Proceedings of the 2003 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY June 2003

Static Verification of Worm and Virus Behavior in Binary Executables using Model Checking

Prabhat K Singh and Arun Lakhotia
The Center for Advanced Computer Studies,
University of Louisiana, Lafayette, LA-70504
(337) 482-6766, -5791 (Fax)
{pks3539, arun}@cacs.louisiana.edu,

Abstract – Use of formal methods in any application scenario requires a precise characterization and representation of the properties that need to be verified. The target, which is desired to be verified for these properties, needs to be abstracted in a suitable form that can be fed to a mechanical theorem prover. The most challenging question that arises in the case of malicious code is "What are the properties that need to be proved?" We provide a decomposition of virus and worm programs based on their core functional components and a method of formally encoding and verifying functional behavior to detect malicious behavior in binary executables.

Index terms – Virus behavior, decompilation, verification, model checking, modeling language, flow graphs

I. Introduction

The high cost of virus and worm infections may be attributed to the highly interconnected nature of today's computers and the reactive nature of anti-virus (AV) technologies. A virus or worm, if undetected, can spread rapidly across the world due to high interconnectivity. Current AV technologies still rely on varying forms of signature-based fingerprinting to characterize a specific virus. Thus a signature database update is required at the end-user machines frequently or during a new virus event. Even a day of delay in the virus analysis and a signature update can be quite expensive as evident from virus timeline reports published. This calls for an approach to fast verification of programs, which does not need to be updated frequently for signatures.

Model checking of software programs has been gaining increased use in program verification tasks, the reason being that it provides sound verification of a property in a given program. Contemporary model checking approaches to verification of security properties require the availability of source code of the program under verification [1]. The verification of security properties in binary executables is problematic mainly due to two reasons:

I. The malicious properties of viruses and worms have to be identified and precisely encoded into a suitable logic formula using predicates that are representative of a particular action by the program. Generation or extraction of such predicates from a binary program is a challenging model checking problem.
II. A virus writer can apply obfuscating transformations using hand written assembly and thus make the process of flow analysis difficult and less reliable.

While studying the virus and worm source code as part of this work, it has been a frequent observation that remarkably different virus source codes (even those which were implemented in different languages) displayed identical *operational* behavior. The front-ends of the publicly available virus generator programs provide the user with a matrix of features to be implemented in the virus. Thus, part of our research has been centered on studying these features with an increased granularity so that we can come out with a detailed property characterization of viruses and worms using formal specification [2].

It has previously been argued that computer viruses are artificial life forms, performing similar functions as biological life forms [3]. Considering this argument as the premise we carry the analogy with artificial life forms further by identifying and studying the *functional organs* of virus and worm programs. The organs are *functional* in sense that they are defined constituents that make up a worm. Unlike a biological life form, the organs of a computer virus may not be physically distinguishable from the rest of the *body*. In fact, the code corresponding to an organ function may be dispersed and interleaved with the code for other computations.

II. Our Approach

Our approach to verifying virus and worm binaries uses a combination of techniques from the reverse engineering and model checking domains. A malicious program behavior is characterized using *predicates*. A predicate is a Boolean outcome of abstract *action* present in a worm or virus program. An action is a sequence of one or more function/system calls, in a program, connected through a flow relationship. We follow a goal directed approach to binary program decompilation where in the final results of

ISBN 0-7803-7808-3/03/$17.00

decompilation are the predicates that can be used to represent the property and the program model of the given binary using the model checker's language. The program properties to be verified are manually formulated using these predicates.

Classifying Computer Worms and Virus behavior

We use the phrase "organ" to imply a functional organ. Each organ function is achieved by the application of an *action* procedure by a *subject* on an *object* within the

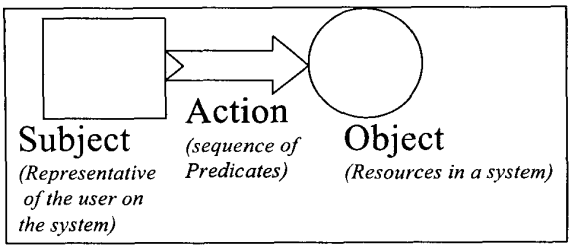

Figure 1: The worm organ abstraction

system. We have identified five functions that are sufficient in describing the internal working, and hence to capture malicious properties, of a virus. These functions are *Survey, Concealment, Propagation, Injection, Self-identification.* Our experience with manual analysis of viruses suggests that these functions are sufficient to describe the behavior of malicious programs.

