Automated Malware Invariant Generation

Arnaldo V. Moura, and Rachid Rebiha,

Abstract—In our days, any social infrastructure relies on computer security and privacy: a malicious intent to a computer is a threat to society. Our project aims to design and develop a powerful binary analysis framework based on formal methods and employ the platform in order to provide automatic in-depth malware analysis. We propose a new method to detect and identify malware by generating automatically invariants directly from the specified malware code and use it as *semantic aware* signatures that we call *malware-invariant*. Also, we propose a host-based intrusion detection systems using automatically generated model where system calls are guarded by pre-computed invariant in order to report any deviation observed during the execution of the application. Our methods provides also technics for the detection of logic bugs and vulnerability in the application. Current malware detectors are "*signature-based*" but is it well-known that Malware writers use *obfuscation* to evade current detectors easily. We propose *automatic semantic aware* detection, identification and model extraction methods, hereby circumventing difficulties met by recent approaches.

Index Terms—Formal Methods, Security, Forensic Computer Science, Static and Dynamic Binary Analysis, Malware/Intrusion/Vulnerability Detection, Identification and Containment.

1 Introduction

Invariant properties are assertions (expressed in a specified logic), that hold true on every possible behaviors of the system. A *malware* is a program that has malicious intent. Examples of such programs include viruses, trojans horses, and worms. Malicicous intent to computers are virulent threat to society. We deeply need to understand the *malicious behavior* in details. All present security systems (anti-virus, detection systems...) suffer form the lack of automation in their malware analysis. In order to provide automatic in-depth malware analysis and precise detection systems, one need to be able to extract automatically the malicious behaviors and not only its syntaxic signature.

We propose a new method to detect and identify malware by generating automatically invariants directly from the specified malware code and use it as *semantic aware* signatures that we call *malware-invariant*. To do so, one need to adapt *formal methods* currently use to verify and proof statically systems correctness.

Current malware detectors are "*signature-based*": the presence of the malicious behavior is detected if the malicious code matches matches byte-signatures. These current malware detectors are based on sound methods as, if the executable matches byte-signatures located in a database of regular expressions that specify byte or instructions sequences.

But the main problem is that malware writers can then use *Obfuscation* [26] to evade current detectors easily. To evade detection, hackers frequently use obfuscation to morph malware and evade detections by injecting code into malwares that preserves malicious behavior and makes the previous signature irrelevant.

The number of derivative malwares by obfuscation increases exponentially each time a new malware type appear. Malware writers can easily generate new undetected virus and then the anti-virus code has to update its signature database frequently to be able to catch the new virus. The main difficulty remain in the updates procedures because the new malware needs to be analyzed precisely and the new signature needs to be created and distributed as soon as it is possible to control the propagation.

- *Rachid Rebiha is with the Faculty of Informatics, University of Lugano USI, Switzerland, Lugano, 9400 and with the Institute of Computing, University of Campinas Unicamp, Brazil, Campinas SP.*
  E-mail: rachid.rebiha@lu.unisi.ch
- *Arnaldo V. Moura is with the Institute of Computing, University of Campinas Unicamp, Brazil, Campinas SP.*
  E-mail: arnaldo@ic.unicamp.br

List of authors are in alphabetic order.

The new strategy would be to generate quasi-static invariants directly from the specified malware code and use it as *semantic-aware* signatures that we call *malware-invariants*. Thus, for one familly of virus we would have only one semantic signature.

We also show how these invariant to detect intrusion. Our intrusion detection system mathematically (no false alarm) prove and report any intrusion once the violation of an application invariant is observed during the execution of the application. Our methods allows also propose how to detect logic bugs and vulnerability in the application..

As the main contribution, we proved that any approach to static analysis based malware/intrusion detection will be strongly reenforced by the presence of pre-comptued invariants and will be weakened by their absence. In the following section we will introduce formal methods and malwares. In section 3, we present a quasi-static binary analysis. Finally we present guarded monitor generation for indtrusion detection and vulnerability auditing.

2 Formal Methods and Malware

2.1 Formal Methods and Verification

Formal methods aim at modeling (e.g. building specifications expressed in a specific logic, design or code) and analysing (e.g. verification or falsification of) a system with methods derived from or defined by underlying mathematically-precise concepts and their associated algorithmic foundations.

