Trusted Isolation Environment: An Attestation Architecture with Usage Control Model

Anbang Ruan[1,2], Qingni Shen[1,2], Liang Gu[2,3], Li Wang[1,2],
Lei Shi[2,3], Yahui Yang[1,2], and Zhong Chen[1,2,3]

[1] School of Software and Microelectronics, Peking University, Beijing, China
[2] Key Laboratory of High Confidence Software Technologies,
Peking University, Beijing, China
[3] Institute of Software, School of EECS, Peking University, Beijing, China
{ruanab,shenqn,wangl,shil,chen}@infosec.pku.edu.cn,
yhyang@ss.pku.edu.cn, guliang05@sei.pku.edu.cn

Abstract. The Trusted Computing Group (TCG) proposed remote attestation as a solution for establishing trust among distributed applications. However, current TCG attestation architecture requires challengers to attest to every program loaded on the target platform, which will increase the attestation overhead and bring privacy leakage and other security risks. In this paper, we define a conceptual model called the Trusted Isolation Environment (TIE) to facilitate remote attestation. We then present the implementation of TIE with our tailored Usage CONtrol model ($UCON_{RA}$) and a set of system-defined policies. With its continuous and mutable feature and obligation support, we construct the TIE with flexibility. Lastly, we propose our attestation architecture with $UCON_{RA}$ gaining the benefits of scalable and lightweight.

Keywords: Remote attestation, trusted computing, usage control, MAC model, isolation.

1 Introduction

Problems brought about by malicious applications are becoming severer as the widely use of distributed applications. There are increasing needs to securely identify the software stacks running on remote systems. People may want to determine that services advertised by a remote server really exist and that the system is not compromised, e.g. the E-banking they are interacting with has not been tampered with. Meanwhile, a server needs to confirm that its remote clients will behave in the expected way. For instance, a game client is not fabricated and will abide the game's rules.

To equip applications (or *challengers*) with the capability of identifying the genuine behaviors of their interacting peers (or *targets*), the Trusted Computing Group (TCG) [1] introduced remote attestation as an important feature in TCG specifications to validate the configuration states of remote platforms (or *target platforms*). TCG-compliant system can build a trust chain from an immutable booting base called the Core Trusted Root for Measurement (CRTM), ultimately to applications [2] by iteratively measuring

each part of the boot sequence. With the measurement of the trust chain, corresponding measurement logs and a known-good measurements list, the challenger can then verify whether the target platform has specific configurations.

In order to discover the existence of malicious programs, the TCG attestation requires the challenger to verify every program loaded on the target platform [1], [2], [3], [4], [5], namely the target, the target's dependencies and all other unrelated programs. However, to attest to those unrelated programs will not only introduce unnecessary overheads, but also make the list of known-good measurements unmanageable [3]. Moreover, to expose information of all programs running on the system will incur privacy leakage and increase security risks [4], [5]. A variety of approaches have been proposed for these deficiencies. However, most of them introduce different limitations. For example, some rely on well-configured security policies on the target platforms [3], [16]. They will introduce extra management complexities especially when the number of applications grows. Another line of works demand modifications to software [13], [14]. Hence they may only be applied to particular kinds of applications. Property-based attestation [4], [5] enhances privacy and scalability for the target platform. However, to clearly define and classify appropriate properties for general applications is still an open issue.

In this paper, we propose a Trusted Isolation Environment (TIE) for improving TCG attestation architecture, with our following efforts: a) With our Usage CONtrol model [6] for Remote Attestation ($UCON_{RA}$) and a set of system-defined $UCON_{RA}$ policies, we construct the TIE for a target application in a dynamic and scalable way, without the needs to manually specify the access control policies for managing complicated access relationship. b) Most operations of the attestation are performed in the target platform, gaining more controllable attestation granularity and timing (with the trustworthy guarantee from underlying trusted hardware [9], [10] and related protection mechanisms [12], [14], [18] to protect the local attestation mechanisms). c) A lightweight and easy-to-implement attestation architecture is proposed in this paper. Moreover, with practices [15], [17] in MAC (Mandatory Access Control) models and enforcement mechanisms, our architecture is easy to be integrated into exist systems.

