DR@FT: Efficient Remote Attestation Framework for Dynamic Systems*

Wenjuan Xu[1], Gail-Joon Ahn[2], Hongxin Hu[2],
Xinwen Zhang[3], and Jean-Pierre Seifert[4]

[1] Frostburg State University
wxu@frostburg.edu
[2] Arizona State University
gahn@asu.edu ,hxhu@asu.edu
[3] Samsung Information Systems America
xinwen.z@samsung.com
[4] Deutsche Telekom Lab and Technical University of Berlin
jean−pierre.seifert@telekom.de

Abstract. Remote attestation is an important mechanism to provide the trustworthiness proof of a computing system by verifying its integrity. In this paper, we propose an innovative remote attestation framework called DR@FT for efficiently measuring a target system based on an information flow-based integrity model. With this model, the high integrity processes of a system are first verified through measurements and these processes are then protected from accesses initiated by low integrity processes. Also, our framework verifies the latest state changes in a dynamic system instead of considering the entire system information. In addition, we adopt a graph-based method to represent integrity violations with a ranked violation graph, which supports intuitive reasoning of attestation results. Our experiments and performance evaluation demonstrate the feasibility and practicality of DR@FT.

1 Introduction

In distributed computing environments, it is crucial to measure whether remote parties run buggy, malicious application codes or are improperly configured by rogue software. Remote attestation techniques have been proposed for this purpose through analyzing the integrity of remote systems to determine their trustworthiness. Typical attestation mechanisms are designed based on the following steps. First, an attestation requester (*attester*) sends a challenge to a target system (*attestee*), which responds with the evidence of integrity of its hardware and software components. Second, the attester derives runtime properties of the attestee and determines the trustworthiness of the attestee. Finally and optionally, the attester returns the attestation result, such as integrity measurement status, to the attestee. Remote attestation can help reduce potential risks that are caused by a tampered system.

* The work of Gail-J.Ahn and Hongxin Hu was partially supported by National Science Foundation (NSF-IIS-0900970 and NSF-CNS-0831360).

D. Gritzalis, B. Preneel, and M. Theoharidou (Eds.): ESORICS 2010, LNCS 6345, pp. 182–198, 2010.

Various attestation approaches and techniques have been proposed. Trusted Computing Group (TCG) [2] specifies trusted platform module (TPM) which can securely store and provide integrity measurements of systems to a remote party. Integrity measurement mechanisms have been proposed to facilitate the capabilities of TPM at application level. For instance, Integrity Measurement Architecture (IMA) [12] is an implementation of TCG approach to provide verifiable evidence with respect to the current run-time state of a measured system. Several attestation methods have been proposed to accommodate privacy properties [7], system behaviors [8], and information flow model [9]. However, these existing approaches still need to cope with the efficiency when attesting a platform where its *system state* frequently changes due to system-centric events such as security policy updates and software package installations. Last but not least, existing attestation mechanisms do not have an effective and intuitive way for presenting attestation results and reflecting such results while resolving identified security violations.

Towards a systematic attestation solution, we propose an efficient remote attestation framework, called Dynamic Remote Attestation Framework and Tactics (DR@FT) to address aforementioned issues. Our framework is based on system integrity property with a *domain-based isolation* model. With this property, the high integrity processes of a system are first verified through measurements and these processes are then protected from accesses initiated by low integrity processes. In other words, the high integrity process protection is verified by analyzing the system's security policy, which specifies system configurations with system behaviors including application behaviors. Having this principle in place, DR@FT enables us verify whether certain applications in the attestee satisfy integrity requirements as part of attestation. Towards attesting a dynamic nature of the systems, DR@FT verifies the latest changes in a system state, instead of considering the entire system information for each attestation inquiry. Through these two tactics, our framework attempts to efficiently attest the target system. Also, DR@FT adopts a graph-based analysis methodology for analyzing security policy violations, which helps cognitively identify suspicious information flows in the attestee. To further improve the efficiency of security violation resolution, we propose a ranking scheme for prioritizing the policy violations, which provides a method for describing the *trustworthiness* of different system states with *risk levels*.

This paper is organized as follows. Section 2 overviews existing attestation work and system integrity models. Section 3 describes a domain-based isolation model which gives the theoretical foundation of DR@FT. Section 4 presents the design requirements and attestation procedures of DR@FT, followed by policy analysis methods and their usages in Section 5. We elaborate the implementation details and evaluation results in Section 6. Section 7 concludes this paper and examines some limitations of our work.

2 Background

2.1 Attestation

The TCG specification [2] defines mechanisms for a TPM-enabled platform to report its current hardware and software configuration status to a remote challenger. A TCG attestation process is composed of two steps: (i) an attestee platform measures hardware and software components starting from BIOS block and generates a hash value.

The hash value is then stored into a TPM Platform Configuration Register (PCR). Recursively, it measures BIOS loader and operating system (OS) in the same way and stores them into TPM PCRs; (ii) an attester obtains the attestee's digital certificate with an attestation identity key (AIK), AIK-signed PCR values, and a measurement log file from the attestee which is used to reconstruct the attestee platform configuration, and verifies whether this configuration is acceptable. From these steps we notice that TCG measurement process is composed of a set of sequential steps up to the bootstrap loader. Thus, TCG does not provide effective mechanisms for measuring a system's integrity beyond the system boot, especially considering the randomness of executable contents loaded by an running OS.

IBM IMA [12] extends TCG's measurement scope to application level. A measurement list *M* is stored in OS kernel and composed of m_0 ... m_i corresponding to loaded executable application codes. For each loaded m_i, an aggregated hash H_i is generated and loaded into TPM PCR, where H_0=H(m_0), and H_i=H(H_{i-1} || H(m_i)). Upon receiving the measurements and TPM-signed hash value, the attester proves the authentication of measurements by verifying the hash value, which helps determine the integrity level of the platform. However, IMA requires to verify the entire components of the attestee platform while the attestee may only demand the verification of certain applications. Also, the integrity status of a system is validated by testing each measurement entry independently, focusing on the high integrity processes. However, it is impractical to disregard newly installed untrusted applications or data from the untrusted network.

