Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence *†

Chongkyung Kil[§], Emre C. Sezer[§], Ahmed M. Azab[§], Peng Ning[§], Xiaolan Zhang[††]

[§] Department of Computer Science
NC State University, Raleigh, NC, 27695
{ckil, ecsezer, amazab, pning}@ncsu.edu

[††] IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532
cxzhang@us.ibm.com

Abstract

Remote attestation of system integrity is an essential part of trusted computing. However, current remote attestation techniques only provide integrity proofs of static properties of the system. To address this problem we present a novel remote dynamic attestation system named ReDAS (Remote Dynamic Attestation System) that provides integrity evidence for dynamic system properties. Such dynamic system properties represent the runtime behavior of the attested system, and enable an attester to prove its runtime integrity to a remote party. ReDAS currently provides two types of dynamic system properties for running applications: structural integrity *and* global data integrity. *In this work, we present the challenges of remote dynamic attestation, provide an in-depth security analysis and introduce a first step towards providing a complete runtime dynamic attestation framework. Our prototype implementation and evaluation with real-world applications show that we can improve on current static attestation techniques with an average performance overhead of 8%.*

Keywords: *Remote attestation, dynamic attestation, runtime integrity, system security, trusted computing.*

1 Introduction

The goal of remote attestation is to enable a computer to prove the integrity of the platform to a remote party. The integrity information allows the remote party to verify the configuration and current state of the platform (e.g., hardware and software stack), and decide whether the attested platform should be trusted. Thus, remote attestation plays an important role in building trust and improving trustworthiness in distributed computing environments. For example, if an integrity violation is detected, the violating component can be distrusted by others in the system. Furthermore, the computer running this component can be excluded from the system to prevent possible error propagation, or to facilitate forensic investigation if it is believed compromised.

Current remote attestation techniques provide integrity evidence of an *attester* to a remote *challenger* by hashing static memory regions, sometimes combined with language specific properties or program input/output [11, 12, 20, 23, 24, 26, 29, 30, 32]. However, all existing attestation approaches focus on the static properties of the computer system by providing hash values of static objects as the integrity evidence to the challenger. Although this *static attestation* provides a straightforward way to attest to the integrity of static objects, it does not apply to dynamic objects in the system. (Hashing dynamic objects is not effective because their contents may change frequently.) In the presence of various runtime threats that may modify the dynamic objects (e.g., buffer overflow attacks), the remote challenger still cannot gain high confidence in a system even if it is statically attested.

In order to address this problem, we introduce the notion of *dynamic attestation*. Dynamic attestation provides the integrity evidence for the dynamic properties of a running system. The dynamic system properties are properties that dynamic objects must satisfy during their lifetime. For example, even though a stack is dynamically changing, a saved frame pointer in the stack must point to the caller's stack frame, and thus the saved frame pointers in the same stack must be linked together. In other words, dynamic system properties represent the valid runtime behavior of the attested system. Dynamic attestation can use such dynamic properties to verify the runtime integrity of the system, and provide integrity evidence to improve the confidence in system integrity.

*The first two authors have contributed equally to this paper.

†This work is supported by the U.S. Army Research Office under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI) and the U.S. National Science Foundation under grant CAREER-0447761. The contents of this paper do not necessarily reflect the position or the policies of the U.S. Government. The material contained in this paper has been cleared through all the authors' affiliations.

978-1-4244-4421-2/09/$25.00 ©2009 IEEE

There are unique challenges in developing a dynamic attestation system. First, due to the diversity in dynamic objects and their properties, it is non-trivial to identify and derive the "known" good states of the dynamic objects. The problem is exacerbated by the fact that such objects are often transient in nature. In contrast, in static attestation, the known good state can be measured by the cryptographic checksum of static objects. Second, dynamic attestation needs to access a potentially large number of dynamic objects and measure their integrity repeatedly. This requires a versatile and efficient measurement mechanism. This differs considerably from the one-time check of static attestation techniques and demands a highly efficient attestation system. Finally, dynamic attestation measures objects that the adversary may have already modified. Thus, the attester must protect itself against potential indirect attacks from the adversary as well.

In this paper, we present a novel dynamic attestation system named ReDAS (Remote Dynamic Attestation System) to address the above challenges. We propose to perform application-level dynamic attestation by attesting to the dynamic properties of running applications. ReDAS monitors running applications and secures any integrity violation evidence using hardware support. As an initial attempt in our research, ReDAS provides the integrity evidence for two dynamic properties: *structural integrity* and *global data integrity* of running applications. These properties are automatically extracted from each application. They represent a program's unique runtime characteristics that must be satisfied during its execution. Thus, attacks modifying dynamic objects related to such properties (e.g., corrupting saved frame pointers) will lead to integrity (violation) evidence, which ReDAS will capture and report during attestation.

In order to prevent an attacker from modifying the integrity evidence, we use the hardware support provided by Trusted Platform Module (TPM) [3]. ReDAS runs in the OS kernel. When it sees an attack interfacing with the OS kernel, it immediately seals the integrity evidence into the TPM, before the attack has any chance to compromise the kernel and the evidence itself. Thus, even if an attacker can compromise the system, he/she cannot modify the integrity evidence without being detected by the remote challenger.

We have implemented ReDAS as a prototype system on Linux, and performed experimental evaluation of ReDAS using real-world applications. The experimental results show that ReDAS is effective in capturing runtime integrity violations. We observe zero false alarms in our test cases and an average performance overhead of only 8%.