The remainder of the paper is organized as follows: Section 2 presents the definition and requirements for the Trusted Isolation Environment. Section 3 describes the formal specification for $UCON_{RA}$ and corresponding system-defined policies to implement the semantics of the TIE. Our attestation architecture with $UCON_{RA}$ is then illustrated in Section 4. Section 5 presents a case study on Firefox for our architecture. Section 6 discusses some related issues. Section 7 summarizes the related works and section 8 concludes this paper and presents our future works.

2   Trusted Isolation Environment

Commonly, an application needs to interact with other software components on the same system to implement its own functionalities. For instance, during an application's execution, it may need to load extra libraries or to read some data or configuration files. We denote those components, whose integrity will subvert the genuine behavior of a program, as the program's *dependencies* (or the *operating-system-level dependencies* [8]). They represent the program's functionalities. As the loading process
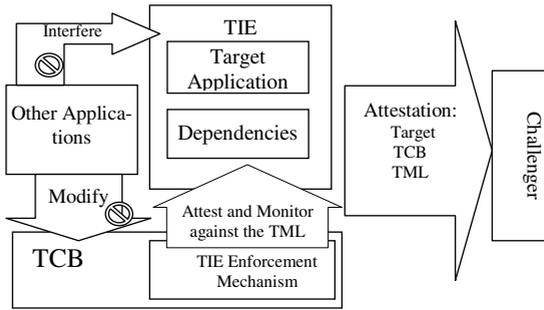
Fig. 1. Remote Attestation with TIE

continues iteratively, a *dependency tree* can be built for a program, with the program as the root of the tree. On the other hand, even if every node on a dependency tree is intact, their behaviors still depend on the genuinely execution of their codes or the accurate interpretation of their data. For example, the subversion of the kernel system-call table will tamper with the genuine behaviors of a program even if its codes are not modified. We denote these instruction-level dependencies [8] to implement the behaviors of the target and its dependencies, whether directly or indirectly, as the target's *Trusted Computing Base* (*TCB*) (see Figure 1), which usually includes the OS kernel, the boot-loader, the BIOS and the CRTM. They represent the capabilities of fulfilling target's functionalities.

In this paper, as depicted in Figure 1, we propose to isolate all entities in a target's dependency tree in a Trusted Isolation Environment (TIE). A TIE for a target can guarantee that, at operating-system-level, a) only entities expected by the target can be imported; b) all entities inside are measured and trusted by the target; c) the integrity of the entities inside is continually protected. A TIE is constructed when a target application is loaded, which is called the TIE Entrance (TIEE). The TIEE can specify an initial Trusted Measurement List (TML), which contains a list of the TIEE's expected dependencies, their genuine measurement values and optional attestation method specifications. Any program loaded in the TIE can also specify its TML to declare its genuine dependencies. Hence, all TMLs define the dependency tree for the target. Entities must be attested to by their parents in the tree when they are entering the TIE (being loaded by entities inside the TIE). All entities can then be trusted by the target in an iterative way and, ultimately, by the challenger. Meanwhile, a TML can also specify rules for inspecting the program's critical state. With the isolation and attestation mechanism of the TIE, a challenger only needs to attest to the target, its TCB and its TML (Figure 1).

2.1   Threat Model

As depicted in Figure 2, adversaries can interfere with the target through three kinds of objects: User Space Objects (USO), Kernel Space Objects (KSO) and Kernel Objects (KO). The USOs are objects existing in the file system (data, libraries). They comprise the most parts of the target' dependencies tree. KSOs are kernel services specified to a program, such as IPC objects, semaphore, etc. They also represent the
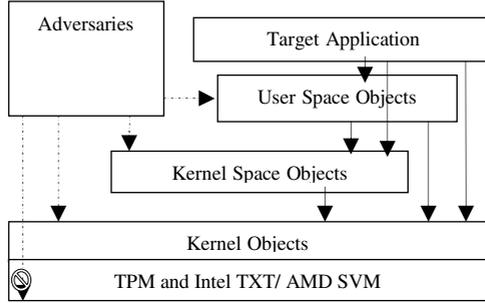
Fig. 2. Threat Model

target's functionalities. Hence they are the target's dependencies. KOs are kernel components (resources and functionalities), e.g. the system call table, kernel codes. They are important parts of the target's TCB. On the other hand, we assume that the adversaries cannot tamper with the secure hardware: the TPM [1], Intel TXT [9] or AMD SVM [10].

