Using Verification Technology to Specify and Detect Malware

Andreas Holzer, Johannes Kinder, and Helmut Veith

Technische Universität München
Fakultät für Informatik
85748 Garching, Germany
{holzera,kinder,veith}@in.tum.de

Abstract. Computer viruses and worms are major threats for our computer infrastructure, and thus, for economy and society at large. Recent work has demonstrated that a model checking based approach to malware detection can capture the semantics of security exploits more accurately than traditional approaches, and consequently achieve higher detection rates. In this approach, malicious behavior is formalized using the expressive specification language CTPL based on classic CTL. This paper gives an overview of our toolchain for malware detection and presents our new system for computer assisted generation of malicious code specifications.

1 Introduction

In the last twenty-five years, model checking has evolved into an industrial-strength framework for the verification of hardware and software. Traditionally, the model checking tool chain assumes that the specifications describe the crucial properties of the system to be analyzed in a positive way, i.e., specifications describe the intended behavior of the system. Our recent approach to malware detection [1] inverts this picture, in that we use specifications to describe malicious behavior. We employ an extension of the temporal logic CTL to specify malicious behavior, and extract a finite state model from the disassembled executable. If the model checker finds out that a specification holds true, then the malware detector reports that the analyzed code is infected. The advantage of our approach over classical malware detection tools is our ability to cover families of malware which use the same attack principle. Our tool is able to detect also previously unknown variants of malware which exhibit behavior similar to that of known malware, but are syntactically different. Classical malware detectors mainly rely on variations of pattern matching using malware signatures from a virus database [2, 3]. Thus, they require an update of the virus databases to detect new malware variants.

Malware specifications differ from “standard” software specifications in crucial aspects. Most importantly, a software specification is usually written in the context of the program to be analyzed, i.e., the specification is created with the assistance of the programmer. Variable names, labels, and constant values are often specific to a program; using them in a specification thus requires an understanding of the program. In the typical malware detection scenario, however, we have only little or no knowledge about the program. We usually do not have access to the source code but only to the compiled binary or byte code of the software. When the program is indeed malicious, it is very unlikely for the programmer to have created the software in an analysis-friendly way. This scenario creates numerous difficulties specific to malware analysis. First, we need to prepare the program to be analyzed in a suitable manner such that we can extract an abstract model from it. Since the program is in binary form this requires disassembly and, for some files, a decryption mechanism similar to that found in commercial anti-virus tools. Second, the malicious code specification has to be applicable to general programs, that is, it must not contain hard coded variable names. Once the first problem has been successfully addressed, i.e., once a candidate for checking has been disassembled, one has to choose a strategy for extracting the abstract model. This choice heavily influences the nature of specifications that later can be verified against the model. The disassembled binary by itself contains only very low level semantic information about the program. Basically, there are two possible strategies for creating a model from the disassembled program and reasoning about its semantics.

– The first option is to perform extensive preanalysis and to try to extract exact semantics from the assembly code. Specifications for such a model then could be relatively short functional descriptions of malicious behavior. The huge drawback of this method is, however, that the preanalysis requires exact and complete semantics for assembly code. The low level nature of x86 assembly makes an exact functional description infeasible with current technology.
– The second option, which we pursued in our approach, is to implement a coarse abstraction that uses the control flow graph of the program as model and ignores machine state other than the program counter. The resulting model then is a state transition system with one assembly instruction per state. With this approach, specifications become more complex as they need to reflect a lower level of behavior; the need to have abstract variable names and values in specifications is immanent.

Therefore we enriched CTL by variables and quantifiers and obtained a new specification logic CTPL [1]. The advantages of this extension can be easily illustrated by the example specification “there is a register that is first set to zero and later pushed onto the stack”, which is on the level of assembly code but abstracts implementation details irrelevant to the malicious behavior. If we try to formalize this specification in CTL, this would result in a large disjunction of the following form:

$$\begin{aligned}&\mathbf{EF}((\texttt{mov eax,0}) \wedge \mathbf{AF}(\texttt{push eax})) \vee\\ &\mathbf{EF}((\texttt{mov ebx,0}) \wedge \mathbf{AF}(\texttt{push ebx})) \vee\\ &\mathbf{EF}((\texttt{mov ecx,0}) \wedge \mathbf{AF}(\texttt{push ecx}))\ \vee \ldots\end{aligned}$$