The contributions of this paper are as follows. First, we raise the need for dynamic attestation and propose a framework for application-level dynamic attestation. To the best of our knowledge, our approach is the first attempt to provide *remote attestation to dynamic system properties*. Second, we identify two types of dynamic properties that can be practically used , and describe how they can be generated and applied towards attestation. Third, we show how the measurement can be done in a manner that is both effective and efficient. Fourth, we propose techniques to protect the attestation system itself from corruption, employing the TPM hardware to protect the integrity evidence. Fifth, we perform a detailed security analysis of ReDAS. Finally, we implement ReDAS on Linux and perform a substantial set of experiments to evaluate the proposed techniques.

Note that this paper serves as an initial attempt for providing complete system integrity evidence through dynamic attestation, but it does not yet provide a complete solution for dynamic attestation. Several open questions remain to be addressed, such as the identification of all dynamic system properties for remote attestation and the inclusion of the kernel in dynamic attestation. Nevertheless, the results in this paper have already advanced the state of the art of attestation (beyond static attestation), and can be used as a foundation for additional research.

The remainder of the paper is organized as follows. Section 2 presents the proposed dynamic attestation techniques. Section 3 describes the prototype implementation of the proposed dynamic attestation system. Section 4 presents the experimental evaluation of our approach using the prototype system. Section 5 discusses related work. Section 6 concludes the paper and proposes some future research directions.

2 ReDAS: A Remote Dynamic Attestation System

The ultimate goal of dynamic attestation is to provide complete system integrity evidence. In this paper, we focus on application-level dynamic attestation as part of this effort. Specifically, we develop an application-level dynamic attestation system named ReDAS (Remote Dynamic Attestation System). As an initial attempt in our research, ReDAS currently provides two dynamic properties: structural integrity and global data integrity of running applications. In this section, we present our assumptions and threat model, describe the ReDAS architecture for measuring dynamic properties in user applications, and discuss some of our design choices.

2.1 Assumptions and Threat Model

At the hardware level, we assume that the attester's computer is equipped with a TPM chip and its BIOS supports Core Root of Trust Measurement (CRTM). The CRTM measures the computer's hardware configurations (e.g., ROMs, peripherals) and provides the root of trust for ReDAS. We assume that the attester's Operating System (OS) kernel is trusted at boot time. This can be ensured by using existing trusted boot techniques [19]. However, the OS kernel may

still be compromised at run time due to potential vulnerabilities that can be exploited remotely [1]. Furthermore, each application binary is measured before it is loaded into memory for execution. This can be done by using static attestation techniques [26]. As a result, any malicious changes in the application's binary before load time will be visible to the remote challenger during attestation. We assume that the application's source code is available. Source code access is required to collect dynamic properties of applications. Finally, we assume that the attester and the challenger can establish an authenticated communication channel using the cryptographic support provided by TPM [13].

In our threat model, an adversary is capable of launching remote attacks against the attester, but does not have physical access to the attester's computer. In other words, the attacker cannot launch hardware attacks (e.g., resetting the platform configuration registers (PCRs) in the TPM without rebooting the computer [34]). Even though the adversary can gain full control (e.g., root) over the attester's system by exploiting application level vulnerabilities, we assume that ReDAS detects such attacks before the kernel is compromised and takes the necessary actions (e.g., extend the integrity evidence into a PCR in the TPM).

2.2 ReDAS Overview

As discussed earlier, we propose to use dynamic system properties for dynamic attestation. In the attestation process, the remote challenger is expected to send an attestation request to the dynamic attestation service running on the attester's system. The dynamic attestation service then provides the integrity evidence of the attester's system to respond to the challenger.

In order to provide dynamic attestation and the required integrity evidence, we propose to have three components in ReDAS: 1) *dynamic property collector*, 2) *integrity measurement component*, and 3) *attestation service*. Figure 1 illustrates the ReDAS architecture.

The dynamic property collector is an off-line component that collects dynamic properties from the target applications' source code and binary. The integrity measurement component and the attestation service form the *ReDAS runtime system* (shown in the dashed box in Figure 1). The runtime system takes the dynamic properties extracted by the dynamic property collector, and runs on the attester to perform integrity measurement and provide dynamic attestation service.

2.3 Dynamic System Properties

Dynamic system properties are the properties that dynamic objects in the system must satisfy during their lifetime. Finding a complete set of dynamic system properties is a non-trivial task due to the diversity in dynamic objects and their properties. As an initial effort, we identify two types of dynamic properties that can be practically used in dynamic attestation: *structural integrity* and *global data integrity*. Although other simple dynamic properties (e.g, the canary word in stack [9]) can be integrated into ReDAS easily, the challenge is to find a *complete* set of dynamic properties that can be used to precisely measure the system integrity and can also be verified *efficiently* at runtime. This paper explores the feasibility of a dynamic attestation system that is both effective (can detect dynamic attacks) and efficient (low performance impact). Even though ReDAS does not guarantee to examine the complete set of dynamic properties, our results (see Section 4) demonstrate that it is able to detect the runtime attacks in our test cases with zero false positives and minimal performance overhead. We will investigate the feasibility of finding the complete set of dynamic properties in our future work.

2.3.1 Structural Integrity

It is observed in [16] that even though memory objects in the writable sections of a running application are dynamically modified, they still follow certain patterns, particularly when the application binary is generated by a modern compiler such as GCC. For example, the return address of a function in the (dynamic) stack must point to the instruction following the `call` instruction. As another example, memory chunks allocated in the heap must be linked together if they are allocated via `malloc` family functions. Such patterns are referred to as *structural constraints* [16]. We say that an application (execution) has *structural integrity* if all its structural constraints are satisfied at runtime. The full list of structural constraints used in ReDAS can be found in [16].

