Detecting Malicious Code by Model Checking

Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart,
and Helmut Veith

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

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

Keywords: Model Checking, Malware Detection.

1 Introduction

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

In contrast to the viruses of the pre-Internet era, creating an e-mail worm that infects hundreds of thousands of computers nowadays does not require knowledge of systems or even assembly language programming. For example, *NetSky* and *MyDoom* were written in Visual C++, do not appear to be very

K. Julisch and C. Kruegel (Eds.): DIMVA 2005, LNCS 3548, pp. 174–187, 2005.

skillfully engineered and contain obvious bugs in some of the versions. This trend is further intensified by the availability of virus toolkits which allow unskilled persons to create a new virus with a few mouse clicks.

During the last years it became evident that shortly after a new worm is released into the wild, several modified versions of the worm appear (either written by the same author or by individuals who somehow got hold of the source code). As a result of these developments, we see new worm derivatives appearing on the Internet almost every day. While these new versions differ only slightly from the original in terms of functionality, the resulting binary file can be quite different, depending on the compiler in use and its optimization settings; this problem worsens if *executable packers* such as UPX [15] or FSG [9] are used.

Current anti-virus products use rather straightforward (but yet computationally efficient) detection methods, most notably static signature matching and, more recently, dynamic analysis [1]. Static signature matching employs a database containing characteristic binary code sequences of known malware and matches these sequences against executables. Dynamic analysis executes the potentially infected programs in a controlled environment (sandbox) and checks for suspicious program behavior at runtime. These two approaches have the following two substantial drawbacks:

- Signature matching requires an up-to-date database of characteristic viral code sequences. In order to keep the false positives rate of the virus detector low, signatures are chosen so that one signature exactly matches one version of a virus or worm. In particular, the signature will thus not match against worm derivatives. This hypothesis was certified by Christodorescu and Jha in tests with commercially available virus scanners [4]; their tests showed that even naive modifications of the viral code, such as the insertion of a single `nop` instruction, can totally foil the detection process. Typically, modified worms spread quickly, which leads to a window of vulnerability between the release of a worm variant and the next update of the signature libraries. In this time span a novel virus or worm derivative cannot be detected by conventional anti-virus products. It would thus be highly desirable to have a virus scanner that reliably detects a virus or worm together with a large class of its potential derivatives.
- On the other hand, while dynamic analysis promises to solve some of the problems of static signature matching, it can be foiled by appropriate virus design. In particular the behavior of an executable is observed only over a limited timespan, which does not allow predictions of future malicious actions.

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

Bergeron et. al. [2] concentrate on the detection of suspicious system call sequences. In particular, they reduce the control flow graph of an executable to a subgraph containing only the nodes representing certain system calls and check