Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Static analysis based invariant detection for commodity operating systems

Feng Zhu[*], *Jinpeng Wei*

School of Computing and Information Science, Florida International University, Miami, FL 33199, USA

ARTICLE INFO

Article history:
Received 20 August 2013
Received in revised form
24 January 2014
Accepted 20 February 2014

Keywords:
Integrity modeling
Invariants detection
Malware detection
Static analysis
Tools

ABSTRACT

Recent interest in runtime attestation requires modeling of a program's runtime behavior to formulate its integrity properties. In this paper, we study the possibility of employing static source code analysis to derive integrity models of a commodity operating systems kernel. We develop a precise and static analysis-based *data invariant* detection tool that overcomes several technical challenges: field-sensitivity, array-sensitivity, and pointer analysis. We apply our tool to Linux kernel 2.4.32 and Windows Research Kernel (WRK). For Linux kernel 2.4.32, our tool identifies 284,471 data invariants that are critical to its runtime integrity, e.g., we use them to detect ten real-world Linux rootkits. Furthermore, comparison with the result of a dynamic invariant detector reveals 17,182 variables that can cause false alarms for the dynamic detector in the constant invariants category. Our tool also works successfully for WRK and reports 202,992 invariants, which we use to detect nine real-world Windows malware and one synthetic Windows malware. When compared with a dynamic invariant detector, we see similar results in terms of false alarms. Our experience suggests that static analysis is a viable option for automated integrity property derivation, and it can have very low false positive rate and very low false negative rate (e.g., for the constant invariants of WRK, the false positive rate is one out of 100,822 and the false negative rate is 0.007% or seven out of 100,822).

Introduction

Remote attestation is a security mechanism that a party in a distributed environment can employ to determine whether a target computer has the appropriate hardware/software stack and configuration, so it can be trusted (i.e., it has integrity). The idea of remote attestation has been widely adopted. For example, the trusted platform modules (Trusted Platform Modules) chip has become a standard component in modern computers.

Remote attestation has evolved from static attestation to runtime attestation. Traditional remote attestation techniques only ensure that a computer is bootstrapped from trusted hardware and software (e.g., operating systems and libraries), but there has been a consensus in recent years that such static attestations are inadequate (Kil et al., 2009; Loscocco et al., October 2007). This is because runtime attacks such as buffer overflow can invalidate the result of static attestation during the execution of the target system, so a remote challenger cannot gain high confidence in a target system even if it is statically attested (Kil et al., 2009). In order to regain high confidence, the challenger must enhance traditional remote attestation with runtime attestation, or runtime integrity checking.

[*] *Corresponding author*. Tel.: +1 3054952306.
E-mail addresses: fzhu001@fiu.edu, zzztemple@126.com (F. Zhu), weijp@cis.fiu.edu (J. Wei).
http://dx.doi.org/10.1016/j.cose.2014.02.008
0167-4048/© 2014 Elsevier Ltd. All rights reserved.

One of the determining factors of the effectiveness of runtime attestation is the attestation criteria, i.e., the expected integrity properties of the target system. Other than a few static program states (e.g., code segments and constant data), most of the runtime state of a system (normal variables, stack, and heap) cannot be trivially characterized. This uncertainty about the criteria results in two classic attestation errors: false positives and false negatives. False positives happen when the remote challenger endorses an overly stringent criterion that even an uncompromised system fails to meet; and false negatives happen when the challenger endorses an overly loose criterion that a compromised system can also meet (i.e., the remote challenger ends up trusting a corrupted computer). Obviously, both kinds of errors are undesirable for remote attestation.

The root cause for the above attestation errors is the lack of precise specifications of expected integrity properties. While *under-specification* can reduce the rate of false positives by lowering the bar for a target system, it allows a compromised system to obtain trust. On the other hand, *over-specification* errs on the side of safety to ensure that no compromised system can pass the integrity check, but it may raise too many false alarms.