CTPL, however, uses predicates rather than atomic propositions to represent assembler instructions, which allows to quantify over an instruction’s parameters. In CTPL, we can express the same specification using quantifiers as

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

Despite the succinct representation CTPL offers, the design of malicious code specifications is a fairly tedious process which involves writing similarly structured formulas

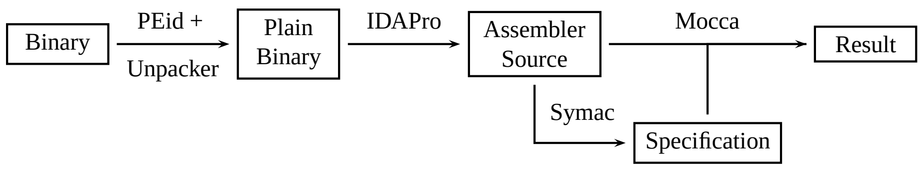

Fig. 1. Malicious Software Detection Process.

several times. Therefore, we augmented the specification design phase by implementing *Symac* (**S**pecification s**y**nthesis for **ma**licious **c**ode), a visual editing tool that aids a user in extracting a specification from a representative malware sample. The editor encapsulates many common patterns and provides support for future automated extraction techniques. In this paper, we first give an overview of our malware detection architecture and then proceed to present our new tool for the creation of malicious code specifications in CTPL.

2 Malware Specification and Detection

Figure 1 depicts our complete tool chain for malicious code detection. Our CTPL model checker *Mocca* expects plain text assembly source code as input to construct the internal model representation, so there is some amount of preprocessing necessary when a new executable is to be checked. The majority of malware is *packed*, i.e., encoded using an executable packer, which at runtime decrypts the program into memory. A packed program is practically immune to static analysis and needs to be decrypted before proceeding with the analysis. Thus, as a first step, we have to determine whether the program is packed and which packing mechanism has been used. It is possible to detect packed files by measuring byte entropy or by looking for known patterns generated by common executable packers. For this step, we resort to PEiD [4], a widely used tool for identifying packed files. A number of specialized unpacking programs and libraries are freely available, so knowing which packer was used to protect the program, the corresponding unpacking tool can be chosen to correctly decrypt the executable in the second step. For unknown packers, we can use generic, emulation-based unpacking methods [5].

After unpacking, the resulting plain binary can be passed to a disassembler. We use Datarescue's state-of-the-art disassembler IDAPro [6] for this task, which generates the assembly source code used as input to the Mocca model checker. Mocca creates an abstract model of the executable by parsing the assembly file. During parsing, it performs some simple syntactical substitutions to disambiguate the assembly code (such as replacing `xor eax,eax` with `mov eax,0`). We model assembly code syntactically as Kripke structures, as illustrated in Figure 2. Every instruction is represented by a corresponding predicate, its parameters are treated as constants. Each line of code corresponds to a state in the Kripke structure that is uniquely identified by a so called *location* modeled by the special predicate $\sharp loc$. Transitions in the Kripke structure are added according to the possible control flow of the code: Instructions without successors (e.g. return statements in intraprocedural analysis) are assigned with a self-loop.

```
1         mov  ebx ,  eax
2         jz   l1
3         mov  ecx ,  ebx
4   l1 :  mov  ebx ,  edx
```

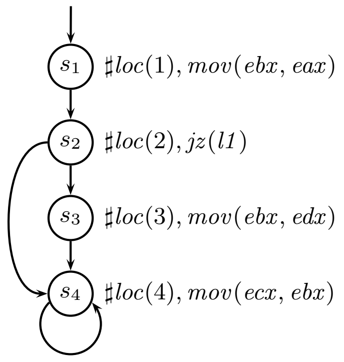
s_1 $\sharp loc(1), mov(ebx, eax)$
s_2 $\sharp loc(2), jz(l1)$
s_3 $\sharp loc(3), mov(ebx, edx)$
s_4 $\sharp loc(4), mov(ecx, ebx)$

Fig. 2. Example of assembler code and its corresponding Kripke structure.