Therefore, an isolated and protected environment must be provided for USOs and KSOs. Moreover, the integrity of KOs and the isolation mechanism must also be protected. The former is implemented as the TIE isolation mechanism and the latter as the TIE protection mechanism. Min Xu et al. [12] presented a tailored Usage Control Model (UCON) [6] for kernel-integrity protection (UCON$_{KI}$). They regarded the OS kernel as a set of Kernel Objects (KO) and can then monitor accesses to them with UCON$_{KI}$ rules. Because our solution is also derived from UCON, it is easy to integrate UCON$_{KI}$ into our model. Moreover, the TIE protection mechanism can be facilitated by the trustworthy guarantee from underlying secure hardware [9], [10], and additional protection mechanisms [12], [14], [18]. Therefore, we will focus on the TIE isolation mechanisms in this paper.

3   Usage Control Model for Remote Attestation

In this section, we will first present an overview of the Usage Control model [6] and our Usage CONtrol model for Remote Attestation (UCON$_{RA}$). Afterwards, we will present the formal definition of UCON$_{RA}$, together with our system-defined policies.

3.1   Usage Control Model Overview

Usage control model (UCON) [6] proposed by Jaehong Park and Ravi Sandhu possesses two distinct features over traditional access control models: *continuity* and *mutability*. The *continuity* means that an access decision is not only made before access but also during access and may result in revocation of access permissions if particular conditions are no longer satisfied. The *mutability* means that attributes of subjects or objects may change as side effects of access, which may also result in a change in ongoing or subsequent access decisions. Policy statements in UCON are categorized as *authorizations*, *obligations* and *conditions*. *Authorizations* refer to functional predicates that have

to be evaluated for usage decisions before the access action (*pre-authorization*) or while the action is performing (*on-authorization*). *Obligations* are mandatory actions that a subject has to perform before obtaining (*pre-obligation*) or exercising rights on an object (*on-obligation*).

3.2   TIE Isolation with UCON$_{RA}$

In this paper, we present a tailored Usage CONtrol model (UCON) [6] called UCON$_{RA}$ (Usage CONtrol for Remote Attestation) to construct and maintain a Trusted Isolation Environment (TIE) for the target application and its dependencies (USO and KSO). In our model, a special label called *tag* is defined for TIE identification. We hence specify a simple authorization rule for TIE isolation: subjects can only access the objects with the same tag. We then define a set of policies as the framework for implementing TIE semantics, which are called the *system-defined policies*. We utilize the *mutability* property and *obligation* of UCON for *TIE entrance attestation*: when a subject inside a TIE trying to access entities outside, a *pre-obligation* will first be executed to measure the entities and verify the measurements against the subject's TML. The expected entities will be assigned with the tag of the TIE, allowing all subsequent access requests. Finally, with the *continuity* feature, we define *on-obligation* and *on-authorization* for TIE introspection. They are executed each time when particular rights are exercised, and can move the compromised entities out of the TIE by revoking their tags.

Unlike most MAC models, which manage complicated access relationship, the main concern of UCON$_{RA}$ is to construct and maintain an isolated environment. Therefore, only rules specified above are necessary. Subjects and objects in UCON$_{RA}$ are discovered and labeled at runtime (by obligations), and obligation actions for attestation and monitors are specified in a TML. A TML is defined and issued by the software vendor who has the best knowledge of the software's security requirements and can be refined by the challenger in an attestation session, who can best defined the security requirements for that session. Hence a TIE can be constructed in scalable and flexible way while introducing only a few management overheads.

3.3   UCON$_{RA}$ Definition

In this section, we will examine the components of our tailored usage control model UCON$_{RA}$. We use the UCON definition in [6].