Structural constraints are obtained by static analysis of the program binary based on the knowledge of how the program binary is generated. They do not produce false alarms, and thus can be safely used in integrity measurement.

In order to collect structural constraints from each application, the dynamic property collector analyzes the application's binary code, and extracts the set of structural constraints of the application. The exact techniques for extracting structural constraints are presented in [16]; we do not repeat them here.

2.3.2 Global Data Integrity

Global variables are another type of dynamic objects in a running application. Although their values may change, there are properties they must satisfy during program execution. Such properties are called *data invariants* [10]. Data invariants are either values of data variables or relations among them that must be satisfied at certain program points. For example, the constant invariant is an actual numeric value that the variable should hold. As another example, the equality invariant means that two data variables must have the same value at that program point. An application (execution) has

[1] We plan to investigate how to attest to the runtime kernel's integrity in future work; it is out of the scope of this paper.

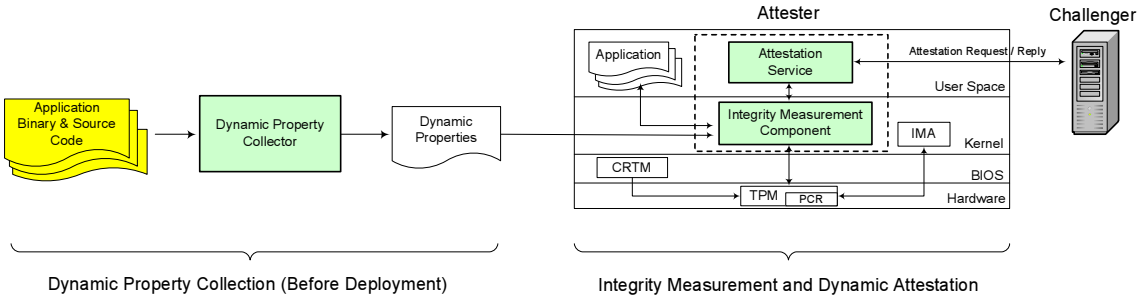

Figure 1. ReDAS architecture

global data integrity if all its data invariants involving global variables are satisfied. The full list of data invariants can be found in [10].

Daikon [10] was developed to collect data invariants through a training phase, where it runs the program multiple times to generate data traces from which the data invariants are extracted. However, we cannot use Daikon's data invariants directly for several reasons. First and foremost, Daikon produces data invariants at function entries and exits. We modify Daikon to produce invariants at system call times, the details of which are presented in section 3.1. Second, Daikon's data invariants produce false alarms which cannot be tolerated in an attestation server. One reason for these false positives is due to the difficulty of defining or verifying some types of data invariants and the second is due to inadequate training [35]. The rest of this section describes our efforts towards eliminating these false positives.

Choosing the Right Invariants. Daikon produces data invariants for all types of variables; however, we use data invariants of only global variables. While structural constraints provide a means for protecting the stack and heap metadata, the global data invariants provide protection for the data segment of an application's virtual address space. We further reduce the number of invariants by selectively choosing those which have been shown to work successfully without many false positives. For example, we removed string invariants as they produce many false positives despite previous attempts at eliminating these false positives [14].

Ensuring the Quality of Daikon Training Phase. In general, good quality training is the result of using a good set of training data. In order to develop a good set of training data for each program, we first identify various uses of the program along with different options and configurations that the program provides. We then generate a set of training scenarios that triggers various program uses and more importantly, ensures the full coverage of the program's source code [2]. To check whether the set of training scenarios covers the complete source code of the program, we use a test coverage tool named `Gcov` [2]. `Gcov` allows us to inspect how much of the program is executed by the input. We revise the training scenarios based on `Gcov`'s result until they completely cover the program's source code. Each training scenario can be further configured with different parameters. After we obtain a set of training scenarios, we change these parameters to generate various training inputs for each scenario. For example, we created 70 different training scenarios for the `ghttpd` web server. After further configuring the training parameters, we obtained 13,000 training sessions.

[2]Note that we aim for full coverage of the source code, rather than full coverage of all execution paths. The latter is equivalent to solving the halting problem.

2.4 Measuring Dynamic System Properties

The integrity measurement component monitors applications at runtime and checks their integrity according to the system properties collected by the dynamic property collector. We propose to include the integrity measurement component in the OS kernel as a loadable kernel module. This allows the integrity measurement component to examine the dynamic objects in various applications. More importantly, it provides isolation between the integrity measurement component and the target applications.

Measurement Points. A critical decision is when to measure the dynamic system properties. In general, fine-grained verification may capture more integrity violations, but also introduces higher performance overhead. On the other hand, coarse-grained verification may miss some integrity violations. There is clearly a trade-off between the quality of integrity measurement and the performance overhead.

We monitor applications at every system call. There are several good reasons for this. First, as we have mentioned earlier, this is the minimum granularity that we require to guarantee that the integrity evidence can be protected via hardware support (e.g., extended into a PCR) before it is corrupted. Second, system calls are the means by which an application interacts with the OS kernel, and are also the entry points for many attacks which compromise the underlying system. Finally, it provides a good balance between effectiveness and performance.