Since integrity properties are attributes of the target system, a precise specification demands a thorough analysis of the target system. Researchers have taken several kinds of approaches to analyze a target system for its integrity properties. Manual analysis relies on domain expertise to specify and prove the correctness of integrity properties. It is applicable to well-understood properties such as the immutability of the Interrupt Descriptor Table (IDT), but it is not scalable to complex software such as the Linux kernel. Therefore, automated tools are much desired to assist a human expert. Dynamic analysis tools such as Gibraltar (Baliga et al., 2008) and ReDAS (Kil et al., 2009) infer likely integrity properties (called invariants) of a system by reading the runtime states (e.g., memory snapshots that contain program variables) of the target system and hypothesizing whether some variables satisfy predefined invariant relationships. One example relationship is that a variable v must always have a constant value k at runtime. However, it is well-known that dynamic analysis has difficulty in exploring all possible program execution paths, so it may generate false invariants. For example, Gibraltar generates about 4673 false invariants for Linux kernel 2.4.20 (Baliga et al., 2008). A typical solution to overcome such shortcomings is to use a large set of test cases (e.g., ReDAS created 70 training scenarios and 13,000 training sessions for the *ghttpd* server). However, how to systematically generate a large number of test cases that can trigger all execution paths in a program remains a challenging research problem in general.

In this paper, we explore the applicability of static analysis for finding integrity properties. The basic idea is to use compiler technology to analyze the behavior of a program to derive its integrity properties, without actually running the program. Static analysis can overcome the limitations of dynamic analysis by exploring all execution paths. For example, if $v = v+2$ is found in the true or false branch of a conditional statement in the target program, then the property that "variable v always has a constant value at runtime" is likely false. However, a dynamic analysis tool cannot observe this assignment if the test cases do not satisfy the condition for the assignment; as a result, a dynamic analysis tool may conclude that v is a constant. Since static analysis has the source code of the program, it has the advantage to reveal all conditions for assignments to a variable, so it can be more precise.

Specifically, we focus on the static detection of one class of integrity properties called *data invariants*. These invariants include properties of global data structures and serve as specifications of data structure integrity (Baliga et al., 2008). For example, they can represent critical system integrity properties such as the immutability of the Interrupt Descriptor Table (IDT) and the system call table. Therefore, they have been checked by state-of-the-art integrity monitors (Kil et al., 2009; Baliga et al., 2008).

Our first contribution is a program analysis tool that can assist a human expert by automatically deriving data invariants from source code, using static analysis. Our tool applies compiler technology to analyze the control and data flows (e.g., assignments, function calls, and conditional statements) of a target program and hypothesizes likely invariants (e.g., constant, membership, bounds, and non-zero). In developing this tool, we have overcome several challenges in large-scale C program analysis, such as field-sensitivity, array-sensitivity, and pointer analysis.

Our second contribution is a thorough study of invariants detection for the Linux kernel and the Windows Research Kernel (Microsoft, Jan 5, 2014) using static analysis. To the best of our knowledge, there has not been a similar study. Both kernels are very complex software posing great challenges for static analysis by their wide use of pointers and complex structures. Our tool is able to process 400,492 lines of Linux kernel (version 2.4.32) code and identify 284,471 invariants essential to the Linux kernel's runtime integrity. To validate the result of our tool (e.g., precision), we develop a dynamic invariant detector (following the spirit of Daikon (Ernst et al., 2007)) and compare it with our static analyzer. The comparison suggests that static invariant detection outperforms dynamic invariant detection in terms of false positives. For example, in the constant invariants category, we find 17,182 variables that can cause false alarms for the dynamic analyzer, while our static tool only misses 18 true invariants (with false negative rate 0.013%). We also develop an invariant monitor based on the result of the static analysis, which detects invariant violations by ten real-world Linux rootkits and generates only one false alarm against benign workloads. Moreover, although Windows Research kernel (version Windows Server 2003), which has 665,969 lines of C code, is even more complex than the Linux kernel, our tool analyzes it successfully and detects 202,992 invariants. Comparison with the result of a dynamic analyzer shows that the dynamic analyzer generates 21,670 false constant invariants while our static tool wrongly classifies only seven true constant invariants as non-constants (with false negative rate 0.007%). We develop an invariant monitor in the same way as the Linux kernel and this monitor successfully detects nine real-world Windows malware samples and one synthetic Windows malware, while emitting only one false alarm. Our study shows that our static tool is