PRIMA [9] is an attestation work based on IMA and CW-Lite integrity model [14]. PRIMA attempts to improve the efficiency of attestation by only verifying codes, data, and information flows related to trusted subjects. On one hand, PRIMA needs to be extended to capture the dynamic nature of system states such as software and policy updates, which could be an obstacle for maintaining its efficiency. On the other hand, PRIMA represents an attestation result with binary decision (trust or distrust) and does not give semantic information about how much the attestee platform can be trusted.

Property-based attestation [7] is an effort to protect the privacy of a platform by collectively mapping related system configurations to attestation properties. For example, "SELinux-enabled" is a property which can be mapped to a system configuration indicating that the system is protected with an SELinux policy. That is, this approach prevents the configurations of a platform from being disclosed to a challenger. However, due to the immense configurations of the hardware and software of the platform, mapping all system configurations to properties is infeasible and impractical.

2.2 Integrity Models

To describe the integrity status for a system, there exist various information flow-based integrity models such as Biba [5], LOMAC [16], Clark-Wilson [13], and CW-Lite [14]. Biba integrity property is fulfilled if a high integrity process cannot read/execute a lower integrity object, nor obtain lower integrity data in any other manner. LOMAC allows high integrity processes to read lower integrity data, while downgrading the process's integrity level to the lowest integrity level that has ever been activated. Clark-Wilson provides a different view of dependencies, which states information can flow from low integrity objects to high integrity objects through a specific program called transaction

procedures (TP). Later, the concept of TP is evolved as a filter in the CW-Lite model. The filter can be a firewall, an authentication process, or a program interface for downgrading or upgrading the privileges of a process.

With existing integrity models, there is a gap between concrete the measurements of a system's components and the verification of its integrity status. We believe an application-oriented and domain-centric approach accommodates the requirements of attestation evaluation better than advocating an abstract model. For example, in a Linux system, a subject in one of traditional integrity models can correspond to a set of processes, belonging to a single application domain. For instance, an Apache domain may include various process types such as `httpd_t`, `http_sysadm_devpts_t`, and `httpd_ prewikka_script_t`. All of these types can have information flows among them, which should be regarded as a single integrity level. Also, sensitive objects in a domain should share the same integrity protection of its subjects. To comprehensively describe the system integrity requirements, in this paper, we propose a domain-based isolation approach as discussed in the subsequent section.

3 Domain-Based Isolation

According to TCG and IMA, the trustworthiness of a system is described with the measured integrity values (hash values) of loaded software components. However, the measurement efficiency and attestation effectiveness are major problems of these approaches since (i) too many components have to be measured and tracked, and (ii) too many known-good hash values are required from different software vendors or authorities. Fundamentally, this requires that in order to trust a single application of a system, every software component in the system has to be trusted; otherwise the attestation result should be negative. In our work, we believe that the trustworthiness of a system is tightly related to the integrity status, which is, in turn, described by a set of integrity rules that are enforced by the system. If any of the rules is violated, it should be detected. Hence, in order to trust a single application domain, we just need to ensure the *system TCB*–including reference monitor and integrity rules protecting the target application domain–are not altered.

Based on this anatomy, we introduce domain-based isolation principles for integrity protection, which are the criteria to describe the integrity status of a system and thus its trustworthiness. We first propose general methodologies to identify high integrity processes, which include *system TCB* and *domain TCB*, and then we specify security rules for protecting these high integrity processes. System TCB (TCB_s) is similar to the concept of traditional TCB [3], which can be identified along with the subjects functioning as the reference monitor of a system [4]. Applying this concept to SELinux [15], for example, subjects functioning as reference monitor such as `checkpolicy` and `loading policy` belong to system TCB. Also, subjects used to support reference monitor such as `kernel` and `initial` should also be included into system TCB. With this approach, an initial TCB_s can be identified, and other subjects such as `lvm` and `restorecon` can be added into TCB_s based on their relationships with the initial TCB_s. Other optional methods for identifying TCB_s are proposed in [10]. Considering the similarity of operating systems and configurations, we expect that the results

would be similar. Furthermore, for attestation purpose, TCB_s also includes integrity measurement and reporting components, such as kernel level integrity measurement agent [1] and attestation request handling agent.

In practice, other than TCB_s, an application or user-space service also can affect the integrity thus the behavior of a system. An existing argument [3] clearly states the situation: "A network server process under a UNIX-like operating system might fall victim to a security breach and compromise an important part of the system's security, yet is not part of the operating system's TCB." Accordingly, a comprehensive mechanism of policy analysis for TCB identification and integrity violation detection is desired. Hence, we introduce a concept called information domain TCB (or simply *domain TCB,* TCB_d). Let *d* be an information domain functioning as a certain application or service through a set of related subjects and objects, domain *d*'s TCB or TCB_d is composed of a set of subjects and objects in information domain *d* which have the same level of security sensitivity. By the same level of security sensitivity, we mean that, if information can flow to some subjects or objects of the domain, it can flow to all others in the domain. That is, they need the same level of integrity protection. Prior to the identification of TCB_d, we first identify the information domain *d* based on its main functions and relevant information flow associated with these functions. For example, a running web server domain consists of many subjects–such as `httpd` process, plugins, tools, and other objects–such as data files, configuration files, and logs.

The integrity of an object is determined by the integrity of subjects that have operations on this object. All objects dependent on TCB_d subjects are classified as TCB protected objects or resources. Thus we need to identify all TCB_d subjects from an information domain and verify the assurance of their integrity. To ease this task, a minimum TCB_d is first discovered. In the situation that the minimum TCB_d subjects have dependency relationships with other subjects, these subjects should be added to domain TCB, or the dependencies should be removed. Based on these principles, we first identify initial TCB_d subjects which are predominant subjects for the information domain *d*. We further discover other TCB_d subjects considering subject dependency relationships with the initial TCB_d through *information flow transitions*, which means that the subjects that can only flow to and from the initial TCB_d subjects should be included into TCB_d. For instance, for a web server domain, `httpd` is the subject that initiates other web server related processes. Hence, `httpd` is the predominant subject and belongs to TCB_d. Then, based on all possible information flows to `httpd`, we identify other subjects such as `httpd-suexec` in TCB_d.

