IJoFCS (2010) 1, 38-48

e-forensic PRESS

The International Journal of
FORENSIC COMPUTER SCIENCE

www.IJoFCS.org

Semantic Malware Resistance Using Inductive Invariants

Rachid Rebiha[1] and Arnaldo Vieira Moura[2]

(1) University of Lugano, Switzerland, and University of Campinas, Brasil.
Emails: rachid.rebiha@lu.unisi.ch and rachid@ic.unicamp.br.
(2) University of Campinas, Brasil. Email: arnaldo@ic.unicamp.br.

Abstract - Modern social infrastructure relies on computer security and privacy. As a result, there are many opportunities for individuals with malicious intent to cause great harm. This work aims to provide a theoretical basis for the design of static and dynamic software platforms that can create a suitable architecture for automatic malware detection and analysis. We demonstrate how formal methods involving programs static and dynamic analysis can be used to build such architectures, and propose automatic semantic malware detection and model extraction methods, circumventing difficulties met by other recent approaches. Our new technique for identifying malware involves automatically extracting invariant elements found in specific malware codes. These "malware-invariants," which remain unchanged even in obfuscated virus strains, can be used by semantic analysis programs as signatures that define the malicious code. We propose a host-based intrusion detection system using automatically generated models, where system calls are guarded by verification with pre-computed invariants. In this way, we can to identify deviations during the execution of applications. Our method also provides a way to detect software bugs and application vulnerabilities. We also show that any malware or intrusion detection system based on a static analysis method will be strongly reinforced by the possession of a database of precompiled invariants.

1 Introduction

Invariant properties are assertions, expressed in a specified logic, that hold true on every possible run of the system. A malware includes any software deemed to have malicious intent. A program that fits this classification could be a virus, trojan horse, or worm, to name some common examples. Malware represents a threat to our increasingly computer-reliant society. Thus, it is vital to understand the behavior of malicious software in detail. All current security systems, including anti-virus and other detection systems, suffer from a lack of automation in their malware analysis. In order

to provide automatic in-depth malware analysis and precise detection, a system must be able to automatically interpret malicious behaviors, not just syntactic signatures. Current malware detectors are "signature-based." That is, the presence of a malicious program is detected when part of its code matches a known database of byte-signatures. Such current malware detectors are based on sound methods checking if the executable matches byte-signatures located in a database of regular expressions that specify byte or instructions sequences. But the main drawback to this approach is that malware writers can obfuscate [1] the code slightly to evade detection. Small changes that alter the program's signature are enough to bypass many security checks. In addition, current intrusion detection systems often suffer from overly coarse abstraction methods. The number of malware variants that can use obfuscation methods increases exponentially each time a new malware type appears.

Malware writers can easily generate new undetected viruses, and anti-virus programs lag behind, as they must be updated very frequently. This update process is often much slower than the spread of the malware, since each new variant must to be analyzed precisely and its signature added to the database.

We propose a better method using the invariant sections of malware code. Invariant properties are aspects that hold true during every possible execution of the software. We seek to identify malware by automatically extracting the "malware-invariants" [42] from a specific code, and using these invariants in semantic analysis software. In order to do so, we need to adopt formal methods currently used to statistically verify a system's correctness. Importantly, malware-invariants remain unchanged in virtually all of the obfuscated versions of the code. In other words, most of the novel strains of the malicious piece of code obtained by obfuscation techniques would still share the same unchanged qualities. Thus, by targeting these invariant properties, an anti-malware system would need only one semantic signature for each virus family.

Using this sort of static analysis with generated invariants, we show how to construct a suitable architecture for host-based intrusion detection systems. Host-based intrusion detection systems, which have seen tremendous progress in recent years, are systems that monitor application execution and report deviations from its programmed statistical model. The main drawback of these detection systems is that they monitor only program control flow structure and lack the precision to resist mimicry attacks [2] and non-control data attacks [3]. Our new strategy addresses the various deficiencies of current intrusion detection systems. Our model captures the control flow structure and the data flow characteristics of a program. It is automatically generated with very little monitoring overhead. Further, our method can show mathematically prove and report any intrusion once the violation of an application invariant is observed during the execution of the application. Being rigorous, they allow for no false alarms. Our model does not only prevent mimicry and non-data control flow attacks, it also can be used to detect logic bugs or other vulnerabilities in applications.

As the main contribution, we prove that any malware or intrusion detection system based on static analysis methods will be strongly reinforced by the presence of pre-computed malware invariants, and will also be weakened by their absence.

In Section 2, we present some malware characteristics and obfuscation techniques, as well as some current detection methods. In Section 3, we show how to generate a semantic signature based on a malware invariant. In Section 4, we discuss a theoretical basis that supports the design of our malware analysis platforms using formal methods. In Section 5, we present our intrusion detection monitoring system with vulnerability auditing. Finally, we present our conclusions in Section 6.