Jumps are connected to their target only, conditional jumps to both possible successors. All other instructions are given a fall-through edge to their successor in the code.

Finally, Mocca checks the model against a malicious code specification in CTPL. Since the specification logic allows quantification, we needed to adapt the bottom-up explicit model checking algorithm for CTL [7] to keep track of possible variable assignments. The introduction of quantifiers causes the CTPL model checking problem to become **PSPACE**-complete [8]. Therefore, the model checker uses several optimizations to reduce the number of procedures checked and to keep the number of tracked variable assignments low. Finally, the model checker reports whether the assembly file satisfies the specification, i.e., whether it is malicious or not.

3 Computer Aided Specification Synthesis

Malware detection in general works by the principle of matching signatures against programs to be scanned. With classical anti-virus tools, nearly every new malware requires an update of the signature database. In our setting, CTPL specifications take the place of malicious code signatures and allow to match whole classes of malware. Due to the broad scope of CTPL specifications, updates are only necessary when a new malware exhibits a novel type of malicious behavior.

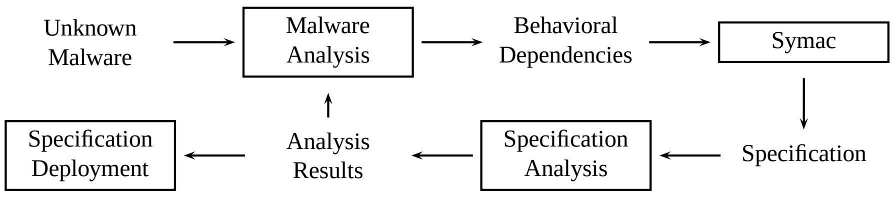

Fig. 3. Tool-supported Specification Generation Process.

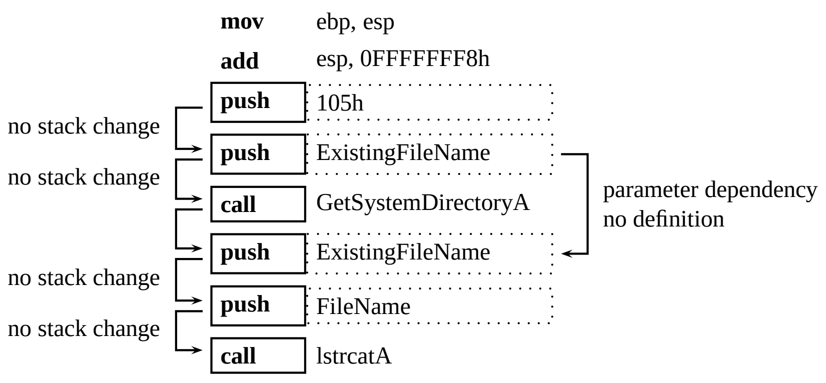

Fig. 4. Code fragment of *Small.aw*, annotated with behavioral dependencies.

To create new specifications, we follow the development process shown in Figure 3. The unpacked, disassembled code of new malware is initially loaded into Symac and manually analyzed to locate routines that exhibit characteristic malicious behavior. Once a portion of malicious code is found, we proceed to identify those instructions which are of particular relevance for program behavior. These instructions are typically system or library calls and instructions used for passing data from one call to the other. For example, consider the fragment of assembly code in Figure 4 taken from the Trojan dropper *Small.aw*. It contains two function calls that we identified as characteristic for the malware's behavior. The arguments passed to the calls are written onto the stack by two pairs of push instructions. The string buffer `ExistingFileName` is shared by both function calls. We use *Small.aw* as working example to show how a specification formula is synthesized from malicious assembly code.

The user interactively selects relevant nodes in the control flow graph (selected instructions are enclosed by boxes in Figure 4) and specifies dependencies between them (indicated by arrows). The behavior of the code fragment should be captured by the resulting specification in a general way, so it is important to encode only those dependencies between instructions that are relevant to the behavior. The user can choose between the following different types of dependencies to describe the relevant relationships between nodes:

– **Parameter abstraction**: Parameter abstractions substitute instruction parameters by variables, e.g., to allow the allocation of different registers or memory variables. In our example, the constant `105h` is irrelevant for the description of the malicious behavior and is therefore abstracted away by a variable (indicated by a dotted box).
– **Temporal restriction**: This restriction states that the first instruction has to appear before the second instruction. In our example, temporal restrictions have been added between any two instructions connected by an arrow.
– **No-stack-change restriction**: This restriction states that the first instruction has to appear before the second instruction, and that the stack is not changed by instructions that are executed in-between (in Figure 4, these restrictions ensure the correct parameter setup for the function calls)

$$\begin{aligned}
\Psi_{event}(i) &= \mathbf{EF}\varphi_i \\
\Psi_{temp}(i,j) &= \mathbf{EF}(\varphi_i \wedge \mathbf{EX}(\mathbf{EF}\varphi_j)) \\
\Psi_{stack}(i,j) &= \mathbf{EF}(\varphi_i \wedge \mathbf{EX}(\mathbf{E}[(\forall t.\neg push(t) \wedge \neg pop(t))\mathbf{U}\varphi_j])) \\
\Psi_{def}(i,j,v) &= \mathbf{EF}(\varphi_i \wedge \mathbf{EX}(\mathbf{E}[(\forall v'.\neg mov(v,v') \wedge \neg lea(v,v'))\mathbf{U}\varphi_j])) \\
&\quad \text{where } v \neq v'
\end{aligned}$$

$$\Phi_{push}(l,t) = \sharp loc(l) \wedge push(t) \qquad \Phi_{call}(l,t) = \sharp loc(l) \wedge call(t)$$

$$\begin{array}{lll}
\varphi_1 = \Phi_{push}(l_1, c_1) & \varphi_2 = \Phi_{push}(l_2, dir) & \varphi_3 = \Phi_{call}(l_3, \texttt{GetSystemDirA}) \\
\varphi_4 = \Phi_{push}(l_4, dir) & \varphi_5 = \Phi_{push}(l_5, c_2) & \varphi_6 = \Phi_{call}(l_6, \texttt{lstrcatA})
\end{array}$$

$$\exists l_1, l_2, l_3, l_4, l_5, l_6, c_1, c_2, dir.\Psi_{stack}(1,2) \wedge \Psi_{stack}(2,3) \wedge \Psi_{stack}(4,5) \wedge \Psi_{stack}(5,6) \wedge \Psi_{temp}(3,4) \wedge \Psi_{def}(2,4,dir)$$

Fig. 5. Formula patterns and instantiations corresponding to code fragment in Figure 4.

- **No-definition restriction**: This restriction states that the first instruction has to appear before the second instruction, and that there is a parameter of the second instruction that is abstracted by a variable whose value is not changed in-between.
- **Parameter dependency**: Parameter dependency ensures that the mapping of a variable in two instances of parameter abstraction is actually the same. For example, the parameter `ExistingFileName` has to be abstracted by the same variable in both push instructions. The additional no-definition restriction further guarantees that `ExistingFileName` contains the same value.

Symac prohibits cyclic dependencies, allowing a straightforward automatic generation of CTPL formulas using standard graph traversal algorithms. Every element in a finite computation path is represented by a formula $\sharp loc(l) \wedge asmInstr(par_1, \ldots, par_n)$, where the variable l references the location of the element in the Kripke structure, the predicate $asmInstr$ denotes an instruction, and the parameters $par_1, \ldots, par_n$ are either constants or variables. Building upon these basic instruction formulas, Symac generates different types of specification formulas obeying the defined dependencies. Figure 5 shows the patterns Ψ_{stack}, Ψ_{def}, Ψ_{temp}, and Ψ_{event}. The simplest pattern $\Psi_{event}(i)$ just states that some instruction, represented by φ_i, will eventually occur. We handle the restriction to a temporal order between two instruction formulas φ_i and φ_j by instantiating the pattern $\Psi_{temp}(i,j)$. Ψ_{stack} prohibits stack alteration between given instructions. $\Psi_{def}(i,j,v)$ prohibits the redefinition of variable v between two given instructions.

After instantiation of these patterns, the generated formulas are connected by a conjunction. More complex patterns can be achieved by synchronizing individual formulas through the sharing of location variables in multiple location predicates. Every unbound variable is existentially quantified, leading to closed formulas. Finally, the formulas for all single paths are connected by a disjunction. The lower part of Figure 5 shows the instruction formulas for our example and the resulting formula that contains instantiations of the according behavioral patterns.