To protect the identified TCB_s and TCB_d, we develop principles similar to those in Clark-Wilson [13]. Clark-Wilson leverages transaction procedures (TP) to allow information flow from low integrity to high integrity processes. Hence, we also develop the concept of filters. Filters can be processes or interfaces [11] that normally are distinct input information channels and are created by a particular operation such as `open()`, `accept()`, or other calls that enable data input. For example, `su` process allows a low integrity process (e.g., staff) being changed to be a high integrity process (e.g., root) by executing `passwd` process, thus `passwd` can be regarded as a filter for processes run by root privilege. Also, high integrity process (e.g., `httpd` administration) can accept low integrity information (e.g, network data) through the secure channel such as `sshd`.

Consequently, `sshd` can be treated as another example of filter for higher privilege processes. With the identifications of TCB_s, TCB_d and filters, for information domain `d`, all the other subjects in a system are categorized as NON-TCB.

Definition 1. Domain-based isolation *is satisfied for an information domain* d *if information flows are from* TCB_d*; or information flows are from* TCB_s *to either* TCB_d *or* TCB_d *protected resources; or information flows are from NON-TCB to either* TCB_d *or* TCB_d *protected resources via filter(s).*

4 Design of DR@FT

DR@FT consists of three main parties: an attestee (the target platform), an attester (attestation challenger), and a trusted authority, as shown in Figure 1. The attestee is required to provide its system state information to the attester to be verified. Here, we assume that an attestee initially is in a *trusted system state*. After certain system behaviors, the system state is changed to a new state.

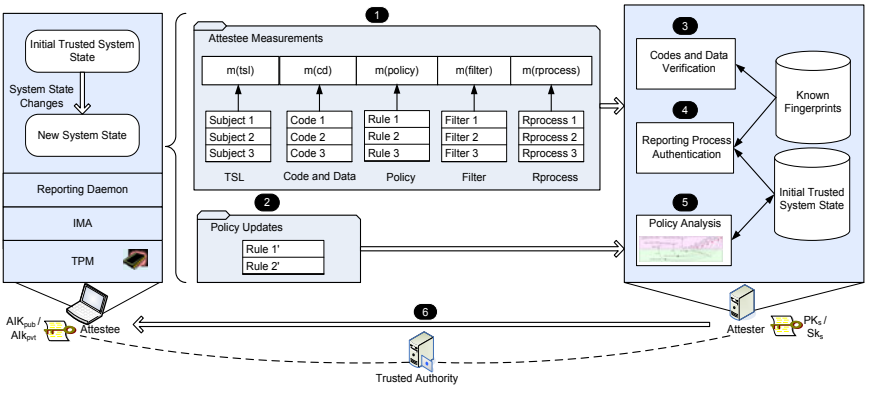


Fig. 1. Overview of DR@FT

An attestation reporting daemon on the attestee gets the measured new state information (step 1) with IMA, and generates the policy updates (step 2). This daemon then gets AIK-signed PCR value(s) and sends to the attester. After the attester receives and authenticates the information, with the attestee's AIK public key certificate from the trusted authority, it verifies the attestee integrity through codes and data verification (step 3), reporting process authentication (step 4) and policy analysis (step 5).

4.1 System State and Trust Requirement

For the attestation purpose, the system state is a snapshot of an attestee system at a particular moment, where the factors characterizing the state can influence the system integrity in any future moment of the system. Based on the domain-based isolation, the attestee system integrity can be represented via information flows, which are characterized by the trusted subject list, filters, policies, and the trustworthiness of TCB_s. Based on these, we define the system state of the attestee as follows.

Definition 2. *A* system state *at the time period i is a tuple T_i=\{ TSL_i, CD_i, $Policy_i$, $Filter_i$, $RProcess_i$ \}, where*

- *TSL_i=\{s_0, s_1....s_n\} represents a set of high integrity processes which corresponds to the set of subjects s_0, s_1....s_n in TCB_s and TCB_d.*
- *CD_i=\{cd (s_0), cd (s_1)....cd (s_n)\} is a set of codes and data for loading a subject $s_j \in TSL_i$.*
- *$Policy_i$ is the security policy currently configured on the attestee.*
- *$Filter_i$ is a set of processes defined to allow information flow from low integrity processes to high integrity processes.*
- *$RProcess_i$ represents a list of processes that measure, monitor, and report the current TSL_i, CD_i, $Filter_i$ and $Policy_i$ information. IMA agent and the attestation reporting daemon are the examples of the $RProcess_i$.*

According to this definition, a system state does not include a particular application's running state such as its memory page and CPU context (stacks and registers). It only represents the security configuration or policy of an attestee system. A system *state transition* indicates one or more changes in TSL_i, CD_i, $Policy_i$, $Filter_i$, or $RProcess_i$. A system state T_i *is trusted* if TSL_i belongs to TCB_s and TCB_d; CD_i does not contain untrusted codes and data; $Policy_i$ satisfies domain-based isolation; $Filter_i$ belongs to the defined filter in domain-based isolation; and $RProcess_i$ codes and data do not contain malicious codes and data and these $RProcess_i$ processes are integrity protected from the untrusted processes via $Policy_i$.

As mentioned, we assume there exists an initial trusted system state T_0 which satisfies. Through changing the variables in T_0, the system transits to states T_1, T_2 ... T_i. The attestation purpose is to verify if any of these states is trusted.