Definition 1. *A UCON$_{RA}$ model has seven components:*

Subjects (S): active processe;

Objects (O): USO (files), KSO (IPC objects, etc.);

Subject attributes (ATT(S)): tag, text, certificates, TML;

Object attributes (ATT(O)): tag, text, certificates;

Rights (R): access;

Authorizations (A): functional predicates that have to be evaluated for usage decisions.

Obligation (OB): mandatory actions that a subject has to perform before obtaining or exercising rights.

Subjects are mainly active processes. Two kinds of objects are concerned: USO and KSO. Tags specify the TIE the entity (subject or object) belonging to. Text, certificates and the TML are essential for attestation. Only the right "access" is needed for TIE isolation.

An authorization is a functional predicate that has to be evaluated for usage decisions. It can be either evaluated before or while the requested right is exercising, which are called *pre-authorization* and *on-authorization* respectively.

Definition 2. *Pre-authorization:*
   preA (ATT(S), ATT(O), R)
   DEFINITION 3. *On-authorization:*
   onA (ATT(S), ATT(O), R)

Pre-authorization examines usage requests using ATT(S), ATT(O), and R, and then decides whether the request is allowed or not. On-authorization implements the continuous feature of UCON [6]. It can revoke access right dynamically when specific predicates (onA) are not satisfied. The evaluating process can be performed periodically based on time or event.

Obligations specify mandatory actions (*obligation actions*) that a subject has to perform before obtaining or on exercising rights on an object (*pre-obligation* and *on-obligation* respectively). Both kinds of obligations can be performed with *postUpdate* actions, which update attributes of specific subjects or objects as a side-effect.

Definition 4. *Pre-obligation with postUpdate:*

   OBS, OBO, and OB: (obligation subjects, obligation objects, and obligation actions, respectively);
   preB : and preOBL, (pre-obligations predicates and pre-obligation actions, respectively);
   preOBL: OBS × OBO × OB;
   preFulfilled: OBS × OBO × OB   {true, false};
   getPreOBL: S × O × R 2^{preOBL}, a function to select pre-obligations for a requested usage;
   preB(s, o, r) = $\bigwedge(obs_i,obo_i,ob_i)$ getPreOBL(s,o,r) preFulfilled(obs_i, obo_i, ob_i);
   preB(s, o, r) = true if getPreOBL(s, o, r) =   ;
   postUpdate(ATT(s)), postUpdate(ATT(o)): an optional procedure to change certain attributes as a consequence of pre-obligations.

The preB predicate evaluates if all the required pre-obligation actions (preOBL) are fulfilled by using preFulfilled and returns either true or false. GetPreOBL decides on what kind of actions are required for requests. OBS is the entity who has to perform the action. OBO is the entity on which the action has to be performed. OB represents what has to be performed. The postUpdate specifies the attributes of subjects or objects to update after the obligation.

Definition 5. *On-obligation:*

   T, a set of time or event elements;
   onB and onOBL, (ongoing-obligations predicates and ongoing-obligation elements, respectively);

onOBL OBS × OBO × OB × T ;
getOnOBL: S × O × R 2^{onOBL}, a function to select ongoing-obligations for a re-
quested usage;
onFulfilled: OBS × OBO × OB × T   {true, false};
onB(s, o, r) = $\wedge$(obs$_i$,obo$_i$,ob$_i$,t$_i$) getOnOBL(s,o,r) onFulfilled(obs$_i$, obo$_i$, ob$_i$, t$_i$);
onB(s, o, r) = true if getOnOBL(s, o, r) =   ;

On-obligations fulfill obligation actions, either periodically or continuously, while the
rights are being exercised. Most actions are similar with the ones defined in pre-
obligation. Particularly, a time parameter T is introduced as part of obligation actions
on OBL. T is likely to define certain time intervals that are either time-based or event-
based. OnB must be assumed to be true all the time though actual obligation verifica-
tion intervals can vary.

3.4 System-Defined Policies

To implement the TIE functionalities, we must first confine communications within
entities in the same TIE, and then make sure only trusted entity can enter the TIE.
Lastly, the dynamic state of a TIE can be introspected. We hence define three polices:
pre-authorization for TIE confinement, pre-obligation for TIE entrance attestation,
and on-obligation/on-authorization for TIE monitors.

Two kinds of decisions can be defined in UCON$_{RA}$.