The final specifications for the Mocca model checker contain a textual and formal description of the corresponding malicious behavior, both generated by Symac. In order to optimize the model checking process, specifications can also contain *clues*—system calls whose presence in a procedure is implied by the specification formula—that enable Mocca to skip irrelevant procedures from exhaustive analysis. Symac automatically derives these clues from a given CTPL formula [9].

4 Related Work

Commercial anti-virus products still mainly rely on classical detection techniques, such as static string matching. Recently, however, more and more virus scanners have begun using sandboxing and monitoring for detecting suspicious behavior. Szor [2] gives an excellent overview on malware detection and analysis techniques used in the industry today. The Digital Immune System (DIS), introduced by White et al. [10] is a system automating the process of malware analysis and signature generation to some extent. It executes infected binaries in a supervised environment, monitors alteration of the system state and attempts to create a signature from the observed data; if the analysis fails, the system alerts a human specialist. Christodorescu and Jha [11] describe a template based approach to semantic malware detection, particularly focusing on malware obfuscated by a set of common assembly level obfuscations. In follow-up work, they prove completeness of their malware detector with respect to these obfuscations [12].

Dwyer et al. [13] identified common patterns of temporal specifications that can be translated into different temporal logics. Wagner et al. [14] describe a method that automatically derives a model of application behavior in order to detect atypical, suspicious behavior.

5 Conclusion and Future Work

In this paper we presented our malware detection tool chain, including our recent mechanism for specification generation. We implemented the graphical tool Symac, that integrates the process of specification development and enables future automated malware analysis and specification extraction. As a next step, we will investigate to what extent the identification of relevant code and dependencies can be automated. Moreover, we plan to employ automatic analysis techniques such as pattern matching or API extraction [14–16]. Further automation of the signature generation process will allow a faster reaction to novel malicious code.

References

1. Kinder, J., Katzenbeisser, S., Schallhart, C., Veith, H.: Detecting malicious code by model checking. In: Proceedings of the GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'05). Volume 3548 of Springer Lecture Notes in Computer Science. (2005) 174–187
2. Szor, P.: The Art of Computer Virus Research and Defense. Symantec Press (2005)
3. Christodorescu, M., Jha, S.: Testing malware detectors. In Avrunin, G.S., Rothermel, G., eds.: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, ACM (2004) 34–44

4. Jibz, Qwerton, snaker, xineohP: PEiD. `http://peid.has.it/` (Last accessed: May 14, 2007)
5. Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S., Veith, H.: Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA (2005)
6. DataRescue sa/nv: IDA Pro. `http://www.datarescue.com/idabase/` (Last accessed: May 14, 2007)
7. Clarke, E., Emerson, E.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Logics of Programs. Volume 131 of Lecture Notes in Computer Science., Springer (1981) 52–71
8. Kinder, J.: Model checking malicious code. Master's thesis, Technische Universität München (2005)
9. Holzer, A.: Description languages for malicious software. Master's thesis, Technische Universität München (2006)
10. White, S., Swimmer, M., Pring, E., Arnold, W., Chess, D., Morar, J.: Anatomy of a commercial-grade immune system. IBM Research White Paper (1999)
11. Christodorescu, M., Jha, S., Seshia, S., Song, D., Bryant, R.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy (S&P 2005), IEEE Computer Society (2005) 32–46
12. Dalla Preda, M., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. In Hofmann, M., Felleisen, M., eds.: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, ACM (2007) 377–388
13. Dwyer, M., Avrunin, G., Corbett, J.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering (ICSE'99), ACM (1999) 411–420
14. Wagner, D., Dean, D.: Intrusion detection via static analysis. In: 2001 IEEE Symposium on Security and Privacy (S&P 2001), IEEE Computer Society (2001) 156–169
15. Liu, C., Ye, E., Richardson, D.J.: Software library usage pattern extraction using a software model checker. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), IEEE Computer Society (2006) 301–304
16. Ammons, G., Bodík, R., Larus, J.: Mining specifications. In: Symposium on Principles of Programming Languages, ACM (2002) 4–16