4.2 Attestation Procedures

Attestee Measurements. The measurement at the attestee side has two different forms, depending on *how much* the attestee system changes. In case any subject in TCB_s is updated, the attestee must be fully remeasured from the system reboot and the attester needs to attest it completely, as this subject may affect the integrity of subjects in $RProcess$ of the system such as the measurement agent and reporting daemon. After the reboot and all TCB_s subjects are remeasured, a trusted initial system state T_0 is built. To perform this re-measurement, the attestee measures a state T_i and generates the measurement list M_i which is added by Trusted subject list (TSL_i) measurement, Codes and data (CD_i) measurement,Policy ($Policy_i$) measurement, Filter ($Filter_i$) measurement and Attestation Process ($RProcess_i$) measurement. Also, $\mathcal{H}(M_i)$ is extended to a particular PCR of the TPM, where $\mathcal{H}$ is a hash function such as SHA1.

In another case, where there is no TCB_s subject updated and the TSL_i or $Filter_i$ subjects belonging to TCB_d are updated, the attestee only needs to measure the updated codes and data loading the changed TSL or filter subject, and generates a measurement list M_i. The generation of this measurement list is realized through the run-time measurement supported by the underlying measurement agent.

To support both types of measurements, we develop an attestation reporting daemon which monitors the run-time measurements of the attestee. In case the run-time

measurements for the TCB_s are changed, the attestee is required to be rebooted and fully measured with IMA. The measurements values are then sent to the attester by the daemon. On the other side, the changed measurement value is measured by IMA and captured with the reporting daemon only if the measurement for TCB_d is changed. Obviously, this daemon should be trusted and is included as part of TCB_s. That is, its codes and data are required to be protected with integrity policy and corresponding hash values are required to be stored at the attester side.

Policy Updates. To analyze if the current state of the attestee satisfies domain-based integrity property, the attester requires information about the current security policy loaded at the attestee side. Due to the large volume of policy rules in a security policy, sending all policy rules in each attestation and verifying all of them by the attester may cause the performance overhead. Hence, in DR@FT, the attestee only generates policy updates from the latest attested trusted state and sends them to the attester for the attestation of such updates.

To support this mechanism, we have the attestation reporting daemon monitor any policy update on attestee system and generate a list of updated policy rules. Note that the policy update comparison is performed between the current updated policy and the stored trusted security policy $Policy_0$ or previously attested and trusted $Policy_{i-1}$. The complexity of this policy update algorithm is *O(nr)*, where *nr* represents the number of the policy rules in the new policy file $Policy_i$.

Codes and Data Verification. With received measurement list M_i and AIK-signed PCRs, the attester first verifies the measurement integrity by re-constructing the hash values and compares with PCR values. After this is passed, the attester performs the analyses. Specifically, it obtains the hash values of CD_i and checks if they corresponds to known-good fingerprints. Also, the attester needs to assure that the TSL_i belongs to TCB_s and TCB_d. In addition, the attester also gets the hash value of $Filter_i$ and ensures that they belong to the filter list defined on the attester side. In case this step successes, the attester has the assurance that target processes on attestee side are proved without containing any untrusted code or data, and the attester can proceed to next steps. Otherwise, the attester sends a proper attestation result denoting this situation.

Authenticating Reporting Process. To prove that the received measurements and updated policy rules are from the attestee, the attester authenticates them by verifying that all the measurements, updates and integrity measurement agent processes in the attestee are integrity protected. That is, the $RProcess_i$ does not contain any untrusted codes or data and its measurements correspond to PCRs in the attester. Also, there is no integrity violated information flow to these processes from subjects of TSL_i, according to the domain isolation rules. Note that these components can also be updated, but after any update of these components, the system should be fully re-measured and attested from boot time as aforementioned, i.e., to re-build a trusted initial system state T_0.

Policy Analysis by Attester. DR@FT analyzes policy using a graph-based analysis method. In this method, a policy file is first visualized into a graph, then this policy graph is analyzed against pre-defined security model such as our domain-based isolation, and a policy violation graph is generated. The main goal of this approach is to give

semantic information of attestation result to the attestee, such that its system or security administrator can quickly and intuitively obtain any violated integrity configuration.

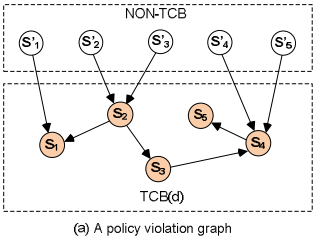
Note that verifying all the security policy rules in each attestation request decrease the efficiency, as loading policy graph and checking all the policy rules one by one cost a lot of time. Thus, we need to develop an efficient way for analyzing the attestee policy. In our method, the attester stores the policy of initial trusted system state T_0 or the latest trusted system state T_i, and its corresponding policy graph is loaded which does not have any policy violation. Upon receiving the updated information from the attestee, the attester just needs to analyze these updates to see if there is new information flow violating integrity.

Through this algorithm, rather than analyzing all the policy rules and all information flows for each attestation, we verify the new policy through only checking the updated policy rules and the newly identified information flow. The complexity of this policy analysis algorithm is *O(nn +nl +nt)*, where *nn* represents the number of changed subjects and objects, *nl* is the number of changed subjects and objects relationship in the policy update file; and *nt* represents the number of changed TCB in the TSL file.

Attestation Result Sending to Attester. In case the attestation is successful, a new trusted system state is developed and the corresponding information is stored at the attester side for subsequent attestations. On the other hand, if the attestation fails, there are several possible attestation results including CD_i *Integrity Fail*, CD_i *Integrity Success,* $RProcess_i$ *Unauthenticated, and* $Policy_i$ *Fail/Success*, and CD_i *Integrity Success,* $RProcess_i$ *Authenticated, and* $Policy_i$ *Fail /Success*. To assist the attestee reconfiguration, the attester also sends a representation of the policy violation graph to the attestee. Moreover, with this policy violation graph, the attester specifies the violation ranking and the trustworthiness of the attestee, which is explained in next section.

5 Integrity Violation Analysis

As we discussed in Section 1, existing attestation solutions such as TCG and IMA lack the expressiveness of the attestation result. In addition to their boolean-based response for attestation result, DR@FT adopts a graph-based policy analysis mechanism, where a policy violation graph can be constructed for identifying all policy violations on the



(a) A policy violation graph

Subject	SubjectRank
S_1	0.2799
S_2	0.3999
S_3	0.0799
S_4	0.4319
S_5	0.3456

(b) SubjectRank in TCB(d)

Path	PathRank
$\langle S'_1, S_1 \rangle$	0.2799
$\langle S'_2, S_2 \rangle$	0.8104
$\langle S'_3, S_2 \rangle$	0.8104
$\langle S'_4, S_4 \rangle$	0.6048
$\langle S'_5, S_4 \rangle$	0.6048

(c) PathRank between NON-TCB and TCB(d)

Fig. 2. Example policy violation graph and rank. The SubjectRank and PathRank indicate the severity of violating paths.

attestee side. We further introduce a risk model built on a ranking scheme, which gives the implication of how severe the discovered policy violations are, and how to efficiently resolve them.

5.1 Policy Violation Graph

Based on domain-based isolation model, we can find out two kinds of *violation paths*, *direct violation paths* and *indirect violation paths*. A direct violation path is a one-hop path through which an information flow can go from a low integrity subject to a high integrity subject. We observe that information flows are transitive in general. Therefore, there may exist information flows from a low integrity subject to a high integrity subject via several other subjects. This multi-hop path is called indirect violation path. All direct and indirect violation paths belonging to a domain can construct a policy violation graph for this domain.

Definition 3. *A policy violation graph for a domain* d *is a directed graph* $G^v = (V^v, E^v)$*:*

- $V^v \subseteq V^v_{NTCB} \cup V^v_{TCBd} \cup V^v_{TCB}$ *where* V^v_{NTCB}*,* V^v_{TCBd} *and* V^v_{TCB} *are subject vertices containing in direct or indirect violation paths of domain* d *and belong to NON-TCB,* TCB_d*, and* TCB_s*, respectively.*
- $E^v \subseteq E^v_{Nd} \cup E^v_{dT} \cup E^v_{NT} \cup E^v_{NTCB} \cup E^v_{TCBd} \cup E^v_{TCB}$ *where* $E^v_{Nd} \subseteq (V^v_{NTCB} \times V^v_{TCBd})$*,* $E^v_{dT} \subseteq (V^v_{TCBd} \times V^v_{TCB})$*,* $E^v_{NT} \subseteq (V^v_{NTCB} \times V^v_{TCB})$*,* $E^v_{NTCB} \subseteq (V^v_{NTCB} \times V^v_{NTCB})$*,* $E^v_{TCBd} \subseteq (V^v_{TCBd} \times V^v_{TCBd})$*, and* $E^v_{TCB} \subseteq (V^v_{TCB} \times V^v_{TCB})$*, and all edges in* E^v *are contained in direct or indirect violation paths of domain* d*.*

Figure 2 (a) shows an example of policy violation graph which examines information flows between NON-TCB and TCB_d[1]. Five direct violation paths are identified in this graph: ¡S'_1, S_1¿, ¡S'_2, S_2¿, ¡S'_3, S_2¿, ¡S'_4, S_4¿, and ¡S'_5, S_4¿, crossing all the boundaries between NON-TCB and TCB_d. Also, eight indirect violation paths exist. For example, ¡S'_2, S_5¿ is a four-hop violation path passing through other three TCB_d subjects S_2, S_3, and S_4.

5.2 Ranking Policy Violation Graph

In order to explore more features of policy violation graphs and facilitate efficient policy violation detection and resolution, we introduce a scheme for ranking policy violation graphs. There are two steps to rank a policy violation graph. First, TCB_d subjects in the policy violation graph are ranked based on dependency relationships among them. The rank of a TCB_d subject shows reachable probability of low integrity information flows from NON-TCB subjects to the TCB_d subject. In addition, direct violation paths in the policy violation graph are evaluated based on the ranks of TCB_d subjects to indicate severity of these paths which allow low integrity information to reach TCB_d subjects. The ranked policy violation graphs are valuable for a system administrator

[1] Similarly, the information flows between NON-TCB and TCB_s, and between TCB_d and TCB_s can be examined accordingly.

as they need to estimate the *risk level* of a system and provide a guide for choosing appropriate strategies for resolving policy violations efficiently.

Ranking Subjects in TCB_d. Our notation of *SubjectRank* (SR) in policy violation graphs is a criterion that indicates the likelihood of low integrity information flows coming to a TCB_d subject from NON-TCB subjects through direct or indirect violation paths. The ranking scheme we introduce in this section adopts a similar process of rank analysis applied in hyper-text link analysis system, such as Google's PageRank [6] that utilizes a link structure provided by hyperlinks between web pages to gauge their importance. Comparing with PageRank which focuses on analyzing a web graph where the entries are *any* web pages contained in the web graph, the entries of low integrity information flows to TCB_d subjects in a policy violation graph are only identified NON-TCB subjects.

Consider a policy violation graph with *N* NON-TCB subjects, and s_i is a TCB_d subject. Let $N(s_i)$ be the number of NON-TCB subjects from which low integrity information flows could come to s_i, $N^{'}(s_i)$ the number of NON-TCB subjects from which low integrity information flows could *directly* reach to s_i, $In(s_i)$ a set of TCB_d subjects pointing to s_i, and $Out(s_j)$ a set of TCB_d subjects pointed from s_j. The probability of low integrity information flows reaching a subject s_i is given by:

$$SR(s_i) = \frac{N(s_i)}{N}\left(\frac{N^{'}(s_i)}{N(s_i)} + \left(1 - \frac{N^{'}(s_i)}{N(s_i)}\right) \sum_{s_j \in In(s_i)} \frac{SR(s_j)}{|Out(s_j)|}\right) \quad (1)$$

SubjectRank can be interpreted as a *Markov Process*, where the states are TCB_d subjects, and the transitions are the links between TCB_d subjects which are all evenly probable. While a low integrity information flow attempts to reach a high integrity subject, it should select an entrance (a NON-TCB subject) which has the path(s) to this subject. Thus, the possibility of selecting correct entries to a target subject is $\frac{N(s_i)}{N}$. After selecting correct entries, there still exist two ways, through direct violation paths or indirect violation paths, to reach a target subject. Therefore, the probability of flow transition from a subject is divided into two parts: $\frac{N^{'}(s_i)}{N(s_i)}$ for direct violation paths and $1 - \frac{N^{'}(s_i)}{N(s_i)}$ for indirect violation paths. The $1 - \frac{N^{'}(s_i)}{N(s_i)}$ mass is divided equally among the subject's successors s_j, and $\frac{SR(s_j)}{|Out(s_j)|}$ is the rank value derived from s_j.

Figure 2 (b) displays a result of applying Equation (1) to the policy violation graph showing in Figure 2 (a). Note that even though subject s_4 has two direct paths from NON-TCB subjects like subject s_2, the rank value of s_4 is higher than the rank value of s_2, because there is another indirect flow path to s_4 (via s_3).

Ranking Direct Violation Path. We further define *PathRank* (PR) as the rank of a direct violation path[2], which is a criterion reflecting the severity of the violation path through which low integrity information flows may come to TCB_d subjects. Direct violation paths are regarded as the entries of low integrity data to TCB_d in policy violation

[2] It is possible that a system administrator may also want to evaluate indirect violation paths for violation resolution. In that case, our ranking scheme could be adopted to rank indirect violation paths as well.

graph. Therefore, the ranks of direct violation paths give a guide for system administrator to adopt suitable defense countermeasures for solving identified violations.

To calculate *PathRank* accurately, three conditions are needed to be taken into account: (1) the number of TCB_d that low integrity flows can reach through this direct violation path; (2) SubjectRank of reached TCB_d subjects; and (3) the number of hops to reach a TCB_d subject via this direct violation path.

Suppose $< s_i^{'}, s_j >$ is a direct violation path from a NON-TCB subject $s_i^{'}$ to a TCB_d subject s_j in a policy violation graph. Let $Reach(< s_i^{'}, s_j >)$ be a function returning a set of TCB_d subjects to which low integrity information flows may go through a direct violation path $< s_i^{'}, s_j >$, $SR(s_l)$ the rank of a TCB_d subject s_l, and $H_s(s_i^{'}, s_l)$ a function returning the hops of the shortest path from a NON-TCB subject $s_i^{'}$ to a TCB_d subject s_l. The following equation is utilized to compute a rank value of the direct violation path $< s_i^{'}, s_j >$.

$$PR(< s_i^{'}, s_j >) = \sum_{s_l \in Reach(< s_i^{'}, s_j >)} \frac{SR(s_l)}{H_s(s_i^{'}, s_l)} \tag{2}$$

Figure 2 (c) shows the result using the above-defined equation to calculate the *PathRank* of the example policy violation graph. For example, $< s_2^{'}, s_2 >$ has a higher rank than $< s_1^{'}, s_1 >$, because $< s_2^{'}, s_2 >$ may result in low integrity information flows to reach more or important TCB_d subjects than $< s_1^{'}, s_1 >$.

5.3 Evaluating Trustworthiness

Let P_d be a set of all direct violation paths in a policy violation graph. The entire rank, which can be considered as a risk level of the system, can be computed as follows:

$$RiskLevel = \sum_{< s_i^{'}, s_j > \in P_d} PR(< s_i^{'}, s_j >) \tag{3}$$

The calculated risk level could reflect the trustworthiness of an attestee. Generally, the lower risk level indicates the higher trustworthiness of a system. When an attestation is successful and there is no violation path being identified, the risk level of the attested system is *zero*, which means an attestee has the highest trustworthiness. On the other hand, when an attestation is failed, corresponding risk level of a target system is computed. A *selective service* could be achieved based on this fine-grained attestation result. That is, the number of services provided by a service provider to the target system may be decided with respect to the trust level of the target system. On the other hand, a system administrator could refer to this attestation result as the evaluation of her system as well as guidelines since this quantitive response would give her a proper justification to adopt countermeasures for improving the system's trustworthiness.

6 Implementation and Evaluation

We have implemented DR@FT to evaluate its effectiveness and performance. Our attestee platform is a Lenovo ThinkPad X61 with Intel Core 2Duo Processor L7500

1.6GHz, 2 GB RAM, and Atmel TPM. We enable SELinux with the default policy based on the current distribution of SELinux [15]. To measure the attestee system with TPM, we update the Linux kernel to 2.6.26.rc8 with the latest IMA implementation [1], where SELinux hooks and IMA functions are enabled. Having IMA enabled, we configure the measurement of the attestee information. After the attestee system kernel is booted, we mount the `sysfs` file system and inspect the measurement list values in `ascii_runtime_measurements` and `ascii_bios_measurements`.

6.1 Attestation Implementation

We start from a legitimate attestee and make measurements of the attestee system for the later verification. To invoke a new attestation request from the attester, the attestation reporting daemon runs in the attestee and monitors the attestee system. This daemon is composed of two main threads: One monitors and gets the new system state measurements, and the other monitors and obtains the policy updates of the attestee. The daemon is also measured and the result can be obtained through the legitimate attestee. Thus the integrity of the daemon can be verified later by the attester. In case the attestee system state is updated due to new software installation, changing policy, and so on, an appropriate thread of the daemon automatically obtains the new measurement values as discussed in 4. The daemon then securely transfers the attestation information to the attester based on the security mechanisms supported by the trusted authority.

After receiving the updated system information from the attestee, the measurement module of the attester checks the received measurements against the stored PCR to prove its integrity. To analyze the possible revised attestee policy, the policy analysis module is developed as a daemon, which is ported from a policy analysis engine. We extend the engine to identify violated information flows from the updated policy rules based on domain-based isolation rules. We also accommodate the algorithm presented in Section 4.2, as well as our rank scheme to evaluate the trustworthiness of the attestee.

