. If a formula φ holds in a state s of the Kripke structure M, we write $M, s \models \varphi$. $\mathbf{A}\,\varphi$ is true in a state s if φ is true for all paths in Π_s; in contrast, $\mathbf{E}\,\varphi$ is true in a state s if there exists a path in Π_s where φ holds. The other operators express properties of one specific path π: $\mathbf{X}\,p$ is true on a path π if p holds in the first state of π^1, $\mathbf{F}\,p$ is true if p holds somewhere in the future on π, $\mathbf{G}\,p$ is true if p holds globally on π, whereas $p\,\mathbf{U}\,q$ is true if p holds on the path π until q holds. In CTL, path and linear-time operators can occur only pairwise (i.e., in the combinations $\mathbf{AX}, \mathbf{EX}, \mathbf{AU}, \mathbf{EU}, \mathbf{AF}, \mathbf{EF}, \mathbf{AG}, \mathbf{EG}$).

For verification purposes, CTL is used to formulate desired properties of a system. For example, in a concurrent system it should be guaranteed that two processes do not enter a critical section at the same time. In CTL, this requirement can be expressed by the formula $\mathbf{AG}\neg(\mathit{crit}_1 \wedge \mathit{crit}_2)$, where crit_1 and crit_2 are propositions assigned to each state in the critical sections of the two processes. As a second example, one may want to ensure that a file opened by a process will eventually be closed again. Using two propositions *open* and *close* which hold in those states where the respective I/O operations are invoked, a CTL specification for this property can be written as $\mathbf{AG}(\mathit{open} \rightarrow \mathbf{AF}\,\mathit{close})$. Conversely, we can specify that a file is never closed without having been previously opened by the formula $\mathbf{A}[\neg\mathit{close}\ \mathbf{U}\ \mathit{open}]$.

Efficient model checking algorithms are known which verify whether a CTL specification φ holds in a Kripke structure. For details, we refer to Clarke et al. [23].

4 Model Extraction

As shown in Figure 1, our proactive worm detection tool employs model checking on a program model in the form of a Kripke structure. In this section we detail the model extraction component, which is a three-stage process: First, the program has to be prepared for disassembly (i.e., by removing dynamic packing or encryption routines) so that the potentially malicious binary code is exposed. After this preprocessing step, the resulting binary code is passed to a disassembler. Finally, a Kripke structure representing the control flow graph of the disassembly is constructed.

4.1 Preprocessing

Most computer worms are *packed* with freely available executable packers such as UPX [24] or FSG [25], which allow to reduce the size of an executable in order to save bandwidth or disk space. These tools compress and/or encrypt an executable and add an extraction routine to the compressed file. Every time the packed executable is run, this routine decompresses and decrypts the original binary into system memory and cedes control to the unpacked binary. Packers are routinely used by malware authors both to reduce the file size (thereby increasing the infection rate) and to 'hide' the malicious code from plain inspection.

In order to get access to the original binary code of a packed program, it first needs to be unpacked. This can either be done dynamically, by allowing the program to load and then dumping the process to disk after decompression has finished [26], or statically with designated unpacking programs. In our implementation, we used static unpackers for UPX, FSG, ASPack, and Petite to obtain unpacked worm samples.

4.2 Disassembling

Once the program is unpacked and one has access to the potentially malicious code, the program is disassembled. In principle, there are two possible disassembly strategies: A *linear sweep* disassembler starts at the entry point of a program and decodes the instructions strictly sequentially, assuming that each instruction is aligned to the next. In contrast, a *recursive traversal* disassembler follows the control flow of the program, resolves the target address of each jump or call instruction and starts disassembling another control flow branch there.

Our current implementation uses IDA Pro [27], a recursive traversal disassembler supported by several heuristics and library signatures. IDA Pro identifies subroutines and local variables, and provides the capability of automatically resolving imported system calls, which allows us to use them in the detection process.

Note that correct disassembly is an important step for any proactive malware detection tool based on static analysis. Unfortunately, program obfuscation methods exist that hinder successful disassembly (see also Section 8.4). Linn and Debray [28] proposed various obfuscation techniques specifically targeted against disassembly. Reversing such obfuscations is possible [29] but out of scope of this paper. Furthermore, the practical relevance of these highly sophisticated obfuscations is currently rather small. The majority of e-mail worms do not employ any obfuscation methods besides the use of executable packers. Typically, a worm is compiled with an industry standard compiler that produces code current disassemblers are able to process successfully.

4.3 Model Builder

Once the assembly source of the potentially malicious program is exposed, we build the program model. Due to complexity reasons, the model builder treats every subroutine of the potentially malicious program separately. The generated program model consists of a set of Kripke structures, each representing the control flow of one subroutine. Each node in the graph represents one machine instruction and is labeled with two *propositions*: A proposition representing the instruction the node corresponds to (including opcode and parameters), and a unique identifier. For example, the instruction `dec ebx` would be represented by the proposition `dec(ebx)`. As unique identifier we use a special proposition #loc(L), where L is some constant value unique to each state; the current implementation uses the offset of the state's instruction as the value of L. More precisely, we produce the Kripke structures as follows:

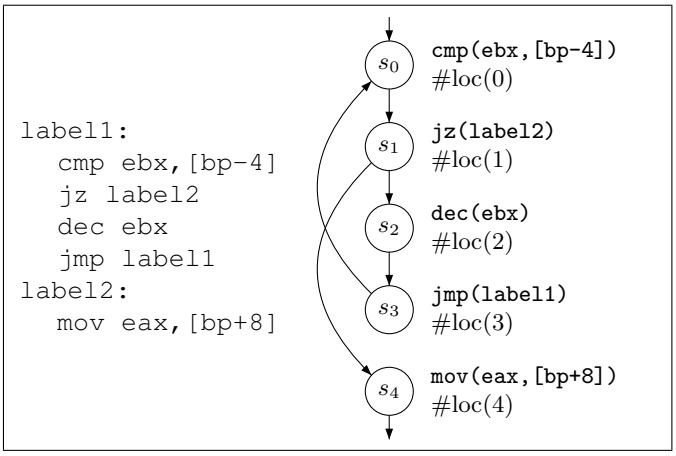


Fig. 2. Executable code sequence and corresponding Kripke structure.

- Every state that is labeled with an unconditional jump (`jmp`) is connected only to its jump target.
- States labeled with a conditional jump (such as `jz`, `jbe`) are modeled as nondeterministic choice; they are connected to both the state representing their immediate successor in the disassembly (the *fall-through successor*) and their jump target.
- Indirect jumps are currently not supported and are thus only connected to themselves to ensure totality of the transition relation (see also Section 8.4).
- Call instructions are connected to their fall-through successor and return statements are treated like indirect jumps. Note that it would be possible to do interprocedural analysis by inlining the called procedure at this point.
- Every other state is connected to its fall-through successor.

Figure 2 demonstrates how a fragment of assembly code is transformed into a Kripke structure according to these rules.

5 Computation Tree Predicate Logic

5.1 Robust Malware Specifications

For malware detection, we use program specifications that describe unwanted (viral) behavior. Specifications for the proactive detection of malicious code differ substantially from specifications used to verify the correctness of programs, since they need to be *robust*: The specifications have to be applicable to all disassembled programs, and should match whenever a program contains the specified *behavior*, regardless of implementation details. Robust specifications should satisfy the following requirements:

- Specifications have to abstract as many implementation details as possible. In particular, they should not specify choices made by the compiler during the compilation step (e.g., register assignment or instruction scheduling).
- Specifications should be resilient against simple instruction level obfuscations, deliberately inserted by malware writers to circumvent detection. Among these obfuscations are dead code insertion (where useless code, such as `nop` or `mov eax,eax`, is inserted between the original instructions), code reordering (where the program is fractured into several pieces, which are ordered randomly and connected through unconditional jumps) and register substitutions (where register names are exchanged consistently in the program). The simplicity of these obfuscations allows to fully automate the obfuscation process, which makes them particularly attractive for use in virus construction kits or polymorphic viruses [30], [31], [32].
- Specifications need to be concise and easy to write.

Classic temporal logics used in verification are unsuited to write robust specifications. For example, consider the statement '*The value `2Fh` is assigned to some register, and the contents of that register is later pushed onto the stack*', which can only be specified in CTL by simply enumerating all possible implementations in one large expression:

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

Robust CTL specifications for assembly code tend to be very large, as a formula invariant against register substitution has to explicitly mention each possible register assignment. In addition, specifications involving memory locations would require inclusion of each individual memory address in the formula, which is clearly infeasible.

In order to account for these difficulties we use the temporal logic CTPL (*Computation Tree Predicate Logic*), which is a subset of first-order CTL [33], [34]. It is equally expressive as CTL, but in general has more succinct specifications. In particular, CTPL allows to write malicious code specifications that differentiate between the opcode of an instruction and its parameters; in addition, the introduction of universal and existential quantifiers simplifies formulas considerably.

5.2 Predicates

As mentioned in Section 3, the basic properties of states in Kripke structures are modeled as atomic propositions. In CTPL, we relax this condition and use predicates as atomic propositions (see Section 4.3). This approach allows specifications for assembly code to differentiate between the opcode of an instruction and its parameters.

More formally, let $\mathcal{U}$ be a finite set of strings. In our setting, $\mathcal{U}$ contains all integer values, all memory addresses, all register names, and combinations thereof (e.g. for indexed addressing). Furthermore, let $\mathcal{N}$ be a set of predicate names. In our setting, $\mathcal{N}$ contains all opcodes and the special symbol #loc. Labels for states in the Kripke structure are chosen from the set $P_{\mathcal{U}}$ of all predicates over elements of $\mathcal{U}$, i.e., $P_{\mathcal{U}} = \{p(t_1, \ldots, t_n) \mid t_1, \ldots, t_n \in \mathcal{U}, p \in \mathcal{N}\}$.

5.3 Syntax of CTPL

Let $\mathcal{X}$ be a set of variables disjoint from $\mathcal{U}$. The syntax of CTPL is defined inductively:

- $\top$ and $\bot$ are CTPL formulas.
- If $p \in \mathcal{N}$ is a predicate of arity $n \geq 0$ and $t_1, \ldots, t_n \in \mathcal{U} \cup \mathcal{X}$, then $p(t_1, \ldots, t_n)$ is a CTPL formula.
- If ψ is a CTPL formula and $x \in \mathcal{X}$, then $\forall x\, \psi$ and $\exists x\, \psi$ are CTPL formulas.
- 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.

A formula is called *closed* if each variable is bound by a quantifier. Without loss of generality, we assume that all bound variables have unique names.

5.4 Semantics of CTPL

We call a partial function B mapping variables $x \in \mathcal{X}$ to constants from the universe $\mathcal{U}$ an environment. In abuse of notation, we let $B(c) = c$ for each constant $c \in \mathcal{U}$. If a CTPL formula φ is true in a state s of a Kripke structure M under environment B, we will write $M, s \models_B \varphi$. A formula φ is true in M (written $M \models \varphi$) if $M, s_0 \models_\emptyset \varphi$ holds for the initial state s_0.

The detailed definition of $\models$ is given in Table 1. In most parts, this definition is similar to the semantics of CTL, modified only to respect the environment B. Rule 1 initializes the environment with the empty binding, implicitly requiring ψ to be closed. Rule 2 defines the semantics of CTPL predicates: A predicate $p(t_1, \ldots t_n)$ over $t_1, \ldots, t_n$ (variables or constants) evaluates to true in a state s with respect to a binding B if and only if s is labeled with $p(B(t_1), \ldots, B(t_n))$. Rule 3 defines the quantifier $\forall$ in the way that $M, s \models_B \forall x\, \psi$ holds if ψ holds with respect to all environments that extend B by

TABLE 1
Semantics of CTPL.

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

a mapping of x to an element of the universe $\mathcal{U}$. Rule 4 handles $\exists$ analogously. All other rules are standard rules for CTL.

6 Specifying Malicious Code Behavior

In this section we show how CTPL can be used to write robust specifications for malicious code, i.e., specifications that abstract from implementation details fixed at compile time and tolerate simple instruction-level obfuscations. These specifications will therefore match against whole classes of malicious programs and allow for proactive detection.

6.1 Modeling Program Behavior

CTPL allows for much flexibility in specifying program behavior. For example, we can formalize the statement '*The program contains an execution path where at some point some register is set to zero and pushed onto the stack in the next instruction*' by the succinct CTPL formula

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

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 specification into '*The program contains an execution path where 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*':

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

Note that this specification does not prevent the presence of instructions between `mov` and `push` that modify the content of the register r. To ensure that the value of `0` is still present in r we can use the CTPL operators **EU**. For example, to disallow any `mov` instruction that writes to register r between the initial `mov` and the final `push`, we can use

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

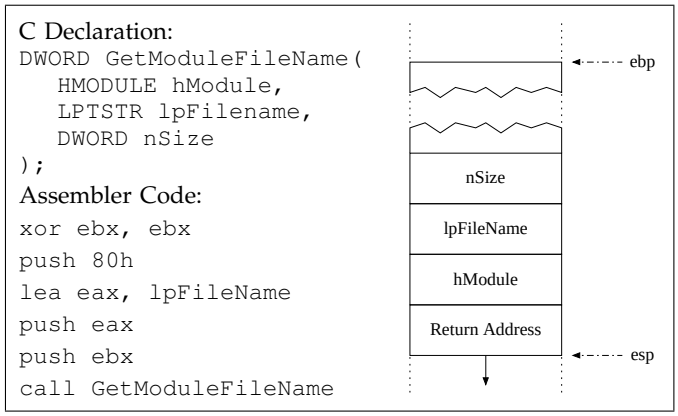


Fig. 3. Local stack frame at the moment of a system call.

Of particular interest for malicious code specifications are calls to the system API, as they are necessary to perform I/O operations, such as network or file access. In assembly language, a system call to the Windows API is made by the `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; for illustration purposes, we use the API 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 ordering produced by the compiler. Such a specification enforces only the correct computation of parameter values and the correct stack layout at the time of the function call. In the specification, we write a conjunction of several different subformulas; one subformula represents the order in which the function parameters are pushed onto the stack, ending with the system call itself, while the other subformulas specify the computation of the individual parameter values. To ensure that the computation of a parameter is completed before it is pushed onto the stack, we enforce an order between certain states across different subformulas using the location predicate #loc(L). As #loc(L) holds in exactly one state for a specific value of L, we can be sure that

multiple occurrences of #loc(L) in the CTPL formula will always reference the same state.

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

$$\exists L \exists r_1 \mathbf{EF}(\texttt{mov}(r_1, 0) \wedge \mathbf{EF}\,\#\text{loc}(L)) \wedge \exists r_2\ \mathbf{EF}(\texttt{push}(r_2) \wedge \mathbf{EF}(\texttt{push}(r_1) \wedge \#\text{loc}(L) \wedge\ \mathbf{EF}\,\texttt{call}(\texttt{fn})))$$

For simplicity, the above formula does not ensure the integrity of registers or the stack. The first subformula (in the first line) expresses that there exists a `mov` instruction in the code that clears some register r_1. After this instruction, there will be some state labeled with the location L, which corresponds to the `push` instruction in which this value is pushed onto the stack. The second subformula (lines 2 and 3) asserts the correct stack layout and 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 subformula) is also pushed onto the stack by the instruction at location L,
- finally a call to function `fn` is executed.

Note that we do not specify a temporal relationship of the two subformulas based on temporal CTPL operators. The only temporal link between them is the location L of the instruction that pushes register r_1.

6.2 An Example: *Klez.h*

The sequence of system calls and their interdependencies are a characteristic feature of a program; therefore they lend themselves to robust specification of malicious programs. In this section we will, as an example, develop a specification for malicious code that operates in a similar manner as the *Klez.h* worm. Worms of the *Klez* family contain infection functionality typical for many e-mail worms. Even though the specification is extracted from one worm instance, it matches several worms that behave in a similar manner (see Section 8.3). The extraction of a malicious code specification from a piece of malware is a manual process; however, a carefully designed specification will match many variants of this malware and malicious code from other families as well.

Figure 4 shows a part of the disassembled infection routine of *Klez.h*. 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, usually a system directory or 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,

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

Fig. 4. Code fragment of the infection routine of *Klez.h.*

2) `lpFilename`: a pointer to a string buffer designated to hold the returned filename, and
3) `nSize`: the size of the string buffer.

In particular, if `hModule` is zero (NULL), the filename of the calling process is returned. 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, and
3) `bFailIfExists`: a flag that determines whether the target file should be overwritten if it already exists.

The fragment of assembly code in Figure 4 displays the invocation of those two system calls in the infection routine, along with the necessary parameter initializations. By abstracting implementation details of the infection routine, we created the CTPL formula in Figure 5. The formula owes its structure to the technique for specifying system calls described in Section 6.1. 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`. The formula is divided into three main subformulas, connected by the $\wedge$ operator. Temporal dependencies 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:

- 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 14, specifies the call to `GetModuleFileName`. It is itself split into three subformulas, which share the following local variables:

```
1  ∃L_m ∃L_c ∃v_File(
2    ∃r_0 ∃r_1 ∃L_0 ∃L_1 ∃c_0(
3      EF(lea(r_0, v_File))
4        ∧EX E(¬∃t(mov(r_0, t) ∨ lea(r_0, t)))U#loc(L_0))
5      ∧EF(mov(r_1, 0)
6        ∧EX E(¬∃t(mov(r_1, t) ∨ lea(r_1, t)))U#loc(L_1))
7      ∧EF(push(c_0)
8        ∧EX E(¬∃t(push(t) ∨ pop(t)))
9          U(push(r_0) ∧ #loc(L_0)
10           ∧EX E(¬∃t(push(t) ∨ pop(t)))
11             U(push(r_1) ∧ #loc(L_1)
12               ∧EX E(¬∃t(push(t) ∨ pop(t)))
13                 U(call(GetModuleFileName)
14                   ∧#loc(L_m))))))
15   ∧∃r_0 ∃L_0(
16     EF(lea(r_0, v_File))
17       ∧EX E(¬∃t(mov(r_0, t) ∨ lea(r_0, t)))U#loc(L_0))
18     ∧EF(push(r_0) ∧ #loc(L_0)
19       ∧EX E(¬∃t(push(t) ∨ pop(t)))
20         U(call(CopyFile) ∧ #loc(L_c)))
21   )
22   ∧EF(#loc(L_m) ∧ EF#loc(L_c))
23 )
```

Fig. 5. CTPL specification for a program that creates copies of itself.

- 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 at which r_1 is pushed, and
- c_0, the size of the string buffer (its correct initialization is not part of the specification).

Line 3 starts the subformula which specifies that the string buffer pointer is stored in r_0. Using the data integrity construction described earlier, line 4 assures that this register is not altered by `mov` or `lea` instructions until the location L_0 is reached. Lines 5 and 6 assert that register r_1 is set to zero and again enforce 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 (where the last two `push` instructions occur at locations L_0 and L_1) and enforce that no stack operations are performed that would break the correct parameter layout. This subformula is concluded by the actual system call to `GetModuleFileName` in line 13, bound to location L_m. Again, the only temporal link between the subformulas is the location predicate.

The system call to `CopyFile` is specified in lines 15 to 21 in a similar way. This time, only two local variables are used:

- r_0, some register (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 r_0 is pushed onto the stack.

The structure is completely analogous to the first system call; lines 16 and 17 assert the loading of the buffer address into r_0; lines 18 and 19 specify r_0 as the first parameter passed to `CopyFile`. This time we can ignore the other parameters because they do not affect the behavior we are interested in.

Line 22 enforces the correct ordering of the two system calls, again by utilizing the location predicate: `GetModuleFileName` has to be called before `CopyFile`, i.e., the location L_m must occur before L_c.

7 Model Checking CTPL

An explicit CTPL model checking algorithm is outlined in Figure 6. It takes a formula φ and a Kripke structure M as input and outputs all states $s \in M$ where φ holds, i.e., $M, s \models_\emptyset \varphi$. We assume that φ is passed as a parse tree, so the algorithm can traverse it in a bottom-up fashion from the smallest subformulas—the predicates—up to the complete formula φ. The algorithm processes each subformula φ' of φ and labels all states with φ' where φ' holds. Alongside each label, the algorithm stores a Boolean formula (*constraint*) in the form of a Boolean circuit, that encodes *all* combinations of variable bindings that make φ' evaluate to true in the respective state. $M \models \varphi$ holds if, after termination of the algorithm, the initial state s_0 of M is labeled with φ and the associated constraint is satisfiable.

The model checker only needs to process predicates, $\bot$, and the operators $\neg$, $\wedge$, $\exists$, **EX**, **EU**, and **AF**, as every formula containing other CTPL connectives can be transformed to use only operators of this subset, due to De Morgan's law and the following equivalences:

$$\top \equiv \neg\bot \qquad \mathbf{EF}\,\psi \equiv \mathbf{E}[\top\,\mathbf{U}\,\psi]$$
$$\forall x\,\psi \equiv \neg\exists x\,\neg\psi \qquad \mathbf{AG}\,\psi \equiv \neg\mathbf{EF}\,\neg\psi$$
$$\mathbf{EG}\,\psi \equiv \neg\mathbf{AF}\,\psi \qquad \mathbf{AX}\,\psi \equiv \neg\mathbf{EX}\,\neg\psi$$
$$\mathbf{A}[\psi_1\,\mathbf{U}\,\psi_2] \equiv \neg(\mathbf{E}[\neg\psi_2\,\mathbf{U}\,(\neg\psi_2 \wedge \neg\psi_1)] \vee \mathbf{EG}\,\neg\psi_2).$$

7.1 Bindings

In contrast to classic CTL, the validity of a CTPL formula depends on the assignment of free variables in the formula, which makes it necessary to maintain bindings between variables $x \in \mathcal{X}$ and constants $c \in \mathcal{U}$. Bindings are generated when the algorithm matches (unifies) a predicate $P(t_1, \ldots, t_n)$ with $t_1, \ldots, t_n \in (\mathcal{U} \cup \mathcal{X})$ in the formula against a predicate $P(c_1, \ldots, c_n)$ with $c_1, \ldots, c_n \in \mathcal{U}$ in a label of the Kripke structure. Every t_i that is a constant has to be equal to the corresponding constant c_i. For every t_i that is a variable, the unification creates a binding $(t_i = c_i)$. The conjunction of these bindings forms the constraint under which $P(t_1, \ldots, t_n)$ holds in the respective state. If there are no variables and all constants match, the constraint is set to $\top$.

As the algorithm traverses the CTPL formula in a bottom-up manner, it propagates and combines constraints in order to deduct the truth of a formula from

the truth values of its subformulas. If a subformula holds only under a certain constraint, this constraint has to be propagated up to every larger formula that includes the subformula. If two subformulas are combined with a binary operator, the accompanying constraints have to be combined as well. For example, if two formulas ψ_1 and ψ_2 hold in a state under the constraints C_1 and C_2, respectively, then $\psi_1 \vee \psi_2$ holds under the combined constraint $C_1 \vee C_2$.

For compactness reasons, the model checking algorithm maintains constraints as circuits with operators of arbitrary arity. We use a lazy evaluation approach in handling these circuits, as the algorithm does not check satisfiability of constraints during runtime. A final satisfiability check is performed for each state only after the labeling is complete.

7.2 The Algorithm in Detail

Starting from the initial labeling L of the Kripke structure, the model checking algorithm maintains a labeling L' that assigns subformulas together with constraints to the nodes of the Kripke structure. More formally, $L' \subseteq (S \times \Phi \times \mathcal{C})$, where Φ is the set of all CTPL formulas and $\mathcal{C}$ is the set of all constraints. In particular, a tuple (s, φ', C) is stored in L', if the subformula φ' holds in state s under the constraint C.

The main loop of the algorithm in Figure 6 iterates over all subformulas φ' of φ in ascending order of size. Depending on the structure of φ', individual procedures are called. After all subformulas have been processed, the algorithm outputs all states labeled with φ whose corresponding constraints are satisfiable.

The first procedure, $\texttt{LABEL}_\text{p}$, handles all subformulas that are predicates, the smallest CTPL formulas. It adds the tuple (s, φ', C) to the labeling relation L' for every state s in which the predicate $p(t_1, \ldots, t_n)$ holds under some variable bindings. 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 2), i.e., whether $(s, p(c_1, \ldots, c_n)) \in L$. Whenever such a predicate has been found, the procedure creates a new constraint C from the conjunction of all variable bindings necessary to unify φ' with the predicate present in the initial labeling of s.

The next subroutine processes negations of the form $\varphi' = \neg\psi$. If the enclosed subformula ψ does not hold in s, i.e., s is not already labeled with ψ, then obviously $M, s \models_\emptyset \neg\psi$ holds and a label $(s, \neg\psi, \top)$ can be added (line 5). If s does have a label ψ, then $\neg\psi$ still holds in s if the constraint for ψ in s is violated. Consequently, the constraint for φ' is created from the negation of the constraint for ψ (unless ψ has an empty constraint).

Conjunctions are handled by the procedure $\texttt{LABEL}_\wedge$. It labels those states in which both subformulas ψ_1 and ψ_2 hold. The constraint C' for their conjunction has to be consistent with the individual constraints associated to ψ_1 and ψ_2, therefore C' is the conjunction of the two individual constraints, $C' := C_1 \wedge C_2$.

Existential quantifiers $\exists x \psi$ are processed in subroutine $\texttt{LABEL}_\exists$. The subroutine iterates over all states labeled with ψ; each state is labeled with φ' and a constraint C', which is a copy of C, where all occurrences of x are replaced by a new variable x_s that is unique to s. Note that $\exists x \psi$ is always evaluated in a specific state s and that the range of assignments for x satisfying ψ depends on s. Thus, we are allowed to replace x with a unique variable for each state. This assures that x is bound individually for each state and that assignments to x from different states do not interfere when constraints are combined in future labeling steps.

The next procedure is responsible for **EX** expressions. It iterates over the parents p of those states s labeled with ψ and labels them with $\varphi' = \mathbf{EX}\,\psi$. In case the parent state has already been labeled through another one of its child states, the new constraint is created as the disjunction of the constraints for parent and child. This way, a state with multiple successors collects a disjunction of all constraints from its children.

Subroutine $\texttt{LABEL}_\text{EU}$ handles the until operator. The algorithm first labels all states with $\varphi' = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ that are labeled with ψ_2 (line 2); note that φ' trivially holds in every state in which ψ_2 holds. Then, the algorithm iteratively labels every state with φ' in which ψ_1 holds and which has a successor labeled with φ'. This iteration is implemented by two nested loops: The outer **for** loop (line 4) plays the role of a fixed point iteration bounded by the maximum length of loopless paths. It ends after $|S|$ iterations or when a fixed point is reached (line 5). The inner loop (line 6) iterates all states s already labeled with φ'. Because $\varphi' = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ holds in these states, φ' will also hold in every predecessor state where ψ_1 holds. Consequently, every predecessor state p that is already labeled with ψ_1 is labeled with φ' (lines 7–13). The constraint C of the new label φ' in p has to take into account both C_1, which is associated with ψ_1 in the label of p, and 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 9). The subroutine checks whether p has already been labeled with φ' during a previous step in the fixed point iteration and adds C into a disjunction with the existing constraint (lines 11 and 12). If the label is new, it is added to the state together with C (line 13).

The procedure calculates constraints for φ' as a chain of conjunctions along all paths leading to states in which ψ_2 holds. Multiple paths passing through a state lead to a disjunction of constraints for φ' in this state. However, we only need to follow loopless paths, as the constraints collected from paths containing loops are always subsumed by the constraint gained from the path that does not repeat the loop. Thus we can safely stop the iteration (the outer **for** loop) after $|S|$ steps, which is the maximum length of loopless paths in M.

The procedure for the operator $\mathbf{AF}\,\psi$ is similar in structure to $\texttt{LABEL}_\text{EU}$. First, all states with a label ψ are initially labeled with φ', as $\mathbf{AF}\psi$ is sure to hold where ψ already does. The inner loop for updating labels (line 6)

Algorithm ModelCheckCTPL:
Input: A Kripke structure $M = \langle S, R, L \rangle$, a CTPL formula φ
Output: The set of states in M which satisfy φ

```
1  L' := ∅
2  for all subformulas φ' of φ in ascending order of size
3     case φ' of
4        ⊥: label no states;
5        p(t_1, ..., t_n): LABEL_p(φ');
6        ∃x (ψ): LABEL_∃(φ');
7        ¬ψ : LABEL_¬(φ');
8        ψ_1 ∧ ψ_2: LABEL_∧(φ');
9        EX ψ: LABEL_EX(φ');
10       E[ψ_1 U ψ_2]: LABEL_EU(φ');
11       AF ψ: LABEL_AF(φ');
12 output all states s where ∃C with (s, φ, C) ∈ L' and SAT(C);

1  procedure LABEL_p(φ')   // φ' = p(t_1, ..., t_n)
2     for all s ∈ S if (s, p(c_1, ..., c_n)) ∈ L then
3        C := ⋀_{i=1,...,n}(t_i = c_i) ;
4        L' := L' ∪ {(s, φ', C)};

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', ⊤)};

1  procedure LABEL_∧(φ')   // φ' = ψ_1 ∧ ψ_2
2     for all s ∈ S
3        if ∃C_1, C_2 with {(s, ψ_1, C_1), (s, ψ_2, C_2)} ⊆ L' then
4           L' := L' ∪ {(s, φ', C_1 ∧ C_2)};

1  procedure LABEL_∃(φ')   // φ' = ∃x ψ
2     for all s ∈ S if ∃C with (s, ψ, C) ∈ L' then
3        copy C into C';
4        replace all variables x in C' with a new variable x_s;
5        L' := L' ∪ {(s, φ', C')};

1  procedure LABEL_EX(φ')   // φ' = EX ψ
2     for all s ∈ S
3        if ∃C with (s, ψ, C) ∈ L' then
4           for all (p, s) ∈ R        // for all parents of s
5              if ∃C_p with (p, φ', C_p) ∈ L' then
6                 L' := (L'\(p, φ', C_p))∪
7                       {(p, φ', C ∨ C_p)};
8              else L' := L' ∪ {(p, φ', C)};

1  procedure LABEL_EU(φ')   // φ' = E[ψ_1 U ψ_2]
2     for all s ∈ S if ∃C with (s, ψ_2, C) ∈ L' then
3        L' := L' ∪ (s, φ', C);
4     for i = 1 to |S|          // cover all loopless paths
5        if L' has not changed then return;
6        for all s ∈ S if ∃C_s with (s, φ', C_s) ∈ L' then
7           for all (p, s) ∈ R        // for all parents of s
8              if ∃C_1 with (p, ψ_1, C_1) ∈ L' then
9                 C := C_s ∧ C_1;
10                if ∃C_p with (p, φ', C_p) ∈ L' then
11                   L' := (L'\(p, φ', C_p))∪
12                         {(p, φ', C ∨ C_p)};
13                else L' := L' ∪ {(p, φ', C)};

1  procedure LABEL_AF(φ')   // φ' = AF ψ
2     for all s ∈ S if ∃C with (s, ψ, C) ∈ L' then
3        L' := L' ∪ (s, φ', C);
4     for i = 1 to |S|          // cover all loopless paths
5        if L' has not changed then return;
6        loop: for all s ∈ S
7           C := ⊤;
8           for all (s, c) ∈ R        // for all children of s
9              if ∃C_c with (c, φ', C_c) ∈ L' then
10                C := C ∧ C_c;
11             else continue loop;
12          if ∃C_s with (s, φ', C_s) ∈ L' then
13             L' := (L'\(s, φ', C_s)) ∪ {(c, φ', C ∨ C_s)};
14          else L' := L' ∪ {(c, φ', C)};
```

Fig. 6. CTPL model checking algorithm.

is again enclosed by a bounded fixpoint iteration (lines 4 and 5). The inner loop checks for every state s whether *all* successors of s are currently labeled with φ'. If this is the case, s is labeled with φ'. The constraint C generated for the new label has to take the constraints of all successor states into account: This is accomplished by successively adding the constraints from child states of s labeled with φ' into a conjunction that is initialized with $\top$. If any child state is not labeled with φ' (line 11), the inner loop continues with a new s. If the current state already has a label φ' (line 12), then its constraint is put into disjunction with the new constraint C (line 13). Otherwise, a new label φ' is added to the state with the constraint C (line 14). The arguments as to why it is sufficient to only cover all loopless paths are completely analogous to the case for **EU**. Excessive growth of constraints can be averted, as in the **EU** case, by exploiting their structural simplicity (see Appendix B).

7.3 Complexity

This section discusses the computational complexity of CTPL model checking in general and of our proposed algorithm in particular.

Theorem 1: The problem of CTPL model checking, i.e., deciding whether $M, s \models \varphi$ for a Kripke structure M, a state s of M, and a CTPL formula φ, is **PSPACE**-complete.

We give a detailed proof of this theorem in Appendix A. Due to **PSPACE**-completeness, we cannot expect a consistently efficient algorithm for CTPL model checking. Analysis of our bottom-up labeling algorithm (Figure 6) yields the following worst case runtime estimate:

Theorem 2: The worst case runtime complexity of the bottom-up explicit model checking algorithm for CTPL is $\mathcal{O}\left(\alpha|S|\,|\varphi|2^{\alpha(2|S|)^{\nu}}\right)$, where α is the maximum arity of predicates, and ν is the maximum nesting depth of φ.

We refer the reader to Appendix B for the proof of this theorem. Note that one exponential level necessarily comes from the time required to check satisfiability of the constraints aggregated during the labeling process, the size of which can grow exponentially for degenerate formulas (checking the satisfiability of Boolean circuits is known to be **NP**-complete, thus requires exponential time in the maximum size of constraints). Note further that the above estimate is extremely conservative. Simple optimizations in the implementation of circuits can, in practice, considerably reduce the constraint size for models derived from assembly programs and thus speed up the satisfiability test. Finally, the constraint size is heavily influenced by the nesting depth of specification formulas, so careful specification design can significantly improve performance. Our practical observations confirm that constraint sizes usually do not exhibit exponential behavior; the experimental results presented in Section 8.3 show that our approach is indeed computationally feasible.

8 The Mocca Malware Detector

We have implemented the CTPL model checking algorithm in Java, along with a model extraction component, forming the *Mocca Malware Detector*. In this section, we discuss details of the implementation and experimental results obtained.

8.1 Specification Language

Although CTPL allows succinct specifications for assembly programs, examples such as the formula in Figure 5 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 macros that encapsulate commonly used specification patterns.

A common occurrence in malicious code specifications are variables used as wildcards that match every parameter of 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 $\mathbf{E}(\neg\exists x\ \texttt{push}(x))\ \mathbf{U}\ (\exists y\ \texttt{pop}(y))$, which would translate into Mocca syntax as `E ( -exists x (push(x)) U (exists y (pop(y)))`. To simplify specifications and improve their human readability by reducing the abundance of `exists` statements, Mocca supports the special wildcard `$*`, 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($*)) U pop($*)`.

In addition to wildcards, Mocca supports the definition of macros that expand to CTPL formulas. Currently, we use the following macros to encode common patterns in specifications of assembly code:

- `%nostack`. Prohibits direct stack manipulation.
- `%noassign(`*variable*`)`. Specifies that no value is explicitly assigned to the given variable (register or memory location) in this state.
- `%syscall(`*function, param1, param2, ...*`)`. System calls usually follow the pattern explained in Section 6.1: Parameters are pushed onto the stack either directly or indirectly (by assigning them to a register which is subsequently pushed) before finally the call is executed. The `%syscall` macro generates a CTPL formula that models exactly this behavior, and covers both direct and indirect parameter initialization.

As an illustrative example, the use of macros allows to write the specification in Figure 5 much more succinctly and naturally as:

```
EF(%syscall(GetModuleFileName, $*,
            $pFile, 0) &
      E %noassign($pFile)
         U %syscall(CopyFile, $pFile))
```

Finally, to improve performance of the model checking process, Mocca allows to specify *clues* in the form of instructions (such as specific system calls) that must syntactically occur in a subroutine. If they do not occur, Mocca decides that the checked subroutine cannot fulfill the specification.

8.2 Experimental Setup

For testing purposes, Mocca has been installed on an AMD Athlon 64 2.2 GHz machine with 2 GB of RAM running Windows XP Professional. To evaluate both performance and detection accuracy, the prototype has been tested on malicious code and benign programs. The test suite consisted of 21 worm variants from 8 different families. Each malicious program was prepared for analysis by disassembling the machine code with IDA Pro. 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. To this end, the test setup also included a set of benign programs that should be correctly classified as innocuous. Note that, as in the case of traditional signature-based matching algorithms, the false positives rate of the malware detector measures only the quality of the specifications (for the model checking algorithm itself always correctly answers whether an executable matches a specification or not). Since the malicious code specifications used in the tests heavily relied on I/O operations, the test suite mainly contained benign programs that have file I/O as their main operation (such as setup programs).

8.3 Results

Table 2 shows the test results for two exemplary CTPL specifications: *CopySelf* refers to the formula described

```
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 can be used
  )))))
& 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)))
```

Fig. 7. Malicious code specification in Mocca syntax. It specifies that a new process is created from a previously opened executable file.

in Figure 5; *ExecOpened* matches a program that first opens a file and later executes it (see Figure 7). This second behavior might not seem malicious at 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.

Every program in the test suite was tested against both specifications. In the second column, we give the number of procedures contained in the assembly file. The processing time (in milliseconds), measured individually for each specification, reflects the runtime of Mocca, namely parsing of the assembly file, construction of the control flow graph, and model checking. Time used for unpacking (by various tools) or disassembly (by IDA Pro) is not included.

The figure shows that Mocca was able to correctly categorize all but one executable, producing no false positives on benign code. Recall that the *CopySelf* specification was originally derived from analyzing the behavior of the worm *Klez.h*. As the results show, this one specification does not only match this family of worms, but a whole class of functionally related malware, among them variants of the *MyDoom*, *Dumaru*, and *NetSky* families. This shows that the proposed approach is able to provide proactive malware detection.

Analysis times on large or complex programs are rather high (up to 30 seconds), but most worms could be detected in less than 2 seconds. Analysis times below 50 milliseconds generally indicate that all procedures were skipped due to clues. We want to stress that the current Java implementation is not optimized for speed; we expect that considerable speedups can be achieved.

TABLE 2
Experimental results.

Tested Program	Proc. count	CopySelf		ExecOpened		Result
		Time	Match	Time	Match	
Badtrans.a	36	10640	n	32828	y	+
Bugbear.a	226	359	y	3641	y	+
Bugbear.e	199	609	n	782	n	−
Dumaru.a	45	1172	y	<10	n	+
Dumaru.b	78	1157	y	16	n	+
Klez.a	73	750	y	110	n	+
Klez.e	130	1859	y	125	n	+
Klez.g	130	1860	y	125	n	+
Klez.h	133	1922	y	125	n	+
MyDoom.a	92	750	y	1172	n	+
MyDoom.i	116	766	y	16	n	+
MyDoom.m	97	750	y	656	n	+
MyDoom.aa	98	735	y	15	n	+
NetSky.b	30	234	y	16	n	+
NetSky.c	31	250	y	<10	n	+
NetSky.d	27	657	y	15	n	+
NetSky.e	31	656	y	<10	n	+
NetSky.p	5	250	y	<10	n	+
Nimda.a	87	31	n	1360	y	+
Nimda.e	87	31	n	1375	y	+
Swen.a	74	27016	y	1313	n	+
Notepad	74	31	n	16	n	+
MSN Messenger	4376	781	n	531	n	+
CVS	1057	406	n	1625	n	+
Regedit	305	47	n	49	n	+
Ghostscript	237	47	n	31	n	+
Setup Programs:						
J2RE	719	12641	n	5829	n	+
Winamp	50	15078	n	20312	n	+
Cygwin	2031	391	n	578	n	+
Mouseware	202	49	n	2375	n	+
Quicktime	629	63	n	47	n	+
NVIDIA Driver	403	64	n	2203	n	+
Real Player	484	47	n	46	n	+
Kerio PFW	1099	297	n	6031	n	+

8.4 Discussion and Future Work

Our approach, as described here, is still subject to a number of limitations, which stipulated further research in enhancing the practical applicability of Mocca. First, the model extraction process is purely syntactic and does not include data flow analysis, since Mocca has no explicit knowledge of instruction semantics. Instead, semantics are implicit in the CTPL specifications: For example, the `%noassign` macro expands to a formula that specifies the most common ways to modify a given variable. Due to performance reasons, the macro does not exhaustively cover all instructions of the x86 architecture that can be used to modify variables. This imprecision renders the results unsound and potentially leads to false positives,

although we did not encounter such cases in our experiments. A rigorous treatment of all instruction semantics and side-effects requires complete and exact specification of an architecture's instruction set. Therefore, we have developed a semantics-driven model extraction process which utilizes specifications of the instruction set [35]; we plan to integrate this component into the Mocca toolchain. Besides increased precision, the data flow analyses included in this extraction process will allow Mocca to handle several instances of indirect control flow and indirect memory accesses.

As mentioned above, the model checking process currently is strictly intraprocedural. Malicious behavior that is split across several procedures cannot be identified unless procedures are inlined, which incurs a significant performance penalty. Selective inlining of only those procedures which include system calls relevant to the specification offers a remedy to this problem. A complete solution, however, can again only be achieved using interprocedural data flow analysis.

Mocca assumes that the external components of the toolchain successfully unpack and disassemble all executables; however, this assumption does not universally hold. This is especially true for arbitrary self-modifying code, which is not accessible to disassemblers such as IDA Pro. One should note, though, that code which cannot be disassembled is suspicious by its own right and should raise a warning—a strategy employed by most commercial virus scanners that include unpacking modules.

It is desirable to have a streamlined and mostly automated process for the creation of CTPL specifications from malware samples. To this end, Holzer et al. [36] have developed a separate tool supplemental to Mocca, which assists a user in the construction of CTPL specifications from a piece of malicious code. The tool is designed to support automated extraction techniques as well, and we plan to investigate how to effectively employ methods from machine learning to extract characteristic code sequences from a set of worms or viruses.

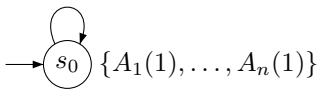
9 Conclusion

In this paper we proposed a proactive malware detector that is specifically targeted towards detection of functionally related worms (or worm variants) without the need of signature updates. Technically, the detector employs model checking. For detection, a finite state model is extracted from a potentially malicious program and checked against a malware specification, expressed in the temporal logic CTPL. CTPL allows to write succinct specifications that capture the typical behavior of a worm, thereby abstracting from implementation details.

The feasibility of the approach was demonstrated by the implementation of a prototype CTPL model checker. Experimental results indicate that it is easily possible to create CTPL malware specifications that match a whole class of functionally related malware, without risks of false positives. Thus, CTPL model checking is a promising approach for robust detection of whole classes of functionally similar malware.

Appendix A
PSPACE Completeness of CTPL Model Checking

We show **PSPACE**-hardness by a reduction from satisfiability of quantified Boolean formulas (QBF). Given a quantified Boolean formula q, we compute a CTPL formula φ from q by replacing every variable a_i in q by a unique unary predicate $A_i(x_i)$ and quantifying the variables x_i in the same way as the corresponding a_i. It is obvious that q is satisfiable if and only if φ is satisfied on the following Kripke structure:



To prove membership in **PSPACE**, we use a recursive top-down model checking algorithm for CTPL, shown in Figure 8, that tries all possible variable bindings one by one. It uses only polynomial space, but its high runtime renders the algorithm unusable in practice. The algorithm passes the formula φ and the initial state of the Kripke structure M to the function `check` that operates recursively on the syntax tree of φ.

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 Boolean return value, so it does not add to the space complexity. In particular, this allows to give the induction basis, because checking of all atomic CTPL formulas, namely $\bot$ (line 4) and all predicates (line 28), does not use space other than its stack frame.

The induction hypothesis is that the maximum space cost $\mathcal{C}(n)$ 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 recursive calls are performed sequentially, so the required space is bounded by $\mathcal{O}(\mathcal{C}(n)+|\varphi| + \log|S|)$, which is polynomial by induction hypothesis.
- $\neg$: This case is immediate from the induction hypothesis.
- **EX** ψ: All recursive calls are sequential, thus the space for every call can be reused. The relation R is already given in the input, so only a counter for

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

```
1  output check(s_0, φ);
2  function check (state s, formula φ') : Boolean
3    case topOperator(φ') of
4      ⊥:        return false;
5      ψ1 ∧ ψ2:  return check(s, ψ1) and check(s, ψ2);
6      ¬ψ:       return not check(s, ψ);
7      EX ψ:     for all (s, c) ∈ R
8                  if check(c, ψ) then return true;
9                return false;
10     E[ψ1 U ψ2]:
11       mark(s, φ');
12       if check(s, ψ2) then return true;
13       else if check(s, ψ1) then
14         for all (s, c) ∈ R
15           if (not isMarked(c, φ')) and check(c, φ') then
16             return true;
17       return false;
18     AF ψ:
19       mark(s, φ');
20       if check(s, ψ) then return true;
21       else for all (s, c) ∈ R
22         if not (isMarked(c, φ') or check(c, φ')) then
23           return false;
24       return true;
25     ∃x ψ:     for all t ∈ U occurring in M
26                 if check(s, ψ[x\t]) then return true;
27               return false;
28     P(t):     if (s, P(t)) ∈ L then return true;
29               else return false;
```

Fig. 8. Recursive model checking algorithm for CTPL.

the **forall** statement is needed. Thus the total space is $\mathcal{O}(\mathcal{C}(n) + |\varphi| + 2\log|S|)$.

- $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$: Note that in this case the `check` function may be called 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 that every state is checked at most once. The space used by sequential `check` calls inside a recursion instance 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 space requirement of $|S| \cdot |\varphi|$. Thus the total space consumption is

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

  This can be simplified to $\mathcal{O}(|S| \cdot (|\varphi| + \log|S|) + \mathcal{C}(n))$, which is obviously polynomial.
- $\mathbf{AF}\,\psi$: The case of **AF** is analogous to the one of **EU**.
- $\exists x\ \psi$: The variable t, which iterates over all symbols of the universe $\mathcal{U}$ used in the model, requires at most $\mathcal{O}(\log|M|)$ space. The individual `check` calls may reuse the same space and thus count only once; substituting the variables in their parameters does not take extra space. Consequently, the total space consumption is $\mathcal{O}(\mathcal{C}(n) + |\varphi| + \log|S| + \log|M|)$, which is again polynomial.

This completes the proof.

APPENDIX B
COMPLEXITY OF THE BOTTOM UP ALGORITHM

We calculate the complexity of the bottom-up model checking algorithm with respect to a number of parameters of the problem instance (the Kripke structure $M = \langle S, R, L\rangle$ and the formula φ). In particular, we use the following notation:

- α denotes the maximum arity of predicates in L,
- ι denotes the maximum number of initial labels per state,
- δ_{out} denotes the maximum outdegree of states in S, and
- δ_{in} denotes the maximum indegree of states in S.

For each procedure handling a specific type of subformula, we recursively analyze its runtime T and the maximum circuit size γ required to store a constraint.

For simplicity, we assume that the labeling relation L' used by the algorithm at runtime is implemented as a two-dimensional array, assigning a constraint to each pair of state and subformula. Thus, references to constraints for states can be retrieved in constant time. Constraints are stored as Boolean circuits of arbitrary arity where the vertices are connected by positive or negative edges (to express negation). Our lazy approach to satisfiability checking together with the use of references for constraints allows constant-time manipulation of constraints in all subroutines except $\texttt{LABEL}_\exists$.

Since predicates are the smallest CTPL formulas, all calls to $\texttt{LABEL}_\text{p}$ are executed before any other procedure. $\texttt{LABEL}_\text{p}$ contains an iteration over $|S|$ states. For every parameter of every predicate from the initial labeling that matches p, one variable assignment is created and stored in the constraint. Overall, we thus have a runtime of $\mathcal{O}(|S| \cdot \iota \cdot \alpha)$ and a maximum constraint size of $\gamma := 2\alpha + 1$, as we need one assignment and one edge per parameter plus one conjunction (which can be of arbitrary arity) to store the constraint.

$\texttt{LABEL}_\neg$ iterates over all states and thus has a runtime of $\mathcal{O}(|S|)$, while the maximum constraint size remains constant since negation can be expressed by inverting the circuit's output edge.

The runtime for $\texttt{LABEL}_\wedge$ is again $\mathcal{O}(|S|)$, however here the maximum circuit size increases to $\gamma = 2\gamma + 3$, since two individual constraints are put into conjunction (and one $\wedge$-node and two edges are required).

The runtime for $\texttt{LABEL}_\exists$ is given by $\mathcal{O}(|S| \cdot \gamma)$, as the algorithm processes all constraints of states labeled with ψ. Here, γ is the current maximum constraint size at the time the procedure is called. Constraint sizes do

not increase since the procedure just renames existing variables.

For every state s, $\texttt{LABEL}_{\texttt{EX}}$ iterates over all of its parents, whose number is bounded by the maximum indegree of nodes δ_{in}. This amounts to a runtime of $\mathcal{O}(|S| \cdot \delta_{in})$. The maximum constraint size grows to $\delta_{out} \cdot (\gamma + 1) + 1$, as every state that is labeled with φ' can inherit one constraint per child.

In $\texttt{LABEL}_{\texttt{EU}}$, initially all states with the label ψ_2 are iterated, requiring $\mathcal{O}(|S|)$ steps. Then, a second loop has a maximum of $|S|$ iterations, while the two inner loops (lines 6 and 7) iterate all states with label φ' and all their parents with label ψ_1, respectively. The runtime for these three nested loops is thus $\mathcal{O}(|S|^2 \cdot \delta_{in})$, which is also the runtime of the whole subroutine.

In order to keep the size of the circuit representing the constraint $C \vee C_p$ (line 12) low, we exploit the structure of the constraint $C_{s,\varphi'}$ for the formula $\varphi' = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ in state s. It is easy to see that $C_{s,\varphi'}$ has the form $(C_{s,\psi_1} \wedge (C_{t_1,\varphi'} \vee \ldots \vee C_{t_n,\varphi'})) \vee C_{s,\psi_2}$, where $t_1, \ldots, t_n$ denote successor states of s for which φ' holds. Note that the constraints $C_{t_i,\varphi'}$ link to other constraints of the same structure as $C_{s,\varphi'}$. To avoid the accumulation of large trees of constraints for every state, we reuse references to the same constraint in the circuit. Thus, each constraint $C_{s',\varphi'}$ corresponding to a state s' is stored only once in the resulting circuit. In worst case, we then have $|S|$ many of these structures consisting of one conjunction, two disjunctions, two constraints for smaller subformulas, and $\delta_{out} + 2$ edges. Accordingly, the complete circuit for one state occupies a space of $|S|(2\gamma + \delta_{out} + 5)$.

The initial loop of $\texttt{LABEL}_{\texttt{AF}}$ iterates all states with label ψ in $\mathcal{O}(|S|)$ steps. The following outer loop is repeated at most $|S|$ times. Inside this loop, there is an iteration over all states ($|S|$) and another nested loop over all children (δ_{out}). Thus, the total runtime is $\mathcal{O}(|S| + |S|^2 \cdot \delta_{out}) = \mathcal{O}(|S|^2 \cdot \delta_{out})$. Similar to the above routine for **EU**, we exploit the structure of the constraints to keep their size low. A constraint $C_{s,\varphi'}$ for $\varphi' = \mathbf{AF}\, \psi$ in state s is of the form $C_{s,\psi} \vee (C_{t_1,\varphi'} \wedge \ldots \wedge C_{t_n,\varphi'})$, where $t_1, \ldots, t_n$ denote successor states of s, whose constraints $C_{t_i,\varphi'}$ for φ' are of the same structure as $C_{s,\varphi'}$. In worst case, we get $|S|$ such structures, which contain one disjunction, one conjunction, one constraint for a smaller subformula, and $\delta_{out}+1$ edges. The final maximum circuit size is thus $|S|(\gamma + \delta_{out} + 3)$.

To analyze the worst case runtime and circuit size, we denote with T_i the total runtime of the main model checking algorithm after i iterations of the for loop over subformulas of φ. Accordingly, γ_j denotes the maximum constraint size for a formula of nesting depth j. The following table shows how T_i and γ_j can be recursively computed according to the formula type. Note that the total runtime of the algorithm is given by T_N, where N denotes the number of subformulas of φ, with $N \leq |\varphi|$. Further, the maximum constraint size is given by γ_ν, where ν is the maximum nesting depth of φ.

Op.	T_{i+1}	γ_{j+1}
p	$T_i + \mathcal{O}(\lvert S\rvert \cdot \iota \cdot \alpha)$	$2\alpha + 1$
$\neg$	$T_i + \mathcal{O}(\lvert S\rvert)$	γ_j
$\wedge$	$T_i + \mathcal{O}(\lvert S\rvert)$	$2\gamma_j + 3$
$\exists$	$T_i + \mathcal{O}(\lvert S\rvert \cdot \gamma_\nu)$	γ_j
EX	$T_i + \mathcal{O}(\lvert S\rvert \cdot \delta_{in})$	$\delta_{out} \cdot (\gamma_j + 1) + 1$
EU	$T_i + \mathcal{O}(\lvert S\rvert^2 \cdot \delta_{in})$	$\lvert S\rvert(2\gamma_j + \delta_{out} + 5)$
AF	$T_i + \mathcal{O}(\lvert S\rvert^2 \cdot \delta_{out})$	$\lvert S\rvert(\gamma_j + \delta_{out} + 3)$

First, we calculate the maximum γ_ν. We set the base case to $\gamma_0 = 2\alpha + 1$, i.e., the size of a constraint for a single predicate. We obtain the worst case by taking the maximum growth in constraint size, which occurs in **EU**. Solving the recursion then yields $\gamma_\nu = \mathcal{O}(\alpha(2|S|)^\nu)$. Since γ grows exponentially, the procedure with the highest worst case runtime is $\texttt{LABEL}_\exists$. Solving the recursion for T_i accordingly yields $T_N = \mathcal{O}(|\varphi|\,|S|\gamma_\nu)$. Taking into account the final SAT solving procedure, which needs to performed for each state and requires exponential time in γ_ν, we have

$$\begin{aligned} T_N &= \mathcal{O}\left(\alpha\,|S|\,|\varphi|(2\,|S|)^\nu + |S|\,2^{\alpha(2\,|S|)^\nu}\right) \\ &= \mathcal{O}\left(\alpha\,|S|\,|\varphi| 2^{\alpha(2\,|S|)^\nu}\right). \end{aligned}$$

Note that if we disregard degenerate formulas consisting predominantly of the unary operators **EX** and $\neg$, we have $\nu \approx \mathcal{O}(\log_2 |\varphi|)$, alleviating the seemingly high complexity of the algorithm for practical purposes.

ACKNOWLEDGMENTS

The authors thank Julia Lawall for pointing out an error in the algorithm in an earlier draft of this article.

REFERENCES

[1] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Detecting malicious code by model checking," in *Second Int'l Conf. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2005)*, ser. LNCS, vol. 3548. Springer, 2005, pp. 174–187.