2 Malware Characterization, Identification, and Obfuscation

2.1 Malware and Virus Characterization

All malware programs display the following behaviors:

1. An infection strategy
2. Malicious actions called "payloads"
3. Boolean control conditions, or triggers, to determine when a payload should be activated.

A classification of malware with respect to their effects and propagation methods has been proposed by McGraw and Morrisett [4]. Adelman [5] and Cohen [6] showed that the security research community lacks a mathematical formalism that can serve as a scientific basis for malware classification. We can define three properties in particular that characterize classes of malware. The first is active propagation, based on whether a malicious program can propagate passively using self-instance replication and/or self-modification. The second is population evolution. A malware family can be characterized by its ability to form variants. The third is the context-dependence exhibited by the malicious program. That is, if the payload release requires other executable code or a pre-compilation step. A classification of some malware types using these three properties is shown in table 1. Other malware types, such as hybrid malware programs, or network assaults like the use of so-called "botnets" for denial-of-service attacks, could be also classified by combining and extending these properties. In subsection 5.2, we will consider more specific characterization properties, like obfuscation and encryptions techniques, that will be relevant to our approach to automated malware analysis.

2.2 Identifying Malware Concealment Behaviors

Current malware detection programs are "signature-based," in that they attempt to match the signature of a suspect program to entries in a database. These malware detectors are based on sound methods, that is, if the executable matches byte-signatures, then they guarantee the presence of the malicious behavior. These databases contain regular expressions that specify byte or instruction sequences that are considered malicious. The main problem with this approach resides in the fact that these methods have a low degree of completeness and these databases must also be updated frequently as new malware is created and spread.

Type	Active Propagation	Variants	Context-free
Virus	Yes	Yes	No
Worm	Yes	Yes	Yes
Spyware	No	No	Yes
Adware	No	No	Yes
Trojan	No	No	No
Back Door	No	No	Partially
Rabbit	Yes	No	Yes
Logic Bomb	No	No	Partially

Table 1: Characterization of malware types

As a result, malware writers can evade detection by injecting code into their programs in such a way that preserves the malicious behavior, but slightly changes the resulting signature. This technique is called obfuscation [1] and, when implemented, makes the malware variants created very difficult to detect. Commercial anti-virus scanners are particularly susceptible to this type of method.

Common obfuscation techniques include polymorphism and metamorphism. In these cases, the virus obfuscates its payload during its latency period but de-obfuscates the malicious code prior to execution. Once a new type of malware appears, the number of derivative versions generated by obfuscation can increase exponentially. In this way, malware writers can easily generate new undetected viruses. Anti-virus databases must be updated to include every variant in its signature database in order to provide complete security.

2.3 Obfuscation Strategies for Infection Mechanisms, Triggers, and Payloads

Malware can be obfuscated it each of its three main mechanisms, namely, infection, trigger-detection, and payload delivery. However, the obfuscation must be reversed before the code can be made functional again. This requires a decryption loop, which de-obfuscates parts of the body of the code in a specific writable memory location. This de-obfuscation mechanism can use a random key or an encryption library. Thus, the encrypted malicious code can be very difficult to detect. Because of this, polymorphic malware detectors sometimes try to detect the decryption loop itself. The unchanging part of an encrypted virus, which can randomly modify its key for each instance, is the decryptor loop. To avoid this, a polymorphic virus modifies its decryption loop at using several automatic transformations, and a single progenitor virus can easily generate billions of loop versions [7]. Metamorphic viruses can change even the body of their code using a variety of obfuscation techniques when they replicate. To modify the decryption loop, or to obfuscate some code, malware programs can use a mutation engine [1] to rewrite the loop with other semantically equivalent sequences of instructions [8] and/or rename register memory locations. This can be done using unconditional jumps, inlining or outlining the body of function codes, using new function calls, inserting junk code, using treaded versions, and a host of other tricks.

Example 1. As an illustration, consider the following two pieces of pseudo-code:

```
1  Function_0();
3  Function_0 {
4   R_1 = 3;
5   R_2 = R_1 + R_2;
6   R_3 = R_1 * R_2;
7   R_4 = R_2 – 3;
8   R_5 = R_4 * R_3;
9   R_6 = R_5 / R_4;
10  Return;}
```

```
1  Function_1(); Function_2();
4  Function_1 {
5   R_l = R_2 / 2;
6   R_2 = 6;
7   R_3 = R_2 + R_1 * R_2;
8   Return;}
10 Function_2 {
11  R_4 = R_1;
12  R_5 = 2 * R_1 * R_1 * R_!;
13  R_6 = R_3;
14  Return;}
```

The code on right is simply an obfuscated version of the code on the left which maintains the overall semantic behavior.

We will see in the following section that decryption loops still share some common features, and that these invariants are more difficult to morph in an automated fashion.

3 Malware Invariant Semantic Signatures

3.1 Static Analysis for Detection and Identification

To be able to analyze an unknown binary code, one needs an intermediate representation [9, 10], as shown in example 2.

