Detecting Malicious Code by Model Checking

Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, Helmut Veith

Technische Universität München
Institut für Informatik
D-85748 Garching bei München
{kinder,katzenbe,schallha,veith}@in.tum.de

Abstract. The ease of compiling malicious code from source code in higher programming languages has increased the volatility of malicious programs: The first appearance of a new worm in the wild is usually followed by modified versions in quick succession. As demonstrated by Christodorescu and Jha, however, classical detection software relies on static patterns, and is easily outsmarted. In this paper, we present a flexible method to detect malicious code patterns in executables by model checking. While model checking was originally developed to verify the correctness of systems against specifications, we argue that it lends itself equally well to the specification of malicious code patterns. To this end, we introduce the specification language CTPL (Computation Tree Predicate Logic) which extends the well-known logic CTL, and describe an efficient model checking algorithm. Our practical experiments demonstrate that we are able to detect a large number of worm variants with a single specification.

Key words: Model Checking, Malware Detection.

1 Introduction

Today's Internet connects a large number of household- and business-owned personal computers running variants of Microsoft's Windows operating system. As recent years have shown, these systems have been an especially attractive target for malicious individuals developing worms—programs that spread autonomously over networks requiring little or no user interaction, like *NetSky* or *Sasser*. Apart from 'classic' Internet worms which exploit vulnerabilities in network services, the most successful and widespread worms have been e-mail worms. This class of worms typically relies on users opening attachments to e-mails out of curiosity. Replicating with this rather primitive method, various versions of *NetSky*, *MyDoom* and *Bagle* have been dominating the worm hitlists for over a year.

In contrast to the viruses of the pre-Internet era, creating an e-mail worm that infects hundreds of thousands of computers nowadays does not require knowledge of systems or even assembly language programming. For example, *NetSky* and *MyDoom* were written in Visual C++, do not appear to be very skillfully engineered and contain obvious bugs in some of the versions. This trend is further intensified by the availability of virus toolkits which allow unskilled persons to create a new virus with a few mouse clicks.

During the last years it became evident that shortly after a new worm is released into the wild, several modified versions of the worm appear (either written by the same author or by individuals who somehow got hold of the source code). As a result of these developments, we see new worm derivatives appearing on the Internet almost every day. While these new versions differ only slightly from the original in terms of functionality, the resulting binary file can be quite different, depending on the compiler in use and its optimization settings; this problem worsens if *executable packers* such as UPX [15] or FSG [8] are used.

Current anti-virus products use rather straightforward (but yet computationally efficient) detection methods, most notably static signature matching and, more recently, dynamic analysis [13]. Static signature matching employs a database containing characteristic binary code sequences of known malware and matches these sequences against executables. Dynamic analysis executes the potentially infected programs in a controlled environment (sandbox) and checks for suspicious program behavior at runtime. These two approaches have the following two substantial drawbacks:

– Signature matching requires an up-to-date database of characteristic viral code sequences. In order to keep the false positives rate of the virus detector low, signatures are chosen so that one signature exactly matches one version of a virus or worm. In particular, the signature will thus not match against worm derivatives. This hypothesis was certified by Christodorescu and Jha in tests with commercially available virus scanners [3]; their tests showed that even naive modifications of the viral code, such as the insertion of a single `nop` instruction, can totally foil the detection process. Typically, modified worms spread quickly, which leads to a window of vulnerability between the release of a worm variant and the next update of the signature libraries. In this time span a novel virus or worm derivative cannot be detected by conventional anti-virus products. It would thus be highly desirable to have a virus scanner that reliably detects a virus or worm together with a large class of its potential derivatives.
– On the other hand, while dynamic analysis promises to solve some of the problems of static signature matching, it can be foiled by appropriate virus design. In particular the behavior of an executable is observed only over a limited timespan, which does not allow predictions of future malicious actions.

Semantic analysis methods (such as static analysis of executables) provide a possibility to overcome these two general problems. Consequently, various approaches for virus detection by formal methods can be found in the literature.

Bergeron et. al. [1] concentrate on the detection of suspicious system call sequences. In particular, they reduce the control flow graph of an executable to a subgraph containing only the nodes representing certain system calls and check whether the subgraph contains suspicious sequences of system calls. Singh and

Lakhotia [14] describe a system that uses the model checker SPIN to check properties of the control flow graph of a suspicious executable against a formula in linear temporal logic (LTL) specifying viral behavior. However, in [12] they express serious doubt about the feasibility of this method and generally of malicious code detection by formal analysis. In the paper closest to our work, Christodorescu and Jha [2] combat common virus obfuscation techniques by transforming virus source code into an malicious code automaton in order to handle inserted dead code and jumps between individual instructions; in addition they use unresolved symbols as placeholders for registers. If the language of the malicious code automaton has a non-empty intersection with the language of an automaton built from the program to be analyzed, then a viral code sequence is present in the program. In particular, their work is dedicated to cope with obfuscated malware.