Formal methods research aims at discovering mathematical techniques and design algorithms to establish the *correctness* of software/hardware/concurrent/embedded/hybrid systems, i.e. to prove that the considered systems are faithful to their specification. On large or infinite systems (with a huge or infinite numbers of reachable state in any of its possible behaviors) *total* correctness is usually not practically possible. That is why we could restrict our focus on safety and liveness properties that any well behaved engineered critical systems must guarantee (e.g. by using *static program analysis*, one could prove a software free of buffer overflow, segmentation fault or non-termination.).

Static Analysis are used to generate and infer *invariant properties*, which are assertions, that hold true on every possible behaviors of the system. Thus static analysis provides provable guarantees that the most exhaustive and rigorous testing methods could not reach.

In infinite state systems, safety properties can be proved by induction. Actually the verification problem of safety properties is reduced to the problem of invariant generation. First, an inductive invariant has to be obtained for the system. This means that the invariant holds in the initial state (initiation condition) of the system and every possible transition preserves it (consecution conditions). That is, if the invariant holds in some state then it continues to hold in every successor state as well. Now if the inductive invariant implies the desired property then the proof is complete. Finding inductive invariants automatically is an essential part in proving safety (such as in program analysis) and liveness properties.

The Floyd-Hoare [10], [11] inductive assertion technique depends on the presence of loop invariants to establish total correctness. Invariants are essential to prove and establish safety properties, (such as no null pointer deferenciation, buffer overflows, memory leak or outbounds array access,...), liveness properties (such as progress or termination.).

In order to tackle the recent most virulent attacks and vulnerabilities, we show how the precision of malware/intrusion detection/identification systems depends on the ease with which one can automate the discovering of non trivial invariants in the application.

2.2 Malware and Virus Charaterisation

A *malware* is a program that has malicious intent. Examples of such programs include viruses, trojans horses, and worms. These malicious intent are structured by a three recurrent behavior:

1) following some *infection strategies*,
2) executing a set of malicious actions, procedures which is called the malware *payload*,

Type	Active Propa.	Inst. Evolution	Context-free
Virus	yes	> 0	no
Worm	yes	> 0	yes
Spyware	no	0	yes
Adware	no	0	yes
Trojan	no	0	no
Back Door	no	0	partially
Rabbit	yes	0	yes
Logic Bomb	no	0	partially

TABLE 1
Charaterisation for malware types.

3) evaluating some boolean control conditions, called *triggers* to defined when the *payload* will be activate.

A classification of malware with respect to their goals and propagation methods is proposed in [23]. Also, [24] [25] shown that the research security community will deeply need a mathematical formalisms that could serve as scientific basis for the classification of malware. We could distinguish three properties that charaterise a class of a considered malware:

- A malware could be propagate passively or actively using self-instance replication and/or self-modification *active propagation*.
- A malware could be characterise by the evolution of the number of malware instance *instance population evolution*
- in order to perform its malicious intent, a malware could depend on external context, e.g. it could require other executable code like binary/interpreted code, or a pre-compilation step (*context-free*).

We give the classification of some types of malware using these three properties in table 1. Some other hybrid types and the network-based denial-of-service types (*botnet*, *zombie network*, ...) could be also considered with combined/extended properties. In the following section 3 we will defined more specific charaterisation properties (obfuscation, encryption technics,...) that we use in our framework for automated analyse of malware.

3 Quasi-Static Binary Analysis

3.1 Identifying Malware Concealment Behaviours

Current malware detectors are "*signature-based*". These current malware detectors are based on sound methods as, if the executable matches byte-signatures, then it guarantees the presence of the malicious behavior. They are equipped with a database of regular expressions that specify byte or instructions sequences that are considered malicious.