Integrity measurement at system call times allows us to verify data invariants and the stack structural constraints. However, they are unsuitable to verify the heap structural constraints, since the structure of the heap is only modified when a memory block is added or deleted in the heap. We

propose to use `malloc` family function call times to check the heap structural constraints. In addition, the dynamic objects in the constructor and destructor sections are only used when the `main` function starts and after it finishes. Thus, we only need to verify them before the `main` function enters/exits.

2.5 Protection of Integrity Evidence

As long as an attester is online to provide network-based services to others, it is subject to remote attacks, which may exploit (both known and unknown) vulnerabilities to compromise the system. A determined attacker may be able to compromise the runtime kernel. Even if ReDAS can detect the integrity violation caused by the attacker, after compromising the kernel, the attacker may attempt to modify the integrity evidence maintained by ReDAS, which is also stored in the kernel. We investigate how to protect the integrity evidence so that even if an attack is successful, the attacker still cannot erase its mark from the integrity evidence without being detected.

To achieve this goal, we propose to take advantage of the hardware support provided by the TPM. The TPM contains Platform Configuration Registers (PCRs), which provide sealed storage for platform configuration measurements. These measurements are normally hash values of entities (e.g., programs, configuration data) on the platform. A subset of PCRs, which are called *static* PCRs (number 0–15), cannot be reset at runtime. Instead, they can only be extended with new measurement values. (This means a new measurement value is hashed together with the previous one in the PCR, and the result is saved in the same PCR.) This property in particular aims to prevent a compromised application from tampering with a previously saved measurement.

In ReDAS, the integrity evidence is extended into TPM PCR 8, which is one of the static PCRs. During the system operations, if none of the dynamic system properties are violated, both the integrity evidence and PCR 8 have their default values. On the other hand, if an integrity violation occurs, the integrity evidence is stored and the PCR is extended with the hash value of the evidence.

At the time of dynamic attestation, the attester sends both the integrity evidence and the value of PCR 8 (protected by the TPM signature) to the challenger. Thus, any malicious modification of the integrity evidence will be visible to the challenger.

At this point we would like to emphasize the importance of being able to "seal" our evidence before an attacker has a chance to corrupt it. Our integrity measurement component can guarantee the integrity of the evidence provided that it takes its actions before a system call is serviced.

2.6 ReDAS Dynamic Attestation Protocol

The ReDAS dynamic attestation protocol is a request/reply protocol, as shown in Figure 2.

1. A : generate random nonce
   A → B : dynamic attestation request, nonce
2. B → A : Integrity evidence, $\text{Sign}_{\text{AIK\_B}}\{\text{nonce, PCR 8}\}$
3. A : verify the TPM signature and the nonce
   if the signature or nonce is invalid then exit
   else validate the correctness of the integrity evidence using PCR 8

A: Challenger **B**: Attester (runs ReDAS)

Figure 2. Dynamic attestation protocol

The ReDAS dynamic attestation protocol is straightforward, and similar to previous static attestation protocols. The only difference is that the attestation response includes the *integrity evidence*, which is the history of all integrity violation proofs stored in the attester. The challenger (A) starts the protocol by generating a random nonce and including it with the dynamic attestation request message. Upon receiving the dynamic attestation request, the attestation service (B) generates a dynamic attestation response message, which includes two parts. The first part is the current integrity evidence from the integrity measurement component, including the list of integrity violation proofs in the order of their occurrences. (Note that this list is empty when there is no integrity violation on the attester.) The second part is the TPM signature of the current PCR 8 value along with the nonce received from the challenger. Once the challenger receives the dynamic attestation response message, it verifies the TPM's signature and the integrity evidence. While the nonce prevents replay attacks, the TPM signature of the hash of the integrity evidence ensures that the integrity evidence cannot be forged.

2.7 Security Analysis

In the following, we explain how we build the chain of trust from hardware up to the integrity evidence. We then discuss possible adversary actions against ReDAS and how we address such attacks.

2.7.1 Establishing Chain of Trust

Hardware and Kernel Integrity Measurement. Our trust building process starts by measuring the hardware of the attesting host. We use the CRTM and trusted boot to measure the initial state of the hardware and the kernel of the attester. This enables a remote challenger to verify if the hardware of the attester is genuine and whether the kernel is configured as expected and booted securely. As part of the kernel, IMA (Integrity Measurement Architecture) [26] is loaded and measured before any other kernel modules and applications. We implement the integrity measurement component as a Loadable Kernel Module (LKM) and have it loaded before any user applications. Thus, IMA will measure the integrity measurement component and attest to its load time integrity.

Application Load-Time Integrity Measurement. Protection against illegal modification of LKM and application binaries is done by using IMA [26]. IMA measures a pro-

gram's binary before it is loaded into memory for execution and protects the integrity of the measurement result by extending its hash value into a PCR. As a result, any illegal modification of a program's binary will be visible due to IMA.

Run-Time Protection of Integrity Measurement Component. The integrity measurement component can be trusted as long as the kernel is not compromised. Since the integrity measurement component intercepts system calls and the kernel's signals, it can capture malicious attacks that try to interfere with the integrity measurement component (e.g., try to remove the integrity measurement component). When the integrity measurement component captures such attacks, it can record the attack information in the integrity evidence and extend its measurement to a PCR. Thus, such attacks will be visible during dynamic attestation.

Integrity Evidence. Due to the protection of integrity evidence using TPM, the adversary cannot forge the PCR value without being detected. During dynamic attestation, the integrity evidence is protected by the TPM signature and the nonce. This prevents the adversary from altering the integrity evidence during transmission or reusing previous integrity evidence.

2.7.2 Possible Attacks and Defense

Given our threat model and a careful study of previous security issues in remote attestation, we categorize the possible threats and show how ReDAS is resilient to these attacks.

Modification. In a modification attack, the adversary tries to disrupt the attestation process by modifying the objects on which the attestation depends. In ReDAS, these include the application binaries, the specification of dynamic system properties to be verified at integrity measurement, and the integrity evidence. ReDAS uses IMA to provide load-time measurement of application binaries. The specification of dynamic system properties is also statically measured and verified before it is loaded into the integrity measurement component. The integrity evidence is protected since its hash value is extended into a PCR. As a result, any illegal modification of these objects will be visible to the remote challenger during dynamic attestation.

Emulation. In this attack, the adversary tries to emulate the attestation process or even emulate the entire host using a Virtual Machine Monitor (VMM). However, these techniques require changing both the hardware and software configurations. By using trusted boot and IMA, the difference in configurations is made visible to the challenger, and as a result, the attestation will fail.

Misdirection. During remote attestation, the remote challenger establishes a secure connection with the attester. In a misdirection attack, the adversary tries to redirect the attestation request to another platform running ReDAS to pass the attestation, and then provides the actual services from the original machine without ReDAS. Fortunately, countermeasures for this attack have already been developed. Goldman et al. [13] proposed to extend the PCR with a measurement that represents the static property of the secure connection endpoint (e.g., SSL public key). Using this new technique, the remote challenger can guarantee that the TPM signature was generated using the same machine at the endpoint of the tunnel.

Runtime Memory Corruption. The adversary may try to compromise an application by exploiting a vulnerability. Even if successful, the adversary is limited in its ability to affect the system. In order to avoid detection by ReDAS, the adversary either has to refrain from making system calls, or ensure that he/she does not modify dynamic objects examined by ReDAS. This limits the adversary's ability to make the exploit useful. The current implementation of ReDAS cannot capture all memory corruption attacks due to the limited dynamic system properties and measurement points. However, this is due to the limited scope of the current effort, rather than the dynamic attestation methodology.

Denial of Service (DoS) Attacks. The adversary may try to launch DoS attacks by repeatedly sending attestation requests. Such attacks are certainly not unique to ReDAS, and there are existing solutions such as rate limiting (e.g., [7]) and client puzzles (e.g., [15]) to prevent or mitigate such attacks.

3 Implementation

To assess the feasibility and effectiveness of our approach, we have implemented ReDAS on Linux kernel version 2.6.22. This section describes the implementation details of the main components in ReDAS.

3.1 Dynamic Property Collector

Each application monitored and attested for its integrity first runs through the dynamic property collection phase, which is done offline to gather the application's dynamic properties (currently structural constraints and data invariants). The dynamic property collector consists of modified Daikon [10], `Gcov` [2] to ensure the quality of the training phase and a collection of Ruby [4] scripts to gather information from the application binaries - mainly the structural constraints.

As we pointed out earlier, Daikon produces data invariants at function entries and exits. It does this by first running the application under a dynamic binary rewriter (Valgrind [22]), and generating a data trace at indicated program points. It then analyses the data traces to obtain the invariants. We modified the initial part so that Daikon produced data traces at system call times instead of function entries and exits. We use `Gcov` to ensure code coverage with our training sets which improves the quality of our training and hence, the resulting data invariants.

The structural information is gathered from the target applications (e.g., code segment range and stack frame sizes of user-defined functions) by parsing the debugging information. This process is fully automated and requires no human interaction. Finally all the information is compiled into a file to be used as input for the integrity measurement component.

3.2 Integrity Measurement Component

Measuring dynamic properties of an application starts when the integrity measurement component intercepts an `execve` system call and identifies the application being loaded.

When the application invokes a system call, the integrity measurement component blocks execution, locates the required dynamic objects (e.g., return addresses in the stack) within the application's memory space and examines their integrity. If the application passes all integrity checks, the integrity measurement component continues with serving the system call. If any checks fail, the related information, including the application's name, the violated constraint (or data invariant), and the current instruction are recorded into the integrity evidence.

When verifying heap related dynamic properties, we take advantage of the existing effort in the GNU C library. The library provides sanity checks for the heap management data, which is the same dynamic object that ReDAS needs to evaluate. If any sanity checks fail, the library invokes the `SIGABRT` signal to the OS kernel. Our integrity measurement component captures these signals, identifies the involved application, and records the integrity violation proof in the integrity evidence. By harnessing the library support, we reduce the burden of keeping track of dynamic memory allocations (deallocations) along with their integrity check.

The integrity measurement component is implemented as a loadable kernel module using SystemTap [5]. SystemTap is a free software simplifying the instrumentation of Linux kernels. The integrity measurement component takes as input the dynamic system properties generated by the dynamic property collector. Since direct file handling is not desirable within the kernel due to security reasons, the integrity measurement component enlists the help of a user level program called *verification initializer*. Communication between the integrity measurement component and the verification initializer is done using a `proc` file generated by the integrity measurement component. The verification initializer is a user level program that reads the dynamic system properties and sends them to the integrity measurement component through the `proc` file. Naturally, the input file needs to be authenticated (e.g., with Merkel hash tree or signature) before being used. Note that access to the proc file is governed by the integrity measurement component which supersedes the regular file access management.

As discussed in Section 2.5, ReDAS extends the integrity evidence into TPM's PCR 8. This is done by calculating the hash of the integrity evidence and then extending the PCR with this new hash value. The initial hash of the integrity evidence is computed within the integrity measurement component. Afterwards, it extends PCR 8 by directly accessing the TPM device driver, which is another loadable kernel module.

3.3 Attestation Service

The attestation service is implemented as a user level network service daemon. As mentioned in Section 2.6, the response from the attestation service includes the TPM's signature on the value of PCR 8 with the nonce received from the challenger and the current integrity evidence. In order to communicate with the TPM, the attestation service uses TrouSerS [6], an open source TPM library. TrouSerS allows the attestation service to read the current value of PCR 8 by using the `Tspi_TPM_PcrRead` function. The attestation service also uses the `Tspi_TPM_Quote` function to get the TPM to sign the value of PCR 8 and the nonce value. The attestation service gets the current integrity evidence from the integrity measurement component through the `proc` file created by the integrity measurement component.

4 Experimental Evaluation

We perform a series of experiments to evaluate both the effectiveness and efficiency of ReDAS. Our objective is to evaluate whether our choice of integrity measurement is a sound tradeoff between improved trust and performance.

4.1 Experimental Setup

All experiments are performed on a host running Ubuntu 7.1 with dual 1.86 GHz Intel Core 2 CPU, 3 GB of RAM, and a 10,000 RPM SATA hard drive. The host is equipped with a TPM that complies with TCG TPM specification v1.2 [3]. The kernel is configured to work with IMA [26]. All test applications are compiled using GCC version 4.1.3.

We choose nine real-world applications for our evaluation experiments. They are publicly available on the Internet, and are known to have different types of vulnerabilities. Table 1 lists the test applications and their known vulnerabilities.

Table 1. Test applications

Name	Description	Known Vulnerability
ghttpd 1.4	a web server	S(CVE-2002-1904)
prozilla 1.3.6	a web download accelerator	S(CVE-2004-1120)
sumus 0.2.2	a 'mus' game server	S(CVE-2005-1110)
newspost 2.1.1	a Usenet news poster	S(CVE-2005-0101)
wsmp3d 0.0.8	a web server with audio broadcasting	H(CVE-2003-0339)
xtelnetd 0.17	a telnet daemon	H(CA-2001-21)
nullhttpd 0.5.0	a web server	H(CVE-2002-1496)
powerd 2.0.2	a UPS monitoring daemon	F(CVE-2006-0681)
openvmps 1.3	a VLAN management server	F(CVE-2005-4714)

S: Stack Overflow, H: Heap Overflow, F: Format String

Table 2 shows some statistics about the data invariants in the test applications. The data invariants column shows the total number of data invariants collected from each application. It also shows minimum, maximum, and average numbers of data invariants to examine at each measurement point (system call). We do not provide any numbers for structural

constraints, as the number of checks performed during runtime cannot be determined statically. Moreover, some of these structural constraints do not require input for verification. For example, the saved frame pointers in an application's stack must be linked together to satisfy the saved frame pointer constraint. These values are determined during runtime by walking the user stack.

Table 2. Data invariants of test applications

Application Name	Data Invariants			
	Total	Min	Max	Average
ghttpd 1.4	34	2	3	2
prozilla 1.3.6	19	1	3	2
sumus 0.2.2	249	24	40	31
newspost 2.1.1	294	22	23	22
wsmp3d 0.0.8	331	10	45	30
xtelnetd 0.17	237	50	135	79
nullhttpd 0.5.0	481	23	33	26
powerd 2.0.2	110	4	8	5
openvmps 1.3	34	2	10	2
Average	199	15	33	22

* Min/Max/Average # of data invariants at measurement points

4.2 Effectiveness

We evaluate the effectiveness of ReDAS in two ways. First, we test ReDAS to see if it triggers any false alarms when the attester is running normally (i.e., under no attacks). Second, we evaluate ReDAS to understand how well it captures integrity violations when the attester is under attack.

To test if ReDAS triggers any false alarms, we first generate test cases for each application. In order to develop the test cases, we take a similar approach to what we used in Daikon's training phase. We try to ensure that the test cases do not overlap with the training data. This was not possible with two applications (`powerd` and `openvmps`)) which have very limited usages. For example, `powerd` only accepts 4 different messages (`OK`, `CANCEL`, `FAIL`, and `SHUTDOWN`). After obtaining the test cases, we run the tests and check if ReDAS generates any false integrity violation proofs. Our results show that none were generated for any of the applications tested.

To evaluate how well ReDAS captures integrity violations caused by runtime attacks, we use the attack programs that exploit the vulnerabilities of the test applications. We obtain most of the attack programs from publicly known web sites (e.g., `www.milw0rm.com`). One shortcoming of this list is that none of the exploits target global data. Although such vulnerable applications exist (e.g., [27, 28]), we do not have access to their source code and can't include them in our test. To compensate for this, we modify an attack program that exploits a format string vulnerability in `powerd` so that the new exploit overwrites the global variable `outtime` in `powerd`. We use powerd 2.0.2* to indicate this new test case. In each experiment, we first launch the application, then use the attack program to generate the malicious input, and send it to the application. Finally, we check the integrity evidence to see if ReDAS captures the integrity violations caused by the attacks.

Table 3 summarizes our evaluation result. It shows the application name, violated dynamic property, and measurement point where ReDAS captures the integrity violation. In all cases, ReDAS captures the integrity violations caused by the attacks. For example, ReDAS reports that `ghttpd` violates the return address constraint at `sys_read` system call.

Table 3. Effectiveness evaluation result

Name	Dynamic Property Violation	Measurement Point
ghttpd 1.4	Return address constraint	3 (`sys_read`)
prozilla 1.3.6	Return address constraint	3 (`sys_read`)
sumus 0.2.2	Return address constraint	4 (`sys_write`)
newspost 2.1.1	Caller-callee constraint	3 (`sys_read`)
wsmp3d 0.0.8	Boundary tag constraint	`malloc`
xtelnetd 0.17	Boundary tag constraint	`malloc`
nullhttpd 0.5.0	Boundary tag constraint	`free`
powerd 2.0.2	Return address constraint	4 (`sys_write`)
powerd 2.0.2*	Equality invariant	4 (`sys_write`)
openvmps 1.3	Return address constraint	102 (`sys_socketcall`)

Overall, our effectiveness evaluation shows a promising result in that our methods of determining dynamic properties are sound, produce no false positives, and are adequate in capturing runtime integrity violations effectively.

4.3 Efficiency

To measure the performance overhead, we run the test applications with and without ReDAS, and compare their throughput. The throughput is measured according to the type of application. For server applications (e.g., `ghttpd`), we use the Apache benchmark [1] to emulate multiple clients (1–20) sending different types of requests to the application. The apache benchmark then outputs the average throughput (# of requests/sec) of the application. For client applications (e.g., `prozilla`), we compute the average speed of executing the application. For example, in `prozilla`, we check the average speed of downloading various types of files from multiple local web sites. In each experiment, we iterate the test 20 times and compute the average throughput of the application. Note that some test applications are not suitable for measuring their performance by using their throughput. For `xtelnetd`, `powerd`, and `openvmps`, we could not obtain any performance benchmark program to measure their performance. It is also not appropriate to measure their throughput using average speed of execution since they are daemon programs that run continuously. For `newspost`, throughput depends on the responding party (e.g., Usenet server) rather than its own execution. Thus, we exclude these applications in the experiments.

Figure 3 illustrates the performance result. The X-axis lists the names of the applications and the Y-axis shows the additional overhead (%) of running the application with ReDAS compared to running it without ReDAS. In each application, three bars show the overhead of verifying structural constraints only, data invariants only, and both. For example, `prozilla` shows a 2.05% overhead when verifying both its structural constraints and data invariants.

The performance is affected by the frequency of system calls, average number of data invariants per system call, and

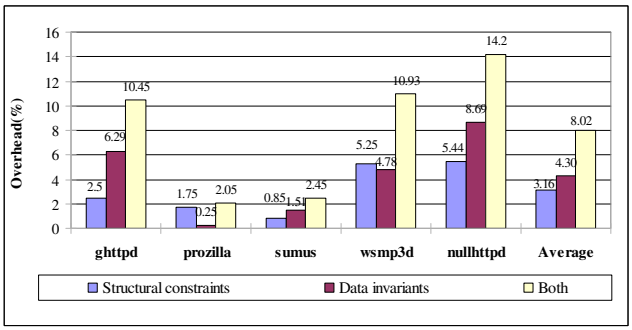

Figure 3. ReDAS Performance Overhead

size of the data invariants to be checked. Despite all these factors, our results show that our ReDAS implementation incurs on average 8% overhead for verifying both types of dynamic properties.

4.4 Limitations

ReDAS inherits the limitations of its integrity measurement component. In our particular case, these theoretical limitations include false negatives (integrity violations that go unnoticed) of three types. The first type is the result of transient attacks that erase or fix their trace before the monitoring component has a chance to observe the integrity violation. The second type of false negatives is due to an incomplete set of dynamic objects. In this scenario, an attack may not violate any of the dynamic properties that ReDAS is monitoring. Finally, our use of Daikon's data invariants limits us to dynamic properties of simple programming constructs. This means that we cannot infer any useful dynamic properties for complex structures (e.g., linked lists).

Despite these limitations, ReDAS has advanced the state of the art of remote attestation. Static attestation cannot handle runtime attacks at all. As the first step of dynamic attestation, ReDAS can now partially address integrity violations caused by runtime attacks. It is necessary to understand the coverage of various dynamic system properties and explore different measurement points to have "perfect" dynamic attestation. We will investigate these issues in the future.

5 Related Work

Our work is closely related to property based attestation [8], which requires a trusted third party certifying a mapping between the properties that the attester claims and the actual configurations of the system. Our work differs from it in the definition of properties. They define the properties as the attester's platform configurations, while the properties in our work are those of dynamic objects that must be satisfied during their lifetime. More importantly, the method in [8] is limited to verifying static properties of the system, while ours is aimed at dynamic attestation.

Multiple static attestation techniques have been proposed to provide attestation in various situations (e.g., IMA [26], SWATT [29], Pioneer [30], [32], [21]). However, all these approaches focus on attestation or measurement of the static parts of the attested systems and applications, but cannot address the dynamic parts, which are equally important for system integrity. BIND [33] and Flicker [20] both use a method for fine-grained runtime code attestation by binding the integrity of the executed code with its input/output, which can be considered one of the attested dynamic system properties. However, our work is more general in addressing the need of dynamic attestation.

There were also efforts [18, 24, 25] to measure the Linux kernel runtime integrity. Copilot [24] uses a PCI card to directly measure the runtime kernel integrity without the operating system's support. Loscocco et al. [18] developed a kernel integrity monitoring program to verify kernel data structures. Petroni and Hicks [25] proposed a technique to monitor the runtime kernel's control-flow integrity. Such efforts are complimentary to our work.