6.2 Evaluation

To assess the proposed attestation framework, we attest our testbed platform with Apache web server installed. To configure the trusted subject list of the Apache domain, we first identify the TCB_s based on the reference monitor-based TCB identification, including the integrity measurement, monitoring agents, and daemon. For TCB_d of the Apache, we identify the Apache information domain, Apache TCB_d, including `httpd_t` and `httpd_suexec_t`, and the initial filters `sshd_t, passwd_t, su_t`, through the domain-based isolation principles. Both TCB_s and TCB_d are identified with a graphical policy analysis tool [17]. We then install the unverified codes and data to evaluate the effectiveness of our attestation framework.

Installing Malicious Code. We first install a Linux rootkit, which gains administrative control without being detected. Here, we assign the rootkit with the domain `unconfined_t` that enables information flows to domain `initrc_t` labeling `initrc` process, which belongs to TCB_s of the attestee. Following the framework proposed in Section 4, the attestee system is measured from the bootstrap with configured IMA. After

getting the new measurement values, the reporting daemon sends these measurements to the attester. Note that there is no policy update in this experiment. Different from IMA, we only measure the TCB_s and TCB_d subjects. After getting the measurements from the attestee, attester verifies them by trying to match the measured hash values. Partial of our measurement shows the initial measurements of the `initrc` (in a trusted initial system state) and the changed value because of the installed rootkit. The difference between these two measurements indicates the original `initrc` is altered, and the attester confirms that the attestee is not in a trusted state.

Installing Vulnerable Software. In this experimentation, we install a vulnerable software called Mplayer on the attestee side. Mplayer is a media player and encoder software which is susceptible to several integer overflows in the real video stream dumuxing code. These flaws allow an attacker to cause a denial of service or potentially execution of the arbitrary code by supplying a deliberately crafted video file. After a Mplayer is installed, a Mplayer policy module is also loaded into the attestee policy. In this policy module, there are several different subjects such as `staff_mplayer_t`, `sysadm_mplayer_t`. Also, some objects are defined in security policies such as `user_mplayer_home_t` and `staff_mplayer_home_t`.

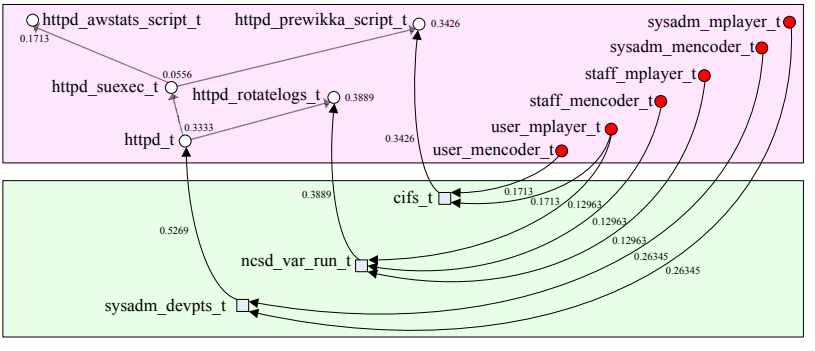


Fig. 3. Information flow verification of Mplayer. The links show the information flow from Mplayer (filled circle nodes) to Apache (unfilled nodes). The rank values on the paths indicate the severity of the corresponding violation paths.

After the Mplayer is installed, the attestation daemon finds that the new measurement of Mplayer is generated and the security policy of the system is changed. As the Mplayer does not belong to TCB_s and Apache TCB_d, the attestation daemon does not need to send the measurements to the attester. Consequently, the daemon only computes the security policy updates and sends the information to the attester.

Upon receiving the updated policies, we analyze these updates and obtain a policy violation graph as shown in Figure 3. Through objects such as `cifs_t, sysadm_devtps_t, ncsd_var_run_t`, information flows from Mplayer can reach Apache domain. In addition, rank values are calculated and shown in the policy violation graph, which guides effective violation resolutions. For example, there are three higher ranked

paths including path from sysadm_devpts_t to httpd_t, from ncsd_var_run_t to httpd_rotatelogs_t, and from cifs_t to httpd_prewikka_script_t. Meanwhile, a risk level value (1.2584) reflecting the trustworthiness of the attestee system is computed based on the ranked policy violation graph.

Once receiving the attestation result shown in Figure 3, the attestee administrator solves the violation that has the higher rank than others. Thus, the administrator can first resolve the violation related to httpd_t through introducing httpd_sysadm_devpts_t.

```
allow httpd_t httpd_sysadm_devtps_t:chr_file {ioctl read write
getattr lock append};
```

After the policy violation resolution, the risk level of the attestee system is lowered to *0.7315*. Continuously, after the attestee resolves all the identified policy violations and the risk level is decreased to be *zero*, the attestation daemon gets a new policy update file and sends it to the attester. Upon receiving this file, the attester verifies whether these information flows violate domain-based isolation integrity rules since these flows are within the NON-TCB–even though there are new information flow compared to the stored $Policy_0$. Thus, an attestation result is generated which specifies the risk level (in this case, *zero*) of the current attestee system. Consequently, a new trusted system state is built for the attestee. In addition, the information of this new trusted system state is stored in the attester side for the later attestation.

6.3 Performance

To examine the scalability and efficiency of DR@FT, we investigate how well the attestee measurement agent, attestation daemon, and the attester policy analysis module scale along with the increased complexity, and how efficiently DR@FT performs by comparing it with the traditional approaches.

In DR@FT, the important situations influencing the attestation performance include system updates and policy changes. Hence, we evaluate the performance of DR@FT by changing codes and data to be measured and modifying the security policies. Based on our study, we observe that normal policy increased or decreased no more than 40KB when installing or uninstalling software. Also, a system administrator does not make the enormous changes over the policy. Therefore the performance is measured with the range from zero to around 40KB in terms of policy size.