But the main problem is that this method is not complete. Malware writers can then use *Obfuscation* [26] to evade current detectors easily. To evade detection, hackers frequently use obfuscation to morph malware. Malware detectors that use a pattern-matching approach (such as commercial virus scanners) are susceptible to obfuscations. In other words, one can evade detections form current syntactic signatures by injecting code into malwares that preserves malicious behavior and makes the previous signature irrelevant (the regular expression would not match the modified binary). For instance, polymorphism and metamorphism are two common obfuscation techniques. A virus could morph itself by encrypting its malicious payload and decrypting it during its execution. A polymorphic virus obfuscates such decryption loop using several transformations (e.g. junk code, ...). Moreover, metamorphic viruses evade detection by changing their code in a variety of ways when they replicate.

Once a new type of malware appears, the number of derivative malwares by obfuscation increases exponentially. Malware writers can easily generate new undetected virus and then the anti-virus code has to update its signature database frequently to be able to catch the new virus the next time it appears.

3.1.1 Encryption Strategies for Infection Mechanisms, Triggers and Payloads

The malware could be encrypted, i.e. all body parts (Infection Mechanisms, Triggers and Payloads) could be first in an encrypted form to avoid detection. But it can not be entirely encrypted to be exacutable, it needs a *dencryption loop*, which decrypt the many malware

body in a specific memory location. Also the dencryption could use a (random) key which vary at any iteration. Also, it could use encryption library which contains cryptographic algorithms. The encrypted part would de very difficult to detect that is why anti-virus, malware detectors concentrate on the detection of dencryption loop.

3.1.2 Semantic Obfuscation Technics and Polymorphism

The unchanging part of an encrypted virus which randomly modify its key for each instance is the dencryption loop. To avoid detection, a *polymorphic* virus modify thier deencyption loop at each instance (a virus could easily generate billion of loop version [27]). To modify the loop, a malware use a *mutation engine* which could rewrite the loop with other semantically equivalent sequences of instructions, renaming register or memory location, reordering the sequences of instruction by an equivalente one, using unconditional jump, using interpreters, inling/outlining the body of function code, using new function calls, junk code, using treaded version, ...

3.2 Malware Invariant as Semantic Aware Signature

To be able to reason directly from unknown vulnerable binary code, one needs an intermediate C-like representation [28] of the binary in order to use our static and dynamic analysis. In [29], [30], [31], malware-detection algorithms try to incorporate instruction semantics to detect malicious behavior. Semantic properties are more difficult to morph in an automated fashion than syntactic properties. The main problem of these approaches is that they relay on semantic information too abstract e.g. *def-use* information.

Instead of dealing with regular expressions they try to match a control flow graph enriched with def-use information to the vulnerable 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 reordering operations that would redefine or use the variables present in the def-use properties).

The new strategy would be to generate quasi-static invariants directly from the specified malware code and use it as *semantic-aware* signatures that we call *malware-invariants*. Now consider a suspicius code. We would like to check if there is one assertion in the malware-invariant data base that holds in one of the reachable program states. To do so, we can use our new formal methods for invariant generation. This will complicate in a serious way the possiblity for the hacker to evade the detection using comon obfuscation technics.

Thus, for one familly of virus we would have only one semantic signature. In order to have a strong malware-invariant signature, one needs to identify self-modified behaviors, de-encryption loops and invariants left by the (de-)encryption algorithms used by the malware. We propose to use/combine and compose many static and dynamic tools to generate automatically binary invariants which are semantic-aware signatures of malwares, i.e. malware-invarant.

3.3 Automatic Generation of Malware Invariant

3.3.1 Static Analysis for Detection and Identification

We say that an analysis is static when the analysis do not run the code. Anti-virus use a data base of signatures and a *scanning* algorithms to look efficiently for several patterns at a time. Each of these patterns representing several different signatures. As we saw in the previous sections, present malware writter evades such pattern-matching technics.

To be able to reason directly from unknown vulnerable binary code, one needs an intermediate binary representation expressed in a logic suitable for our Invariant Generation and model construction methods. We would express the semantics of binary instructions in a C-like notation.

Example 1 *Intermediate representation.*

```
2 ...                    //...
```

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

The right most column shows the semantics of each x86 Executable instruc- tions using a C-like notation.

We could generate static invariants directly from the specified malware code and use it as a semantic-aware signature that we call Malware- Invariants. Those malware-invariant could be computed directly using ours invariant generations together with all other invariants computed using technics from a different nature and provided by tools connected into a communicational framework. We will see that one could ease their computation using a combination of other adapted invariant generation methods and tools. We are able to generates invariant express in several possible logic:

- Non-linear loop invariant with inequality which are assertions, non-linear formulae over inter-relationship values of registers, memory locations, variables, system call attributes. Consider the following piece of code obtain from our intermediate representation transformation. Using our methods we were able to generate the following invariant: $u_0(1-u_0)*eax*ebx*ecx^2+eax*ebx*ecx*R_1+eax*ebx*R_1^2-eax*ebx*ecx-2eax*ebx*R_1+eax*ebx*=0$

```

// initialization
 ...
 int_ u_0;
 ...
((M > 0)&&( Z = 1)&&( U = u_0 )...)
 ...
While ((eax >=1) || (ebx >= z_0 )){
 ...

If(Y > M){
eax = ebx / (eax + ebx);
ebx = eax / (eax + 2 * eax);
}

Else {
ecx = ecx * (R_1 + 1);
R_1 = R_1^2;
}
}
...
```

- Linear logic with uninterpreted function (in order to handle system and function calls)and inequality.
- Heap logic, to generates heap invariant and aliasing.
- We could cite here all logic which has an associated invariant generation technics

The main contribution here is that most of all obfuscation technics presented in section **??** will not change the computed invariants.

3.3.2 Quasi-Static Malware Analysis

On the other hand, we propose a new method that allows computation of exact invariants using likely invariants, properties that are true on all execution trace observed in a training period. We could then generate likely invariants (property true at any program points in the observed execution trace) using Test methods.

Then we could turn likely invariants into invariants using Verification methods like assertion checkers [32]. Some of the likely-invariants computed by a Dynamic Analysis are real invariants. They hold in all possible executions of the pro- gram. For instance, using theorem provers or assertions checkers, one could check if the proposed properties during the Dynamic analysis are invariant.

These computed malware-(likely)invariants will be considered as Signatures. And we will use our adapted pushdown model checking techniques, theorems proving methods discussed in section 2 combined with program verification tools and methods. Then, using the mentioned verification methods we will be able to detect the presence of malicious behavior described by our malware invariants (e.g. if the malware-invariants describe reachable states spaces in the ver- ification process, then we would guarantee that the software contains the suspicious behavior).

Another method is used to first generate Malware invariant ϕ_{sign} as described just above and then use similar (likely) invariant generation over the vulnerable executable code being inspected to generate an invariant ψ_{Exe}. Then one could check if $\psi_{Exe} \Rightarrow \phi_{sign}$ using a theorem prover.

The malware detector method described just above is sound with respect to the signature being considered as the malicious behavior representation. As our methods of signature generation always over-approximate the exact malicious behavior, one needs to consider the case where the malware detection is spurious. In that case, one could be able to re

ne the malware invariant.

On the other hand, we would be able to describe malicious behavior using a pushdown system representation. We will call such pushdown system a Malware Pushdown System. One could use our technique to generate invariants over pushdown systems and use the same methods. Or we could

Also, we propose to combine these computed invariant with the one that are based on data analysis generated from a dencryption.

4 Guarded Monitor

Host-based intrusion detection systems monitor an application execution and report any deviation from its statically built model [33], [34], [35]. The weakness of these systems is that they often rely on overly abstracted models that reflect only the control flow structure of programs and, therefore, are subject to the so-called *mimicry attacks* [36], [37].

More data flow information of the program is necessary in order to prevent non-control-data attacks. We proposed to use automatically generated invariants to guard system calls. We were able to detect mimicry attacks by combining control flow and data flow analysis. Our model can also tackle the non-control-data flow attacks [38]. Our model is built automatically by combining control flow and data flow analysis using state-of-the-art tools for automatic generation and propagation of invariants.

These attacks are all detectable by our model because it capture in a very precise manner the semaantics of the application being protected. During the execution of the application, the associated model simulation will detect any *mimicry* and *non-control-data* as they inevitably violate an invariant specified in the model. We use invariant generation and assertion checking technics to build our model which could be describe as a control flow graph where each program control point is annotated with a set of predicates and logic assertions that has to remain true at that location. It is an *abstract state graph* that captures both the control and the data flow properties of a program.

We use these statically built model to *monitor* an application execution and report any deviation from it (i.e. we report any behaviors that are not possible to simulate by our model).

These monitors are Visibly Pushdown Automaton [39]. In our monitor, invariants are checked through the image of the process before any system call. As we discuss in [14], our method is much more secure while the overhead is reduced drastically compared to Dyck models [40] (which needs to add null system calls at any function call site). In Table **??**, we could find some real world attacks, which avoids state-of-the arts host based intrusion detectors that was detected automatically by our built models (monitors).

We propose the same formal methods for detection of Worm by generation of malware invariant from the input and output behavior at the network perimeter.

5 Vulnerability Auditing

The problem of generation automatically vulnerability of a binary can be reduce to the problem of reachability in program verification. Basically, any program verificatin methods checking for safety properties in source code could directly be applied once the binary code has been interpreted in a higher language (here we don't need decompilation, we just need an intermediate representation for the extraction of the model). As for any decision procedure that could be encoutered during our static analysis, we would choose decision procedures based on Sat Modulo theory tools.

6 Conclusion

The new strategy propose to generate invariants directly from the specified malware code and use it as *semantic-aware* signatures that we call *malware-invariants*.

Consider a suspicius code, we could check if there is one assertion in the malware-invariant

data base that holds in one of the reachable program states. To do so, we can use our new formal methods combined with the classical ones that served for program verification. This will complicate in a serious way the possiblity for the hacker to evade the detection using comon obfuscation technics. Thus, for one familly of virus we would have only one semantic aware signature.

In order to have a strong malware-invariant signature, one needs to identify self-modified behaviors, de-encryption loops and invariants left by the (de-)encryption algorithms used by the malware.

We propose to use/combine and compose many static and dynamic tools to generate automatically binary invariants which are semantic-aware signatures of malwares, i.e. malware-invarant. Any intrusion detection system and mawalre analysis will be weakened by the absence of such invariants.

Acknowledgments

The authors would like to thank the reviewers for their comments and suggestions.

References

[1] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, 1986.

[2] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conf. Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, NY, 1977, pp. 238–252.

[3] ——, "Abstract interpretation and application to logic programs," *Journal of Logic Programming*, vol. 13, no. 2–3, pp. 103–179, 1992.

[4] S. Graf and H. Saïdi, "Construction of abstract state graphs with PVS," in *Computer Aided Verification*, ser. LNCS, O. Grumberg, Ed., vol. 1254. Haifa, Israel: springer, Jun. 1997, pp. 72–83.

[5] M. A. Colon, T. E. Uribe, M. A. Col'on, and T. E. Uribe, "Generating finite-state abstractions of reactive systems using decision procedures," in *in Computer Aided Verification*. Springer, 1998, pp. 293–304.

[6] H. Saidi and N. Shankar, "Abstract and model check while you prove," in *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*. London, UK: Springer-Verlag, 1999, pp. 443–454.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[8] H. Saidi, "Modular and incremental analysis of concurrent software systems," in *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*. Washington, DC, USA: IEEE Computer Society, 1999, p. 92.

[9] D. Baelde, A. Gacek, D. Miller, G. Nadathur, and A. Tiu, "The bedwyr system for model checking over syntactic expressions," in *CADE-21: Proceedings of the 21st international conference on Automated Deduction*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 391–397.

[10] R. W. Floyd, "Assigning meanings to programs," in *Proc. 19th Symp.Applied Mathematics*, 1967, pp. 19–37. 2.1

[11] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 1969. 2.1

[12] Z. Manna and A. Pnueli, *Temporal verification of reactive systems: safety*. New York, NY, USA: Springer-Verlag New York, Inc., 1995.

[13] I. A. Browne, Z. Manna, H. B. Sipma, Z. Manna, and H. B. Sipma, "Generalized temporal verification diagrams," in *In 15th Conference on the Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 1995, pp. 726–765.

[14] R. Rebiha and H. Saidi, "Quasi-static binary analysis: Guarded model for intrusion detection," in *TR-USI-SRI-11-2006*, Nov. 2006. 4

[15] S. Katz and Z. Manna, "A heuristic approach to program verification," in *In the Third International Joint Conference on Artificial Intelligence*, Stanford, CA, 1973, pp. 500–512.

[16] S. M. German, "A program verifier that generates inductive assertions," in *Technical Report TR*, Center for Research in Computing Technology, Harvard U., 1974, pp. 19–74.

[17] B. Wegbreit, "The synthesis of loop predicates," *Commun. ACM*, vol. 17, no. 2, pp. 102–113, 1974.

[18] S. Bensalem, Y. Lakhnech, and H. Saidi, "Powerful techniques for the automatic generation of invariants," in *Proc. of the 8th Int. Conf. on Computer Aided Verification CAV*, Rajeev Alur and Thomas A. Henzinger, Eds., vol. 1102, NJ, USA, 1996, pp. 323–335. [Online]. Available: citeseer.ist.psu.edu/bensalem96powerful.html

[19] M. Karr, "Affine relationships among variables of a program," *Acta Inf.*, vol. 6, pp. 133–151, 1976.

[20] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Tucson, Arizona: ACM Press, New York, NY, 1978, pp. 84–97.

[21] N. Bjorner, A. Browne, and Z. Manna, "Automatic generation of invariants and intermediate assertions," *Theor. Comput. Sci.*, vol. 173, no. 1, pp. 49–87, 1997.

[22] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar, "A technique for invariant generation," in *TACAS: Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2001.

[23] G. McGraw and G. Morrisett, "Attacking malicious code: A report to the infosec research council," *IEEE Software*, vol. 17, no. 5, pp. 33–41, /2000. [Online]. Available: citeseer.ist.psu.edu/mcgraw00attacking.html 2.2

[24] L. M. Adelman, "An abstract theory of computer viruses," in *Advances in Cryptology CRYPTO'88*, 1988. 2.2

[25] F. Cohen, "A short courses on computer viruses," in *Wiley,*, 1990. 2.2

[26] C. Nachenberg, "Computer virus-antivirus co-evolution," *Commun. ACM*, vol. 40, no. 1, pp. 46–51, 1997. 1, 3.1

[27] C. Fisher, "Tremor analysis(pc)," in *Virus Diget, 6(88)*, 1993. 3.1.2

[28] D. Gopan and T. W. Reps, "Low-level library analysis and summarization." in *CAV*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 68–81. [Online]. Available: http://dblp.uni-trier.de/db/conf/cav/cav2007.html#GopanR07 3.2

[29] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*. Oakland, CA, USA: ACM Press, May 2005, pp. 32–46. 3.2

[30] C. M., J. S., and K. 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. 3.2

[31] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray, "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, pp. 377–388. 3.2

[32] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast: Applications to software engineering," in *Journal on Software Tools for Technology Transfer (paper from FASE2005)*, 2007. 3.3.2

[33] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *IEEE Symposium on Security and Privacy*, 2001, pp. 156–169. 4

[34] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, "Formalizing sensitivity in static analysis for intrusion detection," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2004, p. 194. [Online]. Available: http://csdl.computer.org/comp/proceedings/sp/2004/2136/00/21360194abs.htm 4

[35] R. Gopalakrishna, E. H. Spafford, and J. Vitek, "Efficient intrusion detection using automaton inlining," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2005, pp. 18–31. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/SP.2005.1 4

[36] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, R. Sandhu, Ed. Washington, DC, USA: ACM Press, Nov. 2002. [Online]. Available: http://www.cs.berkeley.edu/~daw/papers/mimicry.pdf,http://www.cs.berkeley.edu/~daw/talks/CCS02.ppt,http://www.cs.berkeley.edu/~daw/talks/CCS02.ps(slides) 4

[37] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *Proceedings of the 14th USENIX Security Symposium*, 2005. 4

[38] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. nkar K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Security Symposium*, 2005. 4

[39] R. Alur and P. Madhusudan, "Visibly pushdown languages," in *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 2004, pp. 202–211. 4

[40] J. T. Giffin, S. Jha, and B. P. Miller, "Automated discovery of mimicry attacks." in *RAID*, ser. Lecture Notes in Computer Science, D. Zamboni and C. Krgel, Eds., vol. 4219. Springer, 2006, pp. 41–60. 4