Definition 6. *Allowed decision:*
   allowed(s, o, r)$\Rightarrow$ predicates

Definition 7. *Stopped decision:*
   stopped(s, o, r)$\Rightarrow$ ¬predicates

The allowed(s, o, r) indicates that subject s is allowed right r to object o. According to
[6], the 'implies' connectives is used, which represents the 'necessary condition'.
Henceforth, the decision process can include other rules that might be necessary for
finer and richer controls. The 'stopped(s, o, r)' indicates rights r of subject s to object
o is revoked, and the right-hand-side is a 'sufficient condition', by which means the
dissatisfied of any predicates may cause the 'stopped' decision.
   The TIE confinement policy is quite simple: subjects can only access objects with
the same tag.

Definition 8. *The TIE confinement policy:*
   *T is the tag of a subject (s) or an object (o), representing which TIE it is currently
belonging to (NULL for none).*
   Tag: S -> T, O->T
   ATT(S) = (Tag)
   ATT(O) = (Tag)
   preA1: Tag(S) == Tag(O)
   preA2: Tag(S) == NULL && Tag(O) == NULL
   allowed(S,O,access)$\Rightarrow$ preA1 || preA2

When a subject in the TIE requests to access an object outside, the TIE isolation
mechanism will first decide whether the object is trustworthy by measuring it and

then verifying the measurement against the object's certificate and the certificate against its loader's TML. Moreover, the loader can choose to perform the attestation by itself with only a few changes to its TML as described in Section 5. Expected object will be assigned with the same tag as the subject, by which means to be moved into the TIE. This semantic is implemented by a pre-obligation with post tag update.

Definition 9. *TIE entrance attestation obligation:*

> *TXT is the text of an object.*
> *L is the Trusted Measurement List of a subject.*
> *C is the certificate of an object.*
> *Text: O -> TXT ; TML: S -> L; Certificates: O -> C;*
> *Attest: ((Text(O), Certificates(O), TML(S))   {true, false}*
> *ATT(O) = (Text, Certificates)*
> *ATT(S) = (TML)*
> *OBS = Measurement Agent;*
> *OBO = Text(O);*
> *OB = {Attest};*
> *getPreOBL(s,o,r) = {(measurement agent, text(o), attest)} if (tag(s) != NULL &&*
> *tag(o) == NULL)*
> *allowed ⇨ preFulfilled(getPreOBL(s,o,r))*
> *postUpdate(Tag(OBO)): tag(o) = attest(obo, tml(s), certificates(obo)) ?*
> *tag(s):NULL;*

Monitor obligations can inspect the state of a TIE at runtime. It performs particular actions (specified in TML) while subjects are accessing objects, and updates the attributes when necessary. Meanwhile, monitor authorization can perceive the change of the system state by examine the entities' tags while specified rights are exercising.

Definition 10. *Monitor obligation:*

> *OBS = Monitor Agent*
> *OBO = {S, O}*
> *OB = {Monitor}*
> *Monitor: predicates must be satisfied.*
> *T = {always}*
> *getOnOBL(s, o, r) = {(monitor agent, obo, monitor, always)}*
> *postUpdate(Tag(OBO): Tag(OBO) = monitor ? Tag(OBO) : NULL*
> DEFINITION 11. *Monitor authorization:*
> *allowed(s, o, r) ⇨ true*
> *stopped(s, o, r) ⇨(Tag(S) != Tag(O)) || (Tag(S) == NULL && Tag(O) != NULL)*
> *|| (Tag(S) != NULL && Tag(O) == NULL)*

According to UCON, on-obligation can also specify allowed and stopped decisions, to make or revoked usage decision directly. However, in our model, we clearly separated the duty of obligations and authorizations. The former are responsible for TIE entrance or exit, and the latter are for isolation. Authorizations make decisions simply by the tag of both subjects and objects. Meanwhile, obligations utilize other attributes, and affect the access decision through altering the tags. The separation of duties makes it easy to simplify our model and system-defined polices, and hence reduces the management complexities.