Performance on the attestee side. Based on DR@FT, the attestee has three main factors influencing the attestation performance. (1) Time spent for the measurement: Based on our experimentation, the measurement time increases roughly linearly with the size of the target files. For example, measuring policy files with 17.2MB and 20.3MB requires 14.63 seconds and 18.26 seconds, respectively. Measuring codes around 27MB requires 25.3sec. (2) Time spent for identifying policy updates $T_{Pupdate}$: Based on the specification in Section 4, policy updates are required to be identified and sent to the attester. As shown in Table 1, for a system policy which is the size of 17.2MB at its precedent state, the increase of the policy size requires more time for updating the

Table 1. Attestation Performance Analysis (in seconds)

Policy Change	**Dynamic**				**Static**		
Size	$T_{Pupdate}$	T_{send}	$T_{Panalysis}$	**Overhead**	T_{Psend}	$T_{Panalysis}$	**Overhead**
0	0.23	0	0	0.23	14.76	90.13	104.89
-0.002MB (Reduction)	0.12	0.002	0.02	0.14	14.76	90.11	104.87
-0.019MB (Reduction)	0.08	0.01	0.03	0.12	14.74	89.97	104.34
-0.024MB (Reduction)	0.04	0.02	0.03	0.09	14.74	89.89	104.23
0.012MB (Reduction)	0.37	0.01	0.03	0.41	14.77	90.19	104.96
0.026MB (Addition)	0.58	0.02	0.03	0.63	14.78	90.33	105.11
0.038MB (Addition)	0.67	0.03	0.04	0.74	14.79	90.46	105.25

policy and vice versa. (3) Time spent for sending policy updates T_{Psend}: Basically, the more policy updates, the higher overhead was observed.

Performance on the attester side. In DR@FT, the measurement verification is relatively straightforward. At the attester side the time spent for policy analysis $T_{Panalysis}$ mainly influences its performance. As shown in Table 1, the analysis time roughly increases when the policy change rate increases.

Comparison of dynamic and static attestation. To further specify the efficiency of DR@FT, we compare the overhead of DR@FT with a static attestation. In the static approach, the attestee sends all system state information to an attester, and the attester verifies the entire information step by step. As shown in Table 1, the time spent for static attestation is composed of T_{Psend} and $T_{Panalysis}$, which represent the time for sending policy module and analyzing them, respectively. Obviously, the dynamic approach can dramatically reduce the overhead compared to the static approach. It shows that DR@FT is an efficient way when policies on an attestee are updated frequently.

7 Conclusion

We have presented a dynamic remote attestation framework called DR@FT for efficiently verifying if a system satisfies integrity protection property and indicates integrity violations which determine its trustworthiness level. The integrity property of our work is based on an information flow-based domain isolation model, which is utilized to describe the integrity requirements and identify integrity violations of a system. To achieve the efficiency and effectiveness of remote attestation, DR@FT focuses on system changes on the attestee side. We have extended a powerful policy analysis engine to represent integrity violations with the rank scheme. In addition, our results showed that our dynamic approach can dramatically reduce the overhead compared to static approach. We believe such an intuitive evaluation method would help system administrators reconfigure the system with more efficient and strategic manner.

There are several limitations of our attestation framework. First, DR@FT can attest dynamic system configurations, but it does not attest the trustworthiness of dynamic contents such as application state and CPU context. Second, our risk evaluation does not explain under what kind of condition, what range of the risk value is acceptable for the attestee or attester. In addition, in our work, all verification work is done at the

attester side. There is a possibility that the attester can delegates some attestation tasks to trusted components at the attestee side. In the future, we would further investigate these issues.

References

1. LIM Patch, http://lkml.org/lkml/2008/6/27
2. Trusted computing group, https://www.trustedcomputinggroup.org/home
3. Trusted Computer System Evaluation Criteria. United States Government Department of Defense (DOD), Profile Books (1985)
4. Anderson, A.P.: Computer security technology planning study. ESD-TR-73-51 II (1972)
5. Biba, K.J.: Integrity consideration for secure compuer system. Technical report, Mitre Corp. Report TR-3153, Bedford, Mass. (1977)
6. Brin, S., Page, L.: The anatomy of a large-scale hypertextual Web search engine. Computer networks and ISDN systems 30(1-7), 107–117 (1998)
7. Chen, L., Landfermann, R., Löhr, H., Rohe, M., Sadeghi, A.-R., Stüble, C.: A protocol for property-based attestation. In: ACM STC (2006)
8. Haldar, V., Chandra, D., Franz, M.: Semantic remote attestation: a virtual machine directed approach to trusted computing. In: USENIX Conference on Virtual Machine Research And Technology Symposium (2004)
9. Jaeger, T., Sailer, R., Shankar, U.: Prima: policy-reduced integrity measurement architecture. In: ACM SACMAT (2006)
10. Jaeger, T., Sailer, R., Zhang, X.: Analyzing integrity protection in the selinux example policy. In: USENIX Security (2003)
11. Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: 12th USENIX Security Symposium, p. 11 (August 2003)
12. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementation of a tcg-based integrity measurement architecture. In: USENIX Security (2004)
13. Sandhu, R.S.: Lattice-based access control models. IEEE Computer 26(11), 9–19 (1993)
14. Shankar, U., Jaeger, T., Sailer, R.: Toward automated information-flow integrity verification for security-critical applications. In: NDSS (2006)
15. Smalley, S.: Configuring the selinux policy (2003), http://www.nsa.gov/SELinux/docs.html
16. Fraser, T.: Lomac: Low water-mark integrity protection for cots environment. In: Proceedings of the IEEE Symposium on Security and Privacy (May 2000)
17. Xu, W., Zhang, X., Ahn, G.-J.: Towards system integrity protection with graph-based policy analysis. In: Gudes, E., Vaidya, J. (eds.) Data and Applications Security XXIII. LNCS, vol. 5645, pp. 65–80. Springer, Heidelberg (2009)