In this paper, we propose a novel method to detect malicious code through model checking [5, 6]. Model checking has been successfully used in the past for the verification of both hardware and software. We disassemble a potentially infected executable and construct its control flow graph, containing nodes for all instructions that are present in the executable. We specify malicious behavior by a formula φ in a branching-time temporal logic. To this end, we introduce a new temporal logic CTPL (Computation Tree Predicate Logic) that is as expressive as CTL but allows a succinct and natural representation of malicious code patterns, taking register renaming into account. Finally, we introduce an explicit model checking algorithm for CTPL to verify the absence of malicious patterns in the code. More precisely, if the control flow graph of a program is a model for φ, then the program contains a malicious subroutine. With our prototype implementation we were able to detect several variants of the *NetSky*, *MyDoom* and *Klez* worms with *one single* CTPL formula.

In Section 2 we describe the specification logic CTPL in detail and give an example CTPL formula which describes common worm behavior. Section 3 introduces the model checking algorithm for CTPL and describes the model extraction from a binary file. Finally, we present preliminary results in Section 4.

2 The Specification Logic CTPL

In this section we describe the logic CTPL that we use to specify malicious behavior. Our logic needs to be able to express statements like "In the code there exists a `mov` instruction that loads the constant `937` into *some* register; later, the value contained in *this* register is always pushed onto the stack". In theory this can can be done in a temporal logic such as CTL [7]. For an introduction to temporal logics in the context of verification we refer to [5, 9].

We model the control flow graph of an executable as a Kripke structure, i.e., as a labeled finite graph. A Kripke structure M is a triple $\langle S, R, L\rangle$, where S is a set of states, $R \subseteq S \times S$ is a total transition relation, and $L : S \to 2^P$ is a labeling function that associates a set of propositions (elements of P) to each state. We say that a proposition p holds in a state s, if p is contained in the label of s, i.e., $p \in L(s)$. A path $\pi = s_0, s_1, s_2, \ldots$ in M is a sequence of states $s_i \in M$ with $(s_i, s_{i+1}) \in R$. For a path π, π^i refers to the state at position i, with π^0 being the starting state. Π_s is the set of all paths in M starting at state s.

CTL formulas allow to specify temporal properties of Kripke structures by six special temporal operators $\mathbf{A}, \mathbf{E}, \mathbf{X}, \mathbf{F}, \mathbf{G}, \mathbf{U}$; $\mathbf{A}$ 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. $\mathbf{A}\,\varphi$ is true in a state s if for all paths in Π_s, φ is true; in contrast, $\mathbf{E}\,\varphi$ is true in 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 state 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 $\mathbf{AX}, \mathbf{EX}, \mathbf{AU}, \mathbf{EU}, \mathbf{AF}, \mathbf{EF}, \mathbf{AG}, \mathbf{EG}$). While CTL requires basic knowledge of logic, it can be quickly learned and has been used successfully in order to specify properties of hardware and software.

The example at the beginning of this section can be expressed in CTL as a large formula, containing clauses for all register names:

$$
\begin{aligned}
&\mathbf{EF}(\texttt{mov eax,937} \wedge \mathbf{AF}(\texttt{push eax})) \vee \\
&\mathbf{EF}(\texttt{mov ebx,937} \wedge \mathbf{AF}(\texttt{push ebx})) \vee \\
&\mathbf{EF}(\texttt{mov ecx,937} \wedge \mathbf{AF}(\texttt{push ecx})) \vee \\
&\ldots
\end{aligned}
$$

Here the machine instructions are atomic propositions (i.e., elements of P). This formula essentially expresses that there exists a path in the control flow graph of the executable that contains a `mov` instruction, which is followed later (on every possible computation path) by a corresponding `push` instruction.

In this notation, formulas that model potentially malicious behavior tend to be very large. Typically these formulas must be resistant against register renaming; however, this can only be handled in CTL by explicitly mentioning each possible register assignment in the formula (as shown in the example above). In order to keep the size of the formula small, we introduce an extension of CTL—called CTPL—which is tailored towards the specification of code patterns. While CTPL is not more expressive than CTL, specialized model-checking algorithms can efficiently exploit the more concise representation of CTPL formulas.

In CTPL we allow propositions to be *predicates* of the form $p(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ either represent free variables or constants; each free variable x_i can take on values from a finite set $\mathcal{U}$ called universe. In CTPL model checking, the set of propositions P is the set of all syntactic terms $p(c_1, \ldots, c_n)$, where $c_1, \ldots, c_n$ are elements of $\mathcal{U}$. In our application, the predicate names represent assembler instructions in the natural way, e.g., `cmp ebx,[bp-4]` is represented as `cmp`(`ebx`, `[bp-4]`). In addition, we introduce quantifiers $\exists$ and $\forall$ that allow to quantify over free variables in a predicate. For example, the above CTL formula

1.	$M, s \models \psi$	$\Leftrightarrow$ There is a $\mathcal{B}$ such that $M, s \models_{\mathcal{B}} \psi$.
2.	$M, s \models_{\mathcal{B}} p(x_1, \ldots, x_n)$	$\Leftrightarrow p(\mathcal{B}(x_1), \ldots, \mathcal{B}(x_n)) \in L(s)$.
3.	$M, s \models_{\mathcal{B}} \neg\psi$	$\Leftrightarrow M, s \models_{\mathcal{B}} \psi$ does not hold.
4.	$M, s \models_{\mathcal{B}} \psi_1 \vee \psi_2$	$\Leftrightarrow M, s \models_{\mathcal{B}} \psi_1$ or $M, s \models_{\mathcal{B}} \psi_2$.
5.	$M, s \models_{\mathcal{B}} \psi_1 \wedge \psi_2$	$\Leftrightarrow M, s \models_{\mathcal{B}} \psi_1$ and $M, s \models_{\mathcal{B}} \psi_2$.
6.	$M, s \models_{\mathcal{B}} \forall x\, \psi$	$\Leftrightarrow$ For all $a \in \mathcal{U}$, $M, s \models_{\mathcal{B}[x \mapsto a]} \psi$.
7.	$M, s \models_{\mathcal{B}} \exists x\, \psi$	$\Leftrightarrow$ For some $a \in \mathcal{U}$, $M, s \models_{\mathcal{B}[x \mapsto a]} \psi$.
8.	$M, s \models_{\mathcal{B}} \mathbf{EF}\, \psi$	$\Leftrightarrow$ There is a path π from s containing a state $s_i \in \pi$ such that $M, s_i \models_{\mathcal{B}} \psi$.
9.	$M, s \models_{\mathcal{B}} \mathbf{EG}\, \psi$	$\Leftrightarrow$ There is a path π from s such that $M, s_i \models_{\mathcal{B}} \psi$ for all states $s_i \in \pi$.
10.	$M, s \models_{\mathcal{B}} \mathbf{EX}\, \psi$	$\Leftrightarrow$ There is a successor state s_1 of s such that $M, s_1 \models_{\mathcal{B}} \psi$.
11.	$M, s \models_{\mathcal{B}} \mathbf{E}\, [\psi_1 \mathbf{U} \psi_2]$	$\Leftrightarrow$ For a path $\pi = (s_0, s_1, \ldots)$ where $s = s_0$ there is a $k \geq 0$ such that $M, s_i \models_{\mathcal{B}} \psi_1$ for all $i < k$ and $M, s_j \models_{\mathcal{B}} \psi_2$ for all $j \geq k$.
12.	$M, s \models_{\mathcal{B}} \mathbf{AF}\, \psi$	$\Leftrightarrow$ Every path π from s contains a state $s_i \in \pi$ such that $M, s_i \models_{\mathcal{B}} \psi$.
13.	$M, s \models_{\mathcal{B}} \mathbf{AG}\, \psi$	$\Leftrightarrow$ On every path π from s, there holds $M, s_i \models_{\mathcal{B}} \psi$ in all states $s_i \in \pi$.
14.	$M, s \models_{\mathcal{B}} \mathbf{AX}\, \psi$	$\Leftrightarrow$ For all successor states s_1 of s, $M, s_1 \models_{\mathcal{B}} \psi$.
15.	$M, s \models_{\mathcal{B}} \mathbf{A}\, [\psi_1\, \mathbf{U}\, \psi_2]$	$\Leftrightarrow$ For all paths $\pi = (s_0, s_1, \ldots)$ where $s = s_0$ there is a $k \geq 0$ such that $M, s_i \models_{\mathcal{B}} \psi_1$ for all $i < k$ and $M, s_j \models_{\mathcal{B}} \psi_2$ for all $j \geq k$.

Fig. 1. Semantics of the logic CTPL.

could be expressed succinctly in CTPL as

$$\exists r \mathbf{EF}(\texttt{mov}(r, \texttt{937}) \wedge \mathbf{AF}(\texttt{push}(r))).$$

Syntax and Semantics of CTPL. The syntax of CTPL is the same as the syntax of CTL with the following addition: if φ is a CTPL formula with a free variable x, then both $\forall x\, \varphi$ and $\exists x\, \varphi$ are CTPL formulas. Similar as in the semantics definition of first order logic, we collect bindings for free variables (i.e., assignments between variable names and values from the universe $\mathcal{U}$) in a set $\mathcal{B}$, called environment. $\mathcal{B}[x \mapsto a]$ represents the environment that maps the variable x to a and every other variable y to $\mathcal{B}(y)$. If a formula φ is valid in a state s of a Kripke structure under environment $\mathcal{B}$, we will write $M, s \models_{\mathcal{B}} \varphi$. The detailed definition of the semantics is given in Figure 1. A formula φ is valid in M (written $M \models \varphi$), if $M, s_0 \models \varphi$ for the initial state s_0.

Modeling the behavior of programs in CTPL. As the following examples show, CTPL allows much flexibility in specifying program behavior:

– Code that sets a register to 0 and pushes this value onto the stack with the next instruction can be specified as

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

– By replacing **EX** with **EF**, we can specify a code sequence where other instructions can occur between `mov` and `push`:

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

Note that this specification also allows the presence of instructions between `mov` and `push` that modify the contents of the register r.

– If we want to disallow any change of the register r with a `mov` instruction between the first `mov` and `push`, we can formulate this constraint using **EU**:

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

Of course there are other ways to change the contents of register r, but for simplicity, only `mov` is forbidden here. A similar construction can always be used if the contents of a register must be preserved between two non-consecutive instructions.

If a code fragment calls a function with more than one parameter, multiple `push` instructions will be present before a `call`, pushing the parameters of the function onto the stack. Each `push` will in turn be preceded by other instructions that compute the values of the parameters. CTPL can be used to specify the behavior of such code fragments even in the presence of arbitrarily scheduled independent instructions by enforcing the correct computation of the parameter values and the correct stack layout. In particular, we model this behavior in CTPL by the conjunction of several different subformulas. One subformula represents the order in which the function parameters are pushed onto the stack, while the other subformulas enforce the correct computation of the individual parameter values. In order to tie these subformulas together, we introduce a special location predicate $\#\mathrm{loc}(L)$; each node of the Kripke structure is labeled with a unique number L.

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

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

The first line of the formula expresses that there exists a `mov` instruction in the code that clears a register r_1; at a later time we find an instruction at location L, whose form will be specified later. The second line asserts that we can eventually find a `call` to function `func` that is preceded by a `push` instruction at location L, which in turn is preceded by another `push` instruction that pushes the content of r_2 onto the stack (for simplicity, we have omitted subformulas that ensure integrity of the registers r_1 and r_2 between the `mov` and `push` instructions).

Modeling viral behavior in CTPL. Figure 2 shows a part of the disassembled infection routine of the worm *Klez.h*. It exhibits the typical behavior of e-mail

```
mov     edi, [ebp+arg_0]
xor     ebx, ebx                       clear register ebx
push    edi
        ⋮
lea     eax, [ebp+ExFileName]          store address of the string buffer in eax
mov     [esp+65Ch+var_65C], 104h
push    eax                            push the address of the string buffer
push    ebx                            set first system call argument to NULL
call    ds:GetModuleFileNameA          call GetModuleFileNameA
lea     eax, [ebp+NewFileName]         load address of destination file name
push    ebx                            set third argument to zero
push    eax                            push the address of destination name
lea     eax, [ebp+ExFileName]          fetch address of source name string
push    eax                            push the address as first argument
call    ds:CopyFileA                   call CopyFileA
```

Fig. 2. Code fragment of the infection routine of *Klez.h.*

worms: the code determines the name of its own executable using the Windows API call `GetModuleFileNameA` and then copies this file to a different location (usually a system directory or a shared folder) with the system call `CopyFileA`. The Windows API function `GetModuleFileNameA` takes three parameters, namely a module name and the address and size of the destination string buffer; if the module file name is set to zero (NULL), it returns the name of the running process. The system call `CopyFileA` also takes three parameters: addresses of the strings specifying source and destination files and a Boolean flag. The code in Figure 2 basically consists of those two system calls and instructions that initialize the parameters (the relevant lines of the code fragment are explained in the figure).

Figure 3 shows a CTPL formula that specifies this typical worm behavior; the formula matches code that calls `GetModuleFileNameA` to retrieve its own filename, and afterwards uses the resulting string as a parameter to the system call `CopyFileA`. The formula consists of six subformulas that are tied together with the location predicate and describe the correct computation of the system call arguments. Line 3 specifies that a string buffer pointer is stored in a register r_0; line 4 asserts that a register r_1 is set to zero (NULL). Using the data integrity construction described before, both subformulas assure that these register values are not changed by `mov` or `lea` instructions until the arguments are pushed onto the stack with instructions located at positions L_0 and L_1. Lines 5-8 specify the preparation of the stack and the call to `GetModuleFileNameA`: before invoking the `call` instruction (located at L_m), a constant c_0 and the contents of the previously prepared registers r_0 and r_1 are pushed onto the stack; the latter two `push` instructions occur at locations L_0 and L_1. In addition, we specify (again with the above mentioned data integrity construction) that the stack remains intact until the system call is issued (i.e., no other stack operations occur). Lines

$$
\begin{array}{ll}
1. & \exists L_m \exists L_c \exists v_{File}(\\
2. & \quad \exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0(\\
3. & \quad\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_0, t) \vee \mathtt{lea}(r_0, t)))\mathbf{U}\#\mathrm{loc}(L_0)) \wedge \\
4. & \quad\quad \mathbf{EF}(\mathtt{mov}(r_1, \mathtt{0}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_1, t) \vee \mathtt{lea}(r_1, t)))\mathbf{U}\#\mathrm{loc}(L_1)) \wedge \\
5. & \quad\quad \mathbf{EF}(\mathtt{push}(c_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
6. & \quad\quad\quad \mathbf{U}(\mathtt{push}(r_0) \wedge \#\mathrm{loc}(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
7. & \quad\quad\quad\quad \mathbf{U}(\mathtt{push}(r_1) \wedge \#\mathrm{loc}(L_1) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
8. & \quad\quad\quad\quad\quad \mathbf{U}(\mathtt{call}(\mathtt{GetModuleFileNameA}) \wedge \#\mathrm{loc}(L_m))))) \\
9. & \quad\quad) \\
10. & \quad \wedge(\exists r_0 \exists L_0(\\
11. & \quad\quad \mathbf{EF}(\mathtt{lea}(r_0, v_{File}) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{mov}(r_0, t) \vee \mathtt{lea}(r_0, t)))\mathbf{U}\#\mathrm{loc}(L_0)) \wedge \\
12. & \quad\quad \mathbf{EF}(\mathtt{push}(r_0) \wedge \#\mathrm{loc}(L_0) \wedge \mathbf{EX}\,\mathbf{E}(\neg\exists t(\mathtt{push}(t) \vee \mathtt{pop}(t))) \\
13. & \quad\quad\quad \mathbf{U}(\mathtt{call}(\mathtt{CopyFileA}) \wedge \#\mathrm{loc}(L_c))) \\
14. & \quad\quad)) \\
15. & \quad \wedge\mathbf{EF}(\#\mathrm{loc}(L_m) \wedge \mathbf{EF}\#\mathrm{loc}(L_c)) \\
16. &)
\end{array}
$$

Fig. 3. CTPL formula that matches code creating copies of its own executable.

11-14 specify in a similar manner the preparation of parameters for and the invocation of the system call `CopyFileA`, occurring at location L_c. Finally, line 15 asserts that `GetModuleFileNameA` must be invoked before `CopyFileA`, i.e., the location L_m occurs before L_c. All locations are existentially quantified. It is possible (in a similar way as described above), to construct formulas in CTPL that capture the basic functionality of various types of malicious code.

3 Model Checking Executable Files

In order to model check a program, it is necessary to represent it as a Kripke structure; we do this by extracting its control flow graph. In order to perform fine-grained specifications, every instruction in the program is represented as a node in the graph. Every instruction that is not a (conditional or unconditional) jump is linked to its immediate successor. An unconditional jump (`jmp`) is linked to the jump target. Nodes containing conditional jumps, such as `jz` or `jge`, are linked to both their successor and the jump target, i.e., are modeled as nondeterministic choices in the Kripke structure.

In general, there are two ways to handle procedure calls (`call`): either one builds a separate model for each procedure in the executable or one inlines (non-recursive) subroutines into one single Kripke structure. In our current prototype we follow the first approach.

Each node in the Kripke structure is labeled by a unique location number L (stored as predicate $\#\mathrm{loc}(L)$) and by the assembler instruction, represented as predicate $\mathrm{instr}(\mathit{param}_1, \ldots, \mathit{param}_n)$. Here, instr codes the name of the machine instruction (such as `mov`, `jz` or `lea`) and param_i denote its parameters (see Figure 4). These parameters are always constants representing register names, memory

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

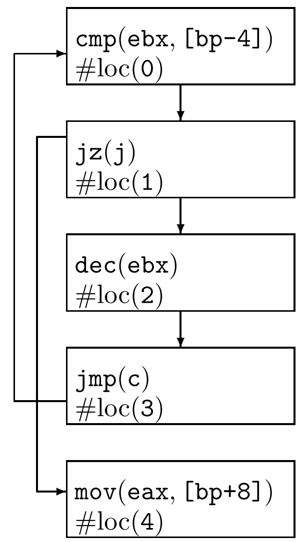

Fig. 4. Executable code sequence and corresponding Kripke structure.

locations or integer operands of the original instruction. Note that the universe $\mathcal{U}$ of parameters is always finite for a fixed disassembled executable.