Obligations above only provide a framework for specifying the occasions to perform particular obligation actions. Obligation actions for particular applications are commonly specified within the TML by their vendors, and can be revised by the challenger. Hence different kinds of monitors and attestation methods can be customized as needed. With the attestation action specifiable in a TML, we can choose the best-fit attestation methods. Moreover, we can specify particular parts of an entity to be attested, achieving the best attestation granularity. Section 5 will present examples for specifying the attestation methods in a TML.

4   Attestation Architecture with UCON$_{RA}$

In this section, we will first describe our attestation architecture and its functional components. Then we will illustrate how they cooperate in the scenarios of TIE construction and remote attestation.

4.1   Attestation Architecture with TIE

Figure 3 illustrates the overview of our attestation architecture with UCON$_{RA}$. Just as a typical Mandatory Access Control (MAC) enforcement architecture [17], our architecture includes three main components: Policy Enforcement Point (PEP), Attribute Repository (AR) and Policy Decision Point (PDP). PEP is a module to intercept access request from the processes, to pull subject and object attributes from AR, and to enforce access executions. AR is used to store subject and object attributes. PDP delivers authorization decisions according to policies stored in the Policy DB. Obligations are stored in the Policy DB as well. To improve performance, the Access Vector Cache (AVC) component caches access decisions. The TIE Registry maintains an entry for each TIE, recording information for their lifetimes. Our architecture also contains two components for obligation action execution, namely Measurement Agent and Monitor. The component of Monitor is actually a monitor enforcement driver to execute specific scripts or programs specified by the software vendor or the challenger for monitoring various states of a TIE. The integrity of these monitor actions must also be guaranteed. Lastly, Attestation aGent (AG) collects related information and manages attestation sessions with the remote parties.
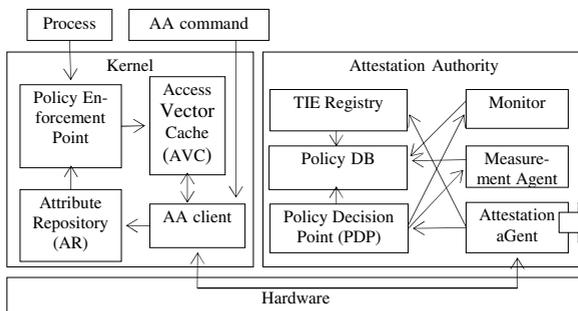

Fig. 3. Attestation Architecture with TIE

Because the attestations for the target's dependencies are performed in the target platform, we move the components for TIE and policy management, obligation action enforcement, and policy decision out of the kernel space into a tiny hypervisor [11], which is called the Attestation Authority (AA), to minimize their TCB and to achieve strong isolation. Therefore an additional component for AA communication is added, named AA Client. Lastly, the AA Command component serves as the interface to user.

4.2 TIE Construction

A TIE is represented as a $UCON_{RA}$ tag. Hence we can control the affiliation of an entity by managing its tag. After we registered the program in the TIE Registry as the TIE Entrance (TIEE), we can initiate it through the AA Command, which will then inform Attestation Authority (AA) to load and parse its Trusted Measurement List (TML) and store obligation actions into Policy DB. AA will measure and load the TIEE and assign it a tag representing the newly created TIE. Afterwards, any entity created by the subjects in the TIE will be assigned with the same tag.

When Policy Decision Point (PDP) receives an access request from a subject inside a TIE to an object outside (with a different or no tag), the attestation obligation is first executed. PDP first searches for particular obligation action, and transfers it (or a default one) to the Measurement Agent (MA), which will attest to the object in the way specified by the obligation action in subject's TML. MA then updates the object's tag in the Attribute Repository (AR) to allow or deny the request. On the other hand, when a subject outside the TIE (untagged) requests to the access an object inside (tagged), Policy Enforcement Point (PEP) will simply deny it, or for shared USOs (Section 5), it will copy the object to a location specific to the TIE and redirect the access requests from the tagged subjects to the new copy. The origin one is then unlabeled, enabling the access request from other untagged subjects.