Example 2.

```
1 ...          //      ...
2 dec ecx      //      ecx <-- ecx -1
3 jnz 004010 B7  //     If (! ecx =0 ) goto 004010 B7
4 mov ecx , eax  //     ecx <-- eax
5 shl eax , 8    //     eax <-- eax << 8
6 ...          //      ...
```

The right column shows the semantics of each set of x86 executable instructions expressed in a C-like notation [9, 10, 48, 49]. Also, [48, 49] provide interface with SMT solvers [50] (automated deduction methods for checking the satisfiability of first-order formulas with respect to some logical theory T of interest) and with decision procedures that support the SMT standard library format [51]. Using such tools, one can obtain a C-like intermediate representation and check automatically if a

specific property, an assertion, holds true at a specific location on every possible execution of the code. These tools will be helpful to check if an assertion is an inductive invariant.

Some malware detection algorithms [11, 12, 13] attempt to incorporate instruction semantics in order to detect malicious behavior. Semantic properties are more difficult to conceal in an automated fashion than are syntactic properties. The main problem with this approach is that it relies on abstracted semantic information *e.g. def-use* information. Instead of dealing with regular expressions, these programs try to match a control-flow graph, enriched with def-use information to the suspect binary code. Those methods eliminate a few simple techniques of obfuscation as it is very simple to obfuscate def-use information by adding any junk code or by reordering operations that would either redefine or use the variables present in the def-use properties.

Our approach to the problem of malware detection generates quasi-static invariants directly from the specified malware code, and uses them as semantic signatures that we call malware-invariants. To detect a piece of malicious code, we could check if there is an assertion in our malware-invariant database that holds in one of the reachable program states. In order to do so, we use formal methods for invariant generation and assertion checking. This will make it much more difficult for a malicious code writer to evade detection using common obfuscation techniques. Thus, to defeat each family of viruses, we would need to generate only a few semantic signatures.

We propose to use, combine and compose many static and dynamic tools in order to automatically generates invariants which are semantic-aware signatures of malwares, i.e. they are malware-invariant.

3.2 Automatic Generation of Malware Invariants

Malware invariants could be computed directly using one of several invariant generation methods, such as based on a complementary logic and a collection of these tools could be made available though a communication framework. We provide a theoretical basis for the construction of a static analysis platform that can form a suitable architecture for the extraction and identification of suspected malicious behaviors. Here, we consider several invariant generation techniques where the automatically computed invariants would be expressed by specific logics.

For example, methods described previously [40, 41], can handle logics with non-linear arithmetic and inequalities. Using these techniques, one can generate non-linear invariants in several guises, including logical assertions, formulae regarding the relationship of values in registers, memory locations, variables, system call attributes, and many others.

Example 3. Consider the following piece of code expressed as an intermediate representation after a transformation process.

```
1 integer R_1,R_2,eax,ebx,ecx,OxN ;
2 ...
3 where (eax=R_1 && ebx=ecx=R_2 && OxN =0)
4 ...
5 while (eax != ebx ){
6 while (eax > ebx ){
7 eax = eax - ebx ;
8 y_4 = OxN + ecx ;
9 }
10 while (ebx > eax ){
11 ebx = ebx - eax ;
12 ecx = ecx + 0xN;
13 }
14 }
15 ...
```

We can, for instance, generate the following invariant:

Invariant Generation [Logic: Non-linear Arithmetic]
eax * ecx + ebx * OxN – R_1 * R_2 = 0

In order to handle system calls and function calls effectively, we consider the techniques described in Tiwari et al. [14, 15], which provide invariants over a logic with uninterpreted function and inequality. This theory uses several level of abstract interpretation [47], and is based on unification theory [16]. However, the

implementation of this method also requires modifications of the CIL tool [17] to generate an intermediate representation before proceeding.

Example 4. Consider the following piece of code with uninterpreted functions called Func_F and Func_G. Note that these could be any system call or function call with a similar signature.

```
1 void
2 Func_File_Descriptor(int fd_1 ,int fd_2){
3 ...
4 int r1 ,r2 ,r3 ,r4 ,r5 ,a;
5
6 if ( fd_1 == fd_2 ) {
7 r1= Func_F(fd_1 - fd_2 );
8 r2= Func_F(Func_F(0));
9 }
10 else {
11 r2= Func_F(r1);
12 }
13 r3= Func_F(r1);
14 a=fd_2 -1;
15 if ( fd_1 == a ){
16 r4= Func_G(fd_1 , 2*fd_1+1);
17 r5= Func_G(fd_1 , Func_G(fd_1 , 2*fd_2-1));
18 }
19 else {
20 r5 = Func_G(fd_1 , r4);
21 }
22 }
```

For this piece of code, we could generate the following invariant

```
Invariant Generation [Logic: Uninterpreted Function]
r3 = r2
fd_2 + -1.000000*a = 1.000000
r2 = Func_F(r1)
r5 = Func_G(fd_1 , r4)
```

We can see that this exercise yielded a set of relationships between function or system calls returned values and attributes. If we interpret Func_F and Func_G as being two common system calls dup() and dup2(), we immediately obtain a similar invariant:

$$(r3=r2)\grave{U}(fd\_2-a=1)\grave{U}(r2=dup(r1))\ \grave{U}(r5=dup2(fd\_1,r4)).$$

We could just as easily have considered any other logical system with an associated invariant generation techniques. Our main contribution here is to demonstrate that many obfuscation techniques, as introduced in Section 2.3, will not affect the computed invariants.

4. Quasi-Static Malware Analysis

We propose new concepts and a theoretical framework [42] that can be used to compute exact invariants, i.e. inductive provable invariant, using hypotheses, i.e. likely invariants, that are true properties on all observed execution traces, in a certain training period. We could then generate likely invariants which are properties that hold at any program point in the observed execution trace, using test methods. We can turn likely invariants into exact invariants using verification methods, such as assertion checkers [18, 19]. We call this new view of program analysis "quasi-static analysis." Figure 1 illustrates the architectural structure of this new framework.

Some of the likely-invariants computed by a Dynamic Analysis are real invariants. They hold in all possible executions of the program. Then, using theorem provers or assertions checkers, one may confirm proposed properties during the dynamic analysis are real invariants. These computed malware-(likely)invariants are then treated as the program signatures. One can then use pushdown model checking techniques and theorems proving methods, combined with program verification tools, to detect the presence of malicious behavior described by our malware-invariants. If the malware invariants describe reachable states in the verification process, then we can guarantee that the software displays the suspicious behavior. Written more formally, a malware invariant Φ_Sign can be identified using the procedure just described. Then, one can use similar invariant generation methods, applied to the vulnerable executable code being inspected, to generate an invariant ψ_Exec. Then, one could check if ψ_Exec $\Rightarrow$ Φ_Sign using a theorem prover, an SMT Solver, or other decision procedure. Alternatively, one could generate a push down system to model

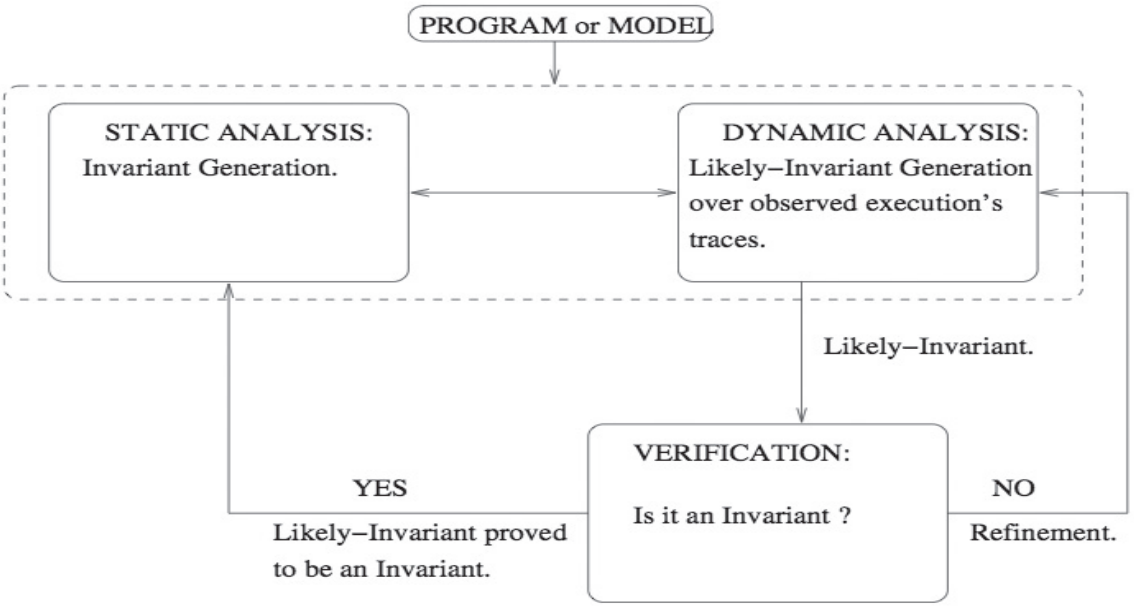

Figure 1: Quasi-static analysis framework

the malicious behavior by combining control flow and data flow analysis. These abstract state graphs, which we call malware PDS, capture both the control and data flow properties of the malicious behaviors. We can then combine malware invariants and malware PDS to form formal signatures. One can, then, build formal signature database as illustrated in figure 2.

On the other hand, we could also extract invariants from the code being inspected using similar methods and check, using decision procedures, if one of these signatures implies a known malware invariant. In addition, we could check if the code behavior allows the possible run in one a previously discovered malware PDS. The malware detector method described in this Section is sound with respect to the signature that is being considered as the representation of malicious behavior. This point is illustrated in Figure 3.