The Model Checking Process

Model Checking [4] is an automated technique that, given a finite-state model of a system and a logical property, systematically checks whether this property holds for a given initial state in that model.

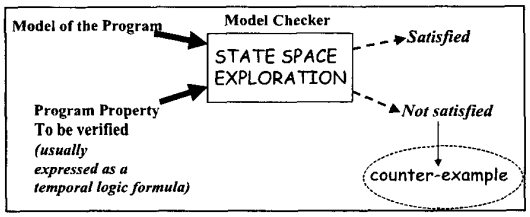

Figure 2: Model checking of programs

Encoding malicious behavior using Linear Temporal Logic

In our system, we encode malicious behavior of viruses and worms using linear temporal logic (LTL) [5] and the predicates defined during the dataflow analysis phase. We have chosen linear temporal logic since it are very expressive and allows encoding temporal ordering of *security sensitive* events that occur during the worm's execution. The LTL formula encodes the set of executions that characterize a worm program behavior. An example malicious behavior may involve the read and transfer of information from a system to another system. The information may comprise of the malicious code itself and the information about other targets that trust the system on which the worm is executing. Thus, this activity can be viewed as an ordering of calls involving the worm code's read action followed by worm code's send action on the network.

III. Implementation

We have developed a prototype worm/virus verifier to illustrate our approach. The malicious code verifier uses the SPIN model checker [6] to statically verify binary executables against property formula of worms and viruses. It takes as input the binary executable and a set of one or more behavioral properties that the program needs to be verified for. If a worm behavior is detected, the trace of the execution path that confirms to the property under test is returned by the prototype. This is returned in the form of a counter-example generated by the model checker.

Generating Virus and Worm Models for Model Checking

We use the Spin model checker for verification. During the process of generating the control flow graph it is annotated with the predicates that were extracted during the data-flow analysis phase. The flow graph also includes information about all imported DLL functions in the form of boolean variables. The control flow graph is translated into Promela, the modeling language for Spin. Conditional branch instructions in the basic block are translated to their Promela equivalent to indicate a non-deterministic choice. This means that the model checker will explore all the branches at a conditional branch instruction, during the verification phase.

The prototype was built as a plugin to the IDAPro disassembler [7]. The control flow graph generation and the data-flow analysis are done using the methodology presented in [8, 9]. The automatic recognition of the C and C++ library functions is achieved through IDAPro's fast library identification and recognition mechanism. Currently, the behavioral properties (LTL formula) are manually fed to the prototype.

IV. Summary

We identify the organs of virus programs in an attempt to characterize malicious behavior in worms using formal specification. We presented a method of encoding malicious behavior using linear temporal logic. The given binary program was translated to a finite model representation, which was then fed to a model checker for verification. The proposed method of representing malicious code is beneficial since it semantically captures the presence of malicious behavior and any ordering between malicious actions by a program. While statically verifying the presence of malice in programs, all possible

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE

execution paths are explored for the verifying the presence of some property, this approach helps in detecting viruses or worms that execute a malicious action only at a certain time or day.

V. REFERENCES.

[1] Hao Chen and David Wagner, "MOPS: An Infrastructure for Examining Security Properties of Software," *Proceedings of the 9th ACM Conference on Computer and Communication Security*. Washington, DC. November 17-21, 2002.

[2] Prabhat K Singh, "A Physiological Decomposition of Virus and Worm Programs," Master Thesis, CACS, University of Louisiana, Lafayette, May 2002.

[3] I. H. Witten, H. W. Thimbleby, G. F. Coulouris, and S. Greenberg, "Liveware: A new approach to sharing data in social networks," *International Journal of Man-Machine Studies*, 1990.

[4] E. M. Clarke et al, "Model Checking,", MIT Press, ISBN: 0262032708.

[5] Amir Pnueli. "The Temporal Logic of Programs," *Proc. 18th IEEE Symp. Foundations of Computer Science*, Providence, Rhode Island, pp. 46-57, 1977.

[6] Gerard J. Holzmann, "The Model Checker Spin," IEEE Transactions on Software Engineering, Vol. 23 No. 5, May 1997.

[7] Ilfak Guilfanov, "IDA Pro disassembler," http://www.datarescue.com/index.htm

[8] C. Cifuentes, "Reverse compilation techniques," PhD dissertation, Queensland University of technology, 1994.

[9] A. Aho, R. Sethi and J Ullman, "Compilers-Principles, Techniques, and Tools," Reading, MA:Addison-Wesley, 1986.

ISBN 0-7803-7808-3/03/$17.00 © 2003 IEEE