Model Checking CTPL. The algorithm to check whether a Kripke structure M is a model of a CTPL formula φ extends the classic explicit model checking algorithm [4], which uses a form of dynamic programming. In particular, our algorithm visits the states of the Kripke structure as often as the classical algorithm, but needs to keep track of the variable bindings which might become exponentially large in the worst case. However, our experiments have demonstrated that this is not a performance bottleneck in practice.

It can be shown that the model checking problem for CTPL is **PSPACE**-complete; the hardness follows by a reduction from QBF, whereas membership can be seen through a variant of the model checking algorithm that uses backtracking and does not keep track of all possible bindings. The complexity of model checking CTPL formulas is thus comparable to the complexity of the model checking problem for LTL. However, **PSPACE**-completeness tells little about the practical performance. In particular, the construction in the **PSPACE**-hardness proof requires an unbounded number of quantifiers ($\forall, \exists$), a situation that will not happen in practice.

The model checking algorithm traverses the formula φ in a bottom-up manner, computing for each state s of the Kripke structure and each subformula φ' of φ, whether φ' holds in s. This information is stored in a labeling relation $L \subseteq (S \times F \times B)$ with S, F, and B being the set of states, the set of CTPL formulas, and the set of bindings, respectively. In particular, a tuple $(s, \varphi', \mathcal{B})$ is stored in L, if the subformula φ' holds in state s with respect to the variable binding $\mathcal{B}$. The model checker uses these labels to recursively evaluate more

$f = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$:

1. **for all** states s
2.   **if** $(s, \psi_2, C_2) \in L$ **then** $L := L \cup (s, f, C_2)$;
3. **while** L has changed **do**
4.   **for all** states s **if** $\exists C_s (s, f, C_s) \in L$ **then**
5.   **for all** $(p, s) \in R$ *// for all parents of s*
6.   **if** $\exists C_1 (p, \psi_1, C_1) \in L$ **then**
7. $C_0 := C_s \wedge C_1$;
8.   **if** $C_0 \not\equiv \perp$ **then**
9.   **if** $\exists C_p (p, f, C_p) \in L$ **then** $L := L \cup (p, f, C_0 \vee C_p)$;
10.   **else** $L := L \cup (p, f, C_0)$;

Fig. 5. Part of the model checking algorithm handling formulas of type $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$.

complicated subformulas of φ; this procedure is iterated up to the full formula φ. $M \models \varphi$ holds if the initial state s_0 of M is finally labeled with φ. For efficiency reasons, the bindings will be represented in the labeling relation as a Boolean formula C; this formalism allows efficient computation of negated bindings. The Boolean formula representing $\mathcal{B}$ will be denoted by C.

Figure 5 shows a typical part of the model checking algorithm, namely the labeling algorithm for a subformula starting with **EU**; the full model checking algorithm can be found in the appendix. In order to find all states where $f = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ holds, the algorithm assumes that all states where ψ_1 or ψ_2 hold are already labeled with ψ_1 or ψ_2. If there exists a state that is labeled with ψ_2, $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ holds in this state and we can label it with f (line 2). If such a state exists, we iteratively search for all predecessor states p of s; if these states are already labeled with ψ_1, then we can label these states also with f (again because $f = \mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ surely holds there). The algorithm continues until the label set does not change any more (lines 4-10). During the process, the bindings are updated accordingly; in particular the bindings C_s of node s are propagated to all parental nodes (line 7). It can be shown that this process terminates and labels all states where $\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$ is valid. In a similar manner, all other subformula types can be treated (see appendix).

4 Results and Future Work

We have implemented a prototype of the CTPL model checker in Java; the program takes an assembler file and a CTPL specification as input. In order to model check an executable, we first disassemble the executable file with Datarescue's IDAPro [10]. However, most e-mail worms use executable packers—tools that compress an executable and prepend an extraction routine that will decompress the binary into system memory every time the resulting executable is run. This makes it necessary as a first step to uncompress the executable in order to obtain its original code. Currently this process is done manually, but it can be automated. The complete toolchain of our prototype is depicted in Figure 6.

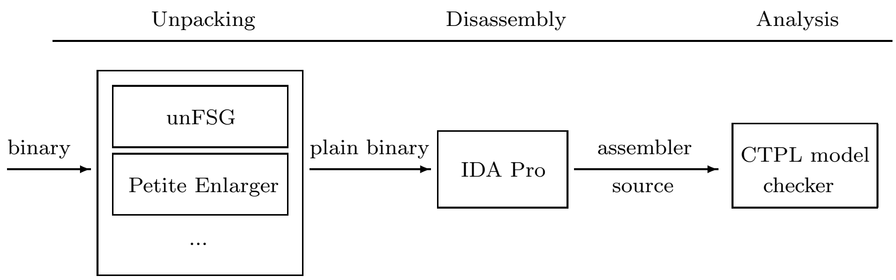

Fig. 6. Toolchain of our prototype.

We have tested our prototype on a set of worms dating from the years 2002–2004, provided by Ikarus Software [11]. Even though there are quite large differences in the compiled binary code between the different versions of one worm, our CTPL specification matched most of the worm derivatives. During our experiments, we even found that carefully written CTPL specifications can apply to several families of worms. Using a slightly more general CTPL formula than the one shown in Figure 3, we were able to prove the malicious behavior of *Klez.a*, *Klez.e*, *Klez.g*, *Klez.h*, *NetSky.b*, *NetSky.c*, *NetSky.d*, *NetSky.e*, *NetSky.p*, *MyDoom.a*, *MyDoom.i*, *MyDoom.m*, and *MyDoom.aa* with this single formula.

With the current prototype, checking a procedure of 150 lines of assembler code takes about 2 seconds on an Athlon XP 2600+ CPU with 512MB of RAM. The prototype implementation of the model checker is not optimized with respect to computation time. We can speed up this model checking algorithm significantly, e.g., by using sophisticated data structures (like Ordered Binary Decision Diagrams) for representing the binding sets. Moreover, simple and fast preprocessing of the assembler input files can eliminate procedures which obviously do not match the specification.

As future work, we see several promising approaches to improve expressive power, performance and usability of our prototype. By replacing the x86 instruction predicates with abstracted forms that capture their operational semantics we can decrease the complexity of CTPL formulas. For example, clearing a register can be abstracted to assign$(r, 0)$, regardless of its concrete implementation (e.g., as `xor eax,eax` or `mov eax,0`). Using such abstractions, more accurate data integrity constructions of the form $\mathbf{E}(\neg\exists t\, \texttt{assign}(r,t))\, \mathbf{U}\, \psi$ can be specified. In addition, as there are several typical construction patterns in specifications, we will provide a macro language that allows the user to write malicious code specifications in a more abstract notation. We also plan to investigate how the performance of the model checking algorithm can be improved by the use of efficient data structures.

5 Conclusions

In this paper, we proposed a novel approach to detect malicious patterns in executable code sequences by model checking. In particular, the behavior of a malicious code sequence is modeled as a formula in a branching time temporal logic called CTPL; this formula is matched against the control flow graph of an executable program by a model checker. CTPL allows for a succinct but yet natural way to specify the behavior of a code fragment.

Using this approach, we were able to write CTPL specifications that capture common mechanisms present in viruses and worms. In particular, we were able to use one CTPL formula to classify several worms together with their derivatives as malicious. The practical results obtained show that CTPL model checking is a promising approach for systematically and reliably detecting computer worms together with functionally similar (but syntactically obfuscated) derivatives.

Acknowledgements. We thank Ikarus Software and Christopher Krügel for their kind support of this project.

References

1. J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y.Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *Symposium on Requirements Engineering for Information Security*, March 2001.
2. M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium (Security'03)*, pages 169–186. USENIX Association, August 2003.
3. M. Christodorescu and S. Jha. Testing malware detectors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'04)*, 2004.
4. E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer, 1981.
5. E. Clarke, O. Grumberg, and D. Long. *Model Checking*. MIT Press, 1999.
6. E. Clarke and B. Schlingloff. *Handbook of Automated Reasoning*, chapter Model Checking, pages 1637–1790. Elsevier, 2001.
7. E. Emerson. *Handbook of Theoretical Computer Science, volume B*, chapter Temporal and Modal Logic, pages 995–1072. Elsevier, 1990.
8. Fast Small Good. `http://www.xtreeme.prv.pl/`. (Last accessed: 16 Dec. 2004).
9. M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.
10. IDA Pro. `http://www.datarescue.com/idabase/`. (Last accessed: 20 Jan. 2004).
11. IKARUS Software. `http://www.ikarus-software.at/`. (Last accessed: 20 Jan. 2004).
12. A. Lakhotia and P. Singh. Challenges in getting 'formal' with viruses. *Virus Bulletin*, September 2003.
13. Norman ASA. Norman sandbox whitepaper. Technical report, 2003.
14. P. Singh and A. Lakhotia. Static verification of worm and virus behavior in binary executables using model checking. In *4th IEEE Information Assurance Workshop*, June 2003.
15. Ultimate Packer for eXecutables. `http://upx.sourceforge.net/`. (Last accessed: 16 Dec. 2004).