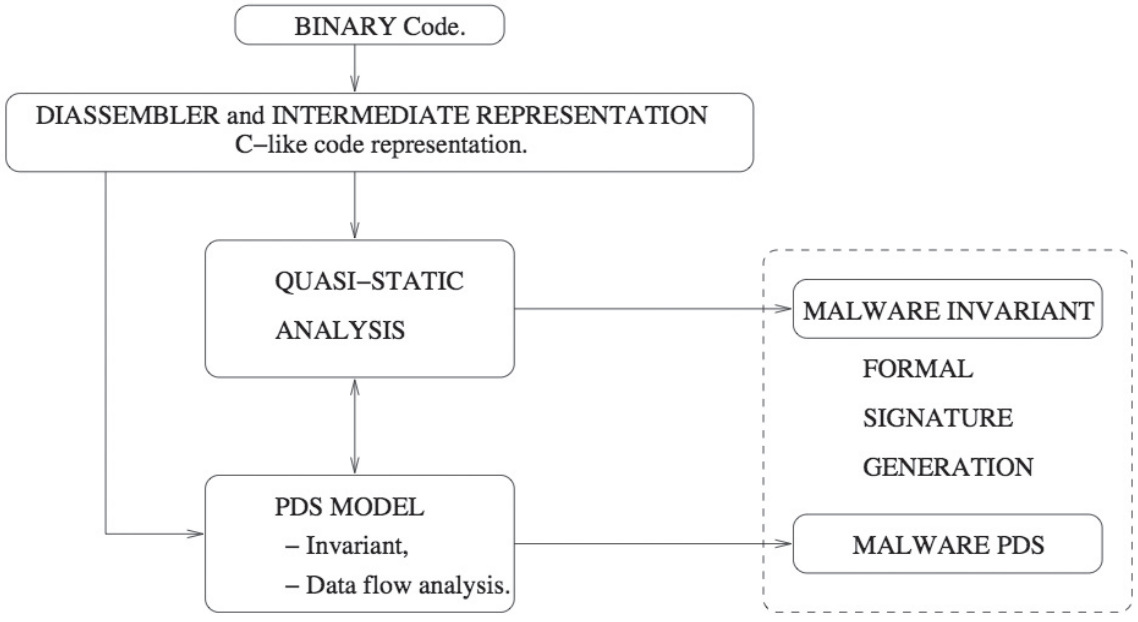

Figure 2: Malware invariant signature and Malware PDS generation.

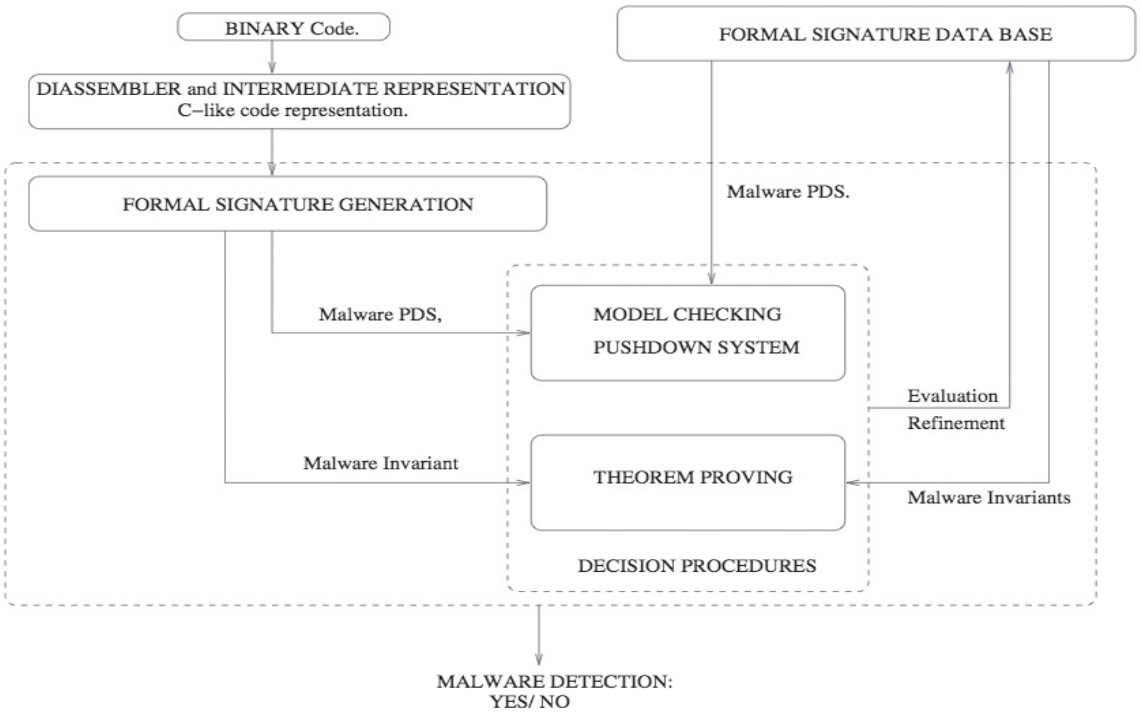

Figure 3: Formal method for malware analysis.