6 Conclusion

In this paper, we introduced the notion of dynamic attestation to overcome limitations of previous static attestation techniques. We present a novel dynamic attestation system called ReDAS. ReDAS provides integrity evidence of running applications by monitoring them during runtime. In this paper, we describe the security challenges related to building such a system and provide effective solutions. The experimental results obtained with real-world applications indicate that ReDAS is effective and practical in performing application-level dynamic attestation.

Although ReDAS extends current remote attestation capability by enabling attestation to dynamic system properties, it is not a complete solution yet. In our future work, we plan to investigate other types of dynamic system properties and different measurement points to improve the integrity measurement capability further. We will also investigate how we can perform dynamic attestation of OS kernels.

References

[1] Apache benchmarking tool. The Apache Software Foundation, 2008. `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[2] Gcov, a Test Coverage Program. available at `http://gcc.gnu.org/onlinedocs/gcc/Gcov.html`.

[3] TPM Specifications Version 1.2 available at `https://www.trustedcomputinggroup.org/downloads/specifications/tpm/tpm`.

[4] Ruby programming language. `http://www.ruby-lang.org/en/`.

[5] SystemTap. `http://sourceware.org/systemtap/`.

[6] TrouSerS. The open-source TCG Software Stack available at `http://trousers.sourceforge.net/`.

[7] D. G. Andersen. Mayday: Distributed Filtering for Internet Services. In *Proceedings of the 4th USENIX Symposium on In-*

ternet Technologies and Systems (USITS '03), Seattle, Washington, March 2003.

[8] L. Chen, R. Landfermann, H. Loehr, M. Rohe, A. Sadeghi and C. Stuble. A Protocol for Property-Based Attestation. In *First ACM Workshop on Scalable Trusted Computing (STC'06)*, Fairfax, Virginia, November 2006.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, 2007.

[11] J. Garay and L. Huelsbergen. Software Integrity Protection Using Timed Executable Agents. In *Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS'06)*, pp. 189-200, Taiwan, March 2006.

[12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, October 2003.

[13] K. Goldman, R. Perez, and R. Sailer. Linking remote attestation to secure tunnel endpoints. In *Proceedings of the first ACM workshop on Scalable Trusted Computing (STC'06)*, pp. 21-24, Fairfax, Virginia, November 2006.

[14] N. Gupta and Z. V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *Proceedings of the 18th Annual International Conference on Automated Software Engineering (ASE 2003)*, 2003.

[15] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion. In *Proceedings of Networks and Distributed Systems Security Symposium (NDSS '99)*, 1999.

[16] C. Kil, E. C. Sezer, P. Ning, and X. Zhang. Automated Security Debugging Using Program Structural Constraints. In *Annual Computer Security Applications Conference (ACSAC 07)*, Miami Beach, Florida, December 2007.

[17] L. C. Lam, W. Li, T. Chiueh. Accurate and automated system call policy-based intrusion prevention. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2006.

[18] P. Loscocco, P. W. Wilson, J. A. Pendergrass and C. D. McDonell. Linux kernel integrity measurement using contextual inspection. In *Proceedings of the 2007 ACM workshop on Scalable Trusted Computing (STC'07)*, Alexandria, Virginia, October/November 2007.

[19] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetho, and S. Yoshihama. Trusted Platform on demand (TPod). Research Report RT0564, IBM, February 2004.

[20] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys'08)*, Glasgow, Scotland, March/April 2008.

[21] M. Milenkovi, A. Milenkovi, and E. Jovanov. Hardware support for code integrity in embedded processors. In *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems (CASE '05)*, San Francisco, California, September 2005.

[22] N. Nethercote and J Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, California, USA, June 2007.

[23] T. Park and K. G. Shin. Soft Tamper-Proofing via Program Integrity Verification in Wireless Sensor Networks. In *IEEE Transactions on Mobile Computing*, vol. 4, no. 3, pp. 297-309, May/June 2005.

[24] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot- A Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of the 13th USENIX Security Symposium*, pp. 179-194, San Diego, California, August 2004.

[25] N. L. Petroni, Jr. and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pp. 103-115, Alexandria, Virginia, October 2007.

[26] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *13th Usenix Security Symposium*, San Diego, California, August 2004.

[27] SecuriTeam. Symantec VERITAS Multiple Buffer Overflows. `http://www.securiteam.com/securitynews/5JP0L2KI0A.html`.

[28] SecuriTeam. Windows 2000 and NT4 IIS .HTR Remote Buffer Overflow. `http://www.securiteam.com/windowsntfocus/5MP0C0U7FU.html`.

[29] A. Seshadri, A. Perrig, L. van Doorn and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *IEEE Symposium on Security and Privacy*, May 2004.

[30] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn and P. Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *20th ACM Symposium on Operating Systems Principles (SOSP 2005)*, Brighton, United Kingdom, October 2005.

[31] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS '07)*, October 2007.

[32] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote Software-based Attestation for Wireless Sensors. In *Proceedings of the 2nd European Workshop on Security and Privacy in Ad Hoc and Sensor Networks*, Visegrad, Hungary, July 2005.

[33] E. Shi, A. Perrig, and L. van Doorn. BIND: A Time-of-use Attestation Service for Secure Distributed Systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 2005.

[34] E. R. Sparks. A Security Assessment of Trusted Platform Modules. Dartmouth University, Technical Report TR2007-597, June, 2007.

[35] C. Xiao. Performance Enhancements for a Dynamic Invariant Detector. Masters thesis, MIT Department of Electrical Engineering and Computer Science, February 2007.