Appendix: Model Checker for CTPL

In the appendix, we present our model checking algorithm for CTPL formulas; all temporal operators of CTPL can be reduced to **EU**, **EX**, and **AF** using standard formula rewrite rules [9]. Thus we only have to treat these three temporal operators. Moreover, we rewrite $\forall x \psi$ as $\neg \exists x \neg \psi$.

Input: a Kripke structure M and a closed CTPL formula F
Output: set of states in M which satisfy F

The constraint sets are always kept in DNF, such that:
$atom : (variable\ [= \mid \neq]\ \texttt{constant})$
$\mathcal{B} :\quad \{atom_1 \wedge \ldots \wedge atom_n\}$
$C :\quad \{\mathcal{B}_1 \vee \ldots \vee \mathcal{B}_m\}$

```
for all subformulas f of formula F in ascending order of size
    case f of
    ⊥:
        label no states;
    p(x_1, ..., x_n):
        stateIteration:  for all states s
            if ∃c_1, ..., c_n (s, p(c_1, ..., c_n), ⊤) ∈ L then
                B := ⊤;
                for i := 1 to n
                    if x_i is a variable then  B := B ∧ (x_i = c_i);
                    else  if x_i ≠ c_i then  continue stateIteration;
                if B ≢ ⊥ then  L := L ∪ (s, f, B);
    ∃x (ψ):
        for all states s
            if ∃C(s, ψ, C) ∈ L then
                C_0 := ⊥;
                for all B ∈ C
                    B_0 := ⊤;
                    for all (v [= | ≠] c) ∈ B
                        if v ≠ x then  B_0 := B_0 ∧ (v [= | ≠] c) ;
                    C_0 := C_0 ∨ B_0;
                L := L ∪ (s, f, C_0);
    ¬ψ :
        for all states s
            if ∃C(s, ψ, C) ∈ L then
                if ¬C ≢ ⊥ then  L := L ∪ (s, f, ¬C);
            else L := L ∪ (s, ψ, ⊤);
    ψ_1 ∧ ψ_2:
        for all states s
            if ∃C_1(s, ψ_1, C_1) ∈ L  and ∃C_2(s, ψ_2, C_2) ∈ L then
                if C_1 ∧ C_2 ≢ ⊥ then  L := L ∪ (s, f, C_1 ∧ C_2);
```

```
ψ1 ∨ ψ2:
    for all states s
        if ∃C1(s, ψ1, C1) ∈ L then  C0 := C1  else C0 := ⊥;
        if ∃C2(s, ψ2, C2) ∈ L then  C0 := C0 ∨ C2;
        if C0 ≢ ⊥ then  L := L ∪ (s, f, C0);
E[ψ1 U ψ2]:
    for all states s
        if ∃C2(s, ψ2, C2) ∈ L then  L := L ∪ (s, f, C2);
    while L has changed do
        for all states s  if ∃Cs(s, f, Cs) ∈ L then
            for all (p, s) ∈ R            // for all parents of s
                if ∃C1(p, ψ1, C1) ∈ L then
                    C0 := Cs ∧ C1;
                    if C0 ≢ ⊥ then
                        if ∃Cp(p, f, Cp) ∈ L then  L := L ∪ (p, f, C0 ∨ Cp);
                        else L := L ∪ (p, f, C0);
EXψ:
    for all states s
        if ∃Cs(s, ψ, Cs) ∈ L then   for all (p, s) ∈ R
            if ∃Cp(p, f, Cp) ∈ L then  L := L ∪ (p, f, Cs ∨ Cp);
            else L := L ∪ (p, f, Cs);
AF ψ:
    for all states s
        if ∃Cψ(s, ψ, Cψ) ∈ L then  L := L ∪ (s, f, Cψ);
    while L has changed do
        stateIteration:  for all states s
            C0 := ⊤;
            for all (s, c) ∈ R            // for all children of s
                if ∃Cc(c, f, Cc) ∈ L then  C0 := C0 ∧ Cc;
                else continue stateIteration;
                if C0 ≡ ⊥ then  continue stateIteration;
            L := L ∪ (s, f, C0);

end for

output all states s with (s, F, C) ∈ L for some C.
```