In Section 3, we described a working theoretical platform, and its associated host-based intrusion detection system, following a similar proof-of-concept. A semantic malware detector can then be built upon this intrusion detection system.

5 Guarded Monitors For Host-based Intrusion Detection Systems

Host-based intrusion detection systems (H-IDS) monitor specific application execution in order to report any deviation from its model of permitted actions, which are pre-computed in an attack free environment [20, 21, 22]. Current H-IDS attempt to check for anomalies in the stream of system calls generated by the execution of applications [20, 21, 22, 23, 24]. These models need to capture as many of the legitimate sequence ofsystem calls as possible. As mentioned before, these monitors often use overly coarse abstraction models that capture only the control-flow structure of programs. These approaches suffer from the lack of any significant data-flow analysis, and this imprecision explains why they can be defeated by:

- automated mimicry attacks [2, 25], which allow the execution of malicious code designed to produce a sequence of system calls that conform to the model,
- non-control-flow attacks [3, 26], and
- logic bugs and vulnerabilities difficult to detect and easy to produce.

To implement our method, we automatically generate invariants in a safe environment. Any subsequent system call is checked against these permitted invariants. Our model would detect any automated mimicry attack, non-control-flow attack, or logic bug, as these events inevitably violate one of our pre-computed allowed invariants. Our monitor captures control and data flow integrity information to insure that:

- system calls occur in the order specified by the monitor (control flow integrity) and
- pre-computed specified invariants are checked at all system call locations (data flow integrity).

Such invariants, which can include assertions, crucial properties about functions and system call arguments, returned values, branch predicates at control locations, input variables, and others, are checked against the image of the process that was obtained while monitoring the execution of the applications. These monitors are visibly pushdown automaton [27], and can be built automatically using a combination of static and dynamic analysis. This has been previously done using predicate abstraction [44], invariant generation techniques [31, 45], likely-invariant generation and dynamic analysis [29], inter-procedural propagation techniques, and assertion checking techniques [30, 46]. Any behavior that is impossible to simulate by our monitor will raise an alarm and the identification of an intrusion. An alarm is a proof of an abnormal behavior issued by the violation of control and data flow integrity induced by an intrusion.

As we discuss in R. Rebiha and H. Saidi [42], our monitor has the following properties:

- Expressivity, in that can detect more attacks than the approaches described in reference [28];
- Effectiveness, since our monitor can detect all attacks generated by the methods of reference [25]; and
- Precision, since our monitor can detect logic bugs and vulnerabilities when determinism is reach during its construction.

Example 5. Consider the program Func_File_Descriptor depicted in Example 3 (Section 3.2). Here we interpret the functions Func_F and Func_G as follows:

```
1 int Func_F(int i_1, int i_2) {
2 int t_1,t_2,t_3;
3 t_1:=i_1; t_2:=i_2 + 1;
4  if (t_1 < t_2){
5    t_3 := t_2 – t_1;
6    t_1 := t_3 + t_1;
7  }
8  else {
9    t_3 := t_1 – t_2;
10   t_2 := t_3 + t_2;
11  }
12 t3 := t1 / t2;
13 return t3;
14 }
```

```
1  int Func_G(int a, int b) {
2  int ret ;
3  if (b < 15) {
4    write(b,..,..);
5    ret := b;
6   }
7  else {
8    close(b,..,..);
9    ret := a;
10  }
11 return ret ;
12 }
```

Figure 5 describes our generated model using invariant generation and propagation. We consider the predicates:

$$B \hat{U} (fd\_1 = fd\_2 - 1) \text{ and } P \hat{U} (b < 15).$$

As we can see we have an action guarded transition system. Transitions are labeled by a deduction rule, such as:

$$(\emptyset B \grave{U} P) \mid \textbf{write}(b,\ldots).$$

In that case, the transition is allowed in our model if and only if

1. The application is willing to perform a system call write(b,…) and
2. The invariant ($\emptyset B \grave{U} P$) is checked against the image of the process and holds at the system call location mentioned above.

Locations in our model are represented by the blue squares in Figure 4. We detect an intrusion if there are no transitions allowed at one location that is not the final location of the automaton. We note that we could add here many more pre-computed invariants to be checked at system call locations, for instance, we could add the one generated in Example 3 (Section 3.2).

Such monitors can also detect the non-control-flow attacks. For instance, we were able

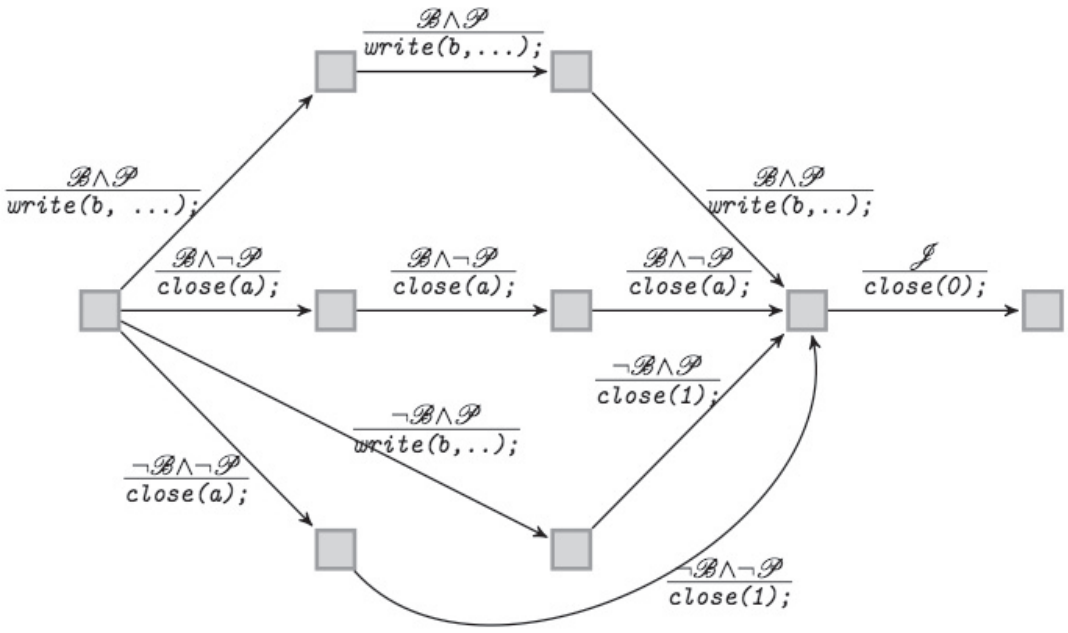

Figure 4: Example of guarded monitor generated for Example 5.

to detect attacks [3] on **ghttpd** where filenames were overwritten, and on **wu-ftpd** where userid's were manipulated. We also detected attacks [36] on the command **rm.c**, where a race condition could rewrite path name. Our approach also allows us to detect attacks [37, 38] where the authentication flags were manipulated.

6 Conclusions

We introduced an extensive invariant-based formal platform for malware analysis. In particular, we proposed a method for generating invariants directly from specified malware code and using them as semantic signatures. These signatures will not be affected by obfuscation methods that create malware variants, which were previously difficult to detect. The invariants found represent a concise semantic summary of the malicious behavior of an entire family of viruses. Following this theoretical framework, we described a set of intrusion detection systems using automatically generated models, where system calls are guarded by pre-computed invariants in order to detect deviations during the execution of applications. Our method also provides a way to detect logical errors and application vulnerabilities. Existing malware detection systems would be strongly reinforced by adopting our invariant detection method. Our platform is also flexible as any invariant generation method could be incorporated. In other words, it is an open, lightweight architecture, where any invariant generation tool can be connected into a tool bus where invariants, expressed in different logics, will help in the identification of malicious behavior or in the construction of a precise intrusion detection system while conserving system resources.

References

[1] Nachenberg, C.: Computer virus-antivirus co-evolution. Commun. ACM 40(1) (1997) pages 46-51.

[2] Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In Sandhu, R., ed.: Proceedings of the 9th ACM Conference on Computer and Communications Security, Washington, DC, USA, ACM Press (November 2002).

[3] Chen, S., Xu, J., Sezer, E.C., Gauriar, P., nkar K. Iyer, R.: Noncontrol-data attacks are realistic threats. In: USENIX Security Symposium (2005).

[4] McGraw, G., Morrisett, G.: Attacking malicious code: A report to the infosec research council. IEEE Software 17(5) , (2000). pages 33-41.

[5] Adelman, L.M.: An abstract theory of computer viruses. In: Advances in Cryptology CRYPTO'88, (1988).

[6] Cohen, F.: A short courses on computer viruses. In: Wiley, (1990).

[7] Fisher, C.: Tremor analysis(pc). In: Virus Diget, 6(88). (1993).

[8] Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, Department of Computer Science, Auckland New Zealand (1997).

[9] Gopan, D., Reps, T.W.: Low-level library analysis and summarization. In Damm, W., Hermanns, H., eds.: CAV. Volume 4590 of Lecture Notes in Computer Science., Springer (2007). pages 68-81.

[10] Brumley, D., Jager, I.: The bap hanbook. In: Technical Report TR. (2009).

[11] Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005), Oakland, CA, USA, ACM Press (May 2005) 32-46.

[12] Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behavior. In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). (2007).

[13] Dalla Preda, M., Christodorescu, M., Jha, S., Debray, S.: A semantics based approach to malware detection. In: POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (2007). pages 377-388.

[14] Gulwani, S., Tiwari, A.: Assertion checking unified. In: Sixth Int. Conf. on Verification, Model Checking and Abstract Interpretation VMCAI'05. (2005).

[15] Gulwani, S., Tiwari, A.: Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In Sestoft, P., ed.: European Symp. on Programming, ESOP 2006. Volume 3924 of LNCS, (2006). pages 279-293.

[16] Baader, F., Siekmann, J.H.: Unification theory. (1994), pages 41-125.

[17] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, London, UK, Springer-Verlag (2002), pages 213-228.

[18] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast: Applications to software engineering. In: Journal on Software Tools for Technology Transfer (paper from FASE2005), (2007).

[19] Xie, Y., Aiken, A.: Saturn: A sat-based tool for bug detection. (2005), pages 139-143.

[20] Wagner, D., Dean, D.: Intrusion detection via static analysis. In: IEEE Symposium on Security and Privacy. (2001) , pages156-169.

[21] Feng, H.H., Gi n, J.T., Huang, Y., Jha, S., Lee, W., Miller, B.P.: Formalizing sensitivity in static analysis for intrusion detection. In: IEEE Symposium on Security and Privacy, IEEE Computer Society, (2004) 194.

[22] Gopalakrishna, R., Spa ord, E.H., Vitek, J.: Efficient intrusion detection using automaton inlining. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (2005), pages 18-31.

[23] Forrest, S., Longsta , T.: A sense of self for unix processes. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy. (May 1996), pages 120-128.

[24] Giffin, J.T., Dagon, D., Jha, S., Lee, W., Miller, B.P.: Environment sensitive intrusion detection. In Valdes, A., Zamboni, D., eds.: Recent Advances in Intrusion Detection (RAID). Volume 3858 of Lecture Notes in Computer Science., Springer (2005) pages 185-206.

[25] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: Proceedings of the 14th USENIX Security Symposium, (2005).

[26] Bhatkar, S., Chaturvedi, A., Sekar, R.: Data flow anomaly detection. In: IEEE Symposium on Security and Privacy, IEEE Computer Society (2006) 48-62.

[27] Alur, R., Madhusudan, P.: Visibly pushdown languages. In: STOC'04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing, New York, NY, USA, ACM (2004), pages 202-211.

[28] Giffin, J.T., Jha, S., Miller, B.P.: Automated discovery of mimicry attacks. In Zamboni, D., Kruegel, C., eds.: RAID. Volume 4219 of Lecture Notes in Computer Science., Springer (2006), pages 41-60.

[29] Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Science of Computer Programming 69(1-3) (December 2007), pages 35-45.

[30] Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with Blast. In Ball, T., Rajamani, S.K., eds.: Model Checking Software, 10th International SPINWorkshop. Portland, OR, USA, May 9-10, 2003, Proceedings. Volume 2648 of Lecture Notes in Computer Science., Springer (2003), pages 235-239.

[31] Gulwani, S., Tiwari, A.: Combining abstract interpreters. In Ball, T., ed.: ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2006. (2006), pages 376-386.

[32] CVE: The common vulnerabilities and exposures (cve) web site http://cve.mitre.org/.

[33] Pekka, K., Kalle, L.: Ssh1 remote root exploit. http://www.hut./kalyytik/hacker/ ssh-crc32-exploit korpinen lyytikainen (2002).

[34] Starzetz, P.: Crc32 sshd vulnerability analysis http://packetstormsecurity.org/0102-exploits/ssh1.crc32.txt (2002).

[35] Rebiha, R., Moura, A.V.: Automated malware invariant generation. In: "Best Paper Award", 6th International Conference on Forensic Computer Science, ICoFSC'2009. (2009).

[36] Rebiha, R., Matringe, N., Moura, A.V.: Endomorphisms for non-trivial non-linear loop invariant generation. In: 5th Int. Conf. Theoretical Aspects of Computing, LNCS (2008) 425–439.

[37] Rebiha, R., Matringe, N., Moura, A.V.: Generation invariants for non-linear hybrid systems by linear algebraic methods. In: 17th Int. Static Analysis Symposium, SAS2010, LNCS (2010).

[38] Rebiha, R., Saidi, H.: Quasi-static binary analysis: Guarded model for intrusion detection. In: TR-USI-SRI-11-2006. (November 2006).

[39] Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In Grumberg, O., ed.: Computer Aided Verification. Volume 1254 of LNCS., Haifa, Israel, springer (June 1997), pages 72-83.

[40] Gulwani, S., Tiwari, A.: Assertion checking over combined abstraction of linear arithmetic and uninterpreted functions. In Sestoft, P., ed.: European Symp. on Programming, ESOP 2006. Volume 3924 of LNCS. (2006), pages 279-293.

[41] Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast: Applications to software engineering. In: Journal on Software Tools for Technology Transfer (paper from FASE2005), (2007).

[42] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fix points. In: Conf. Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, os Angeles, CA, ACM Press, NY (1977), pages 238-252.

[43] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: Binary Analysis for Computer Security, (http://bitblaze.cs.berkeley.edu/)

[44] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: BitBlaze: A New Approach to Computer Security via Binary Analysis. Proceedings of the 4th International Conference on Information Systems Security, (2008).

[45] Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Proceedings of the 8th International Workshop on Satisfiability Modulo Theories. (Edinburgh, England). (2010).

[46] Ganesh, V., David L., D. A Decision Procedure for Bit-Vectors and Arrays. Computer Aided Verification (CAV '07)}, (Berlin, Germany). LNCS (2007).