[2] mi2g Ltd. (2004) MyDoom causes $3 billion in damages as SCO offers $250k reward. [Online]. Available: http://www.mi2g.com/cgi/mi2g/press/280104.php

[3] M. Christodorescu and S. Jha, "Testing malware detectors," in *Proc. ACM/SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA 2004)*. ACM Press, 2004, pp. 34–44.

[4] F. Cohen, "Computer viruses: theory and experiments," *Computers and Security*, vol. 6, no. 1, pp. 22–35, Feb. 1987.

[5] J. Newsome, B. Karp, and D. X. Song, "Polygraph: Automatically generating signatures for polymorphic worms," in *IEEE Symp. Security and Privacy (S&P 2005)*. IEEE Computer Society, 2005, pp. 226–241.

[6] M. R. Chouchane, A. Walenstein, and A. Lakhotia, "Statistical signatures for fast filtering of instruction-substituting metamorphic malware," in *Proc. 5th ACM Workshop Recurring Malcode (WORM 2007)*. ACM Press, 2007, pp. 31–37.

[7] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller, "Environment-sensitive intrusion detection," in *Proc. 8th Int'l Symp. Recent Advances in Intrusion Detection (RAID 2005)*, ser. LNCS. Springer, 2005, pp. 185–206.

[8] T. Holz, F. Freiling, and C. Willems, "Toward automated dynamic malware analysis using CWSandbox," in *IEEE Symp. Security and Privacy (S&P 2007)*. IEEE Computer Society, 2007, pp. 32–39.

[9] A. Vasudevan and R. Yerraballi, "Cobra: Fine-grained malware analysis using stealth localized-executions," in *IEEE Symp. Security and Privacy (S&P 2006)*. IEEE Computer Society, 2006, pp. 264–279.

[10] U. Bayer, A. Moser, C. Krügel, and E. Kirda, "Dynamic analysis of malicious code," *J. Comput. Virol.*, vol. 2, no. 1, pp. 67–77, 2006.
[11] K. Natvig, "Sandbox technology inside AV scanners," in *Proc. 2001 Virus Bulletin Conf.* Virus Bulletin, Sep. 2001, pp. pp 475–487.
[12] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *IEEE Symp. Security and Privacy (S&P 2007)*. IEEE Computer Society, 2007, pp. 231–245.
[13] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proc. ACM SIGPLAN 2005 Conf. Programming Language Design and Implementation (PLDI 2005)*. ACM Press, 2005, pp. 213–223.
[14] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proc. 10th European Software Engineering Conf. / 13th ACM SIGSOFT Int'l Symp. Foundations of Software Engineering (ESEC/FSE 2005)*. ACM Press, 2005, pp. 263–272.
[15] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," in *Symp. Requirements Engineering for Information Security (SREIS 2001)*, 2001.
[16] P. Singh and A. Lakhotia, "Static verification of worm and virus behavior in binary executables using model checking," in *4th IEEE Information Assurance Workshop*. IEEE Computer Society, Jun. 2003.
[17] A. Lakhotia and P. Singh, "Challenges in getting 'formal' with viruses," *Virus Bulletin*, pp. 15–19, Sep. 2003.
[18] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *USENIX Security Symposium*. USENIX Association, Aug. 2003, pp. 169–186.
[19] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant, "Semantics-aware malware detection," in *IEEE Symp. Security and Privacy (S&P 2005)*. IEEE Computer Society, 2005, pp. 32–46.
[20] M. Dalla Preda, M. Christodorescu, S. Jha, and S. K. Debray, "A semantics-based approach to malware detection," in *34th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2007)*. ACM Press, 2007, pp. 377–388.
[21] E. Clarke and E. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*, ser. LNCS, vol. 131. Springer, 1981, pp. 52–71.
[22] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in CESAR." in *Symp. Programming*, ser. LNCS, vol. 137. Springer, 1982, pp. 337–351.
[23] E. Clarke, O. Grumberg, and D. Long, *Model Checking*. MIT Press, 1999.
[24] M. Oberhumer and L. Molnár. (2008) Ultimate Packer for eXecutables. [Online]. Available: http://upx.sourceforge.net/
[25] Xtreeme. (2008) Fast Small Good. [Online]. Available: http://www.xtreeme.prv.pl/
[26] M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith, "Software transformations to improve malware detection," *J. Comput. Virol.*, vol. 3, no. 4, pp. 253–265, Nov. 2007.
[27] Hex-Rays SA. (2008) IDA Pro. [Online]. Available: http://www.hex-rays.com/idapro/
[28] C. Linn and S. K. Debray, "Obfuscation of executable code to improve resistance to static disassembly." in *Proc. 10th ACM Conf. Computer and Communications Security (CCS 2003)*. ACM Press, 2003, pp. 290–299.
[29] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static disassembly of obfuscated binaries." in *USENIX Security Symposium*. USENIX Association, 2004, pp. 255–270.
[30] P. Ször, "Attacks on Win32," in *Proc. 2000 Virus Bulletin Conf.* Virus Bulletin, 2000, pp. pp 47–68.
[31] ——, *The Art of Computer Virus Research and Defense*. Addison Wesley, 2005, ch. 7, pp. 251–294.
[32] E. H. Spafford, "Computer viruses as artificial life." *Artificial Life*, vol. 1, no. 3, pp. 249–265, 1994.
[33] H. Hungar, O. Grumberg, and W. Damm, "What if model checking must be truly symbolic," in *Proc. IFIP WG 10.5 Advanced Research Working Conf. Correct Hardware Design and Verification Methods (CHARME 1995)*, ser. LNCS, vol. 987. Springer, 1995, pp. 1–20.
[34] J. Bohn, W. Damm, O. Grumberg, H. Hungar, and K. Laster, "First-order-CTL model checking," in *18th Conf. Foundations of Software Technology and Theoretical Compupter Science (FSTTCS 1998)*, ser. LNCS, vol. 1530. Springer, 1998, pp. 283–294.
[35] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proc. 20th Int'l Conf. Computer Aided Verification (CAV 2008)*, ser. LNCS, vol. 5123. Springer, 2008, pp. 423–427.
[36] A. Holzer, J. Kinder, and H. Veith, "Using verification technology to specify and detect malware," in *11th Int'l Conf. Computer Aided Systems Theory (EUROCAST 2007)*, ser. LNCS, vol. 4739. Springer, 2007, pp. 497–504.

Johannes Kinder is currently a doctoral student at Technische Universität Darmstadt, Germany. He holds a Master's level degree in computer science (Diplom-Informatiker) from Technische Unversität München. His research interests include software model checking, static analysis of low level code, software security, and malicious code detection.

Stefan Katzenbeisser received the PhD degree from the Vienna University of Technology, Austria. After working as a research scientist at Technische Universität München, Germany, and as a senior scientist at Philips Research, he joined the Technische Universität Darmstadt as assistant professor. His current research interests include Digital Rights Management, security aspects of digital watermarking, data privacy, software security and cryptographic protocol design. Currently, he is an associate editor of the IEEE Transactions on Dependable and Secure Computing and the EURASIP Journal on Information Security. He is a member of the IEEE, ACM and IACR.

Christian Schallhart works as a scientist at Technische Universität München, Germany. His general research interests are centered upon formal methods and their applications with the vision to contribute in improving daily software development practice. Currently he is pursuing research in test case generation, bounded model checking, and runtime verification. Prior to joining Technische Universität München, he worked as software architect in different domains ranging from the entertainment industry to financial systems. He obtained a PhD degree in computer science from the Vienna University of Technology.

Helmut Veith is the head of the Formal Methods in Systems Engineering Group at Technische Universität Darmstadt, and an adjunct professor at Carnegie Mellon University. He previously worked as a professor at Technische Universität München, as a visiting scientist at Carnegie Mellon, and as an assistant and associate professor at Vienna University of Technology. He has made research contributions to computer-aided verification, software engineering, computer security, complexity theory, and mathematical logic. Helmut Veith has a diploma in computational logic and a doctoral degree in computer science *sub auspiciis praesidentis* from Vienna University of Technology.