4.3 Remote Attestation

Figure 4 illustrates our remote attestation procedure. Administrator first informs Attestation aGent (AG) to initiate the TIE Entrance (TIEE) and construct the TIE (step 1.1-1.2). When the TIEE connects to a challenger, AG will first transfer the measurement of the TCB (from the CRTM up to the OS kernel and AA) and the target
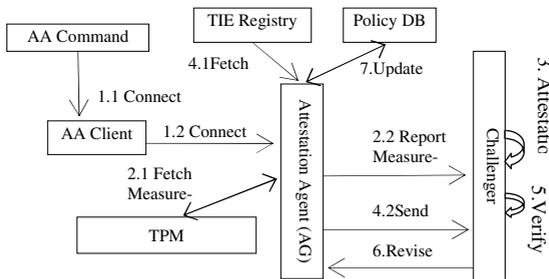

Fig. 4. Remote Attestation Procedure

application to the challenger (step 2.1-2.2) for attestation (step 3), and negotiates the Trusted Measurement List (TML) with the challenger (step 4.1-4.2). The challenger then verifies the TML (step 5) and returns a revised TML to AG when needed (step 6). Afterwards, AG updates the Policy DB in accordance with the new TML (step 7) and then loads the target application. In addition, the PDP and PEP, though not involved in the attestation process directly, are responsible for maintaining the TIE.

5   Case Study

In this section, we will present a case study on Firefox. We will illustrate a TML for the Firefox, and describe its TIE's building and attestation process in our architecture.

5.1   TML Specification for Firefox

Typically, the TML will be defined and signed by the Firefox's vendor. In our case, we downloaded the package of Firefox-3.0.5 from its official site and manually specified its TML by identifying all legal dependencies, calculating their measurements and specifying related measurement methods (if needed). Table 1 shows an abridged view of the TML.

Only the security-related files need to be attested, such as the firefox-bin (the TIEE), the shared libraries (.so files), and all files in the directories of components, modules, and searchplugins. Resource files such as icon files (.gif files) will not affect the genuine behavior of Firefox. Henceforth, their attestation methods and measurements in the TML are specified as NONE. By specifying the Attestation Method entry, our architecture can easily support attestation for particular parts of a file, e.g. for configuration files as browserconfig, we specify that the browser.startup.home page entry must be assigned to the default value.

Web services may demand to load particular plugins, which can also be specified through the TML negotiation phase. For files that might be altered, we specify Mutable flags in their TML entries. When the TIE is exiting, those files are hashed and the measurement entries are updated. Every time files with Shared flags are referenced by entities with a different or no tag, Attestation Agent (AA) will first copy them to a dedicated directory for the current TIE, redirect all existing access requests and untag

Table 1. Abridged TML for Firefox-3.0.5

File Name	Example Measurement	Attestation Method
firefox-bin	C9D250E82A4C3CFED2246E1CF2 A39526E7CC3CC3	Full file
libfreebl3.so	1A34E6AFE7B07FBD1CD3C1D94C 485A68D598B9DC	Full file. Shared
*.gif	NONE	NONE
browsercon- fig.properties	http://en- US.start2.mozilla.com/firefox?client= firefox-a&rls=org. mozilla:en- US:official	ENTRY: brows- er.startu p.homepage
blocklist.xml	E8579795550DF3C2B4C4F474148E BAF98D214C44	Full file. Mutable.

the original one to allow access from those entities (libfreebl3.so). Monitors for state introspecting are usually challenger-specific. Hence they are often specified as obligation actions by the challenger in the TML negotiation phrase. Obligation actions are executed by monitor module specific to a program, which should also provided by software vendors. They will be invoked by the Monitor component of AA according to the system-defined monitor obligation rules and obligation actions in the TML.

One difficulty is that the Firefox's GUI system is depending on the integrity of xorg server, which is commonly started before the TIE of Firefox. Because our architecture currently only supports the attestation for the files to load, we confine the xorg server in another TIE and add an entry in Firefox's TML to refer to the TML of the xorg server. When Firefox is started, AA will compare the xorg TML reference to the TML of the currently running xorg server. If they match, an authorization policy will be added to the Policy DB, permitting communication between the two TIEs.

5.2 TIE Isolation for Firefox

Firefox-bin is first registered as a TIEE in the TIE Registry together with its TML and then initiated through the AA command, which will inform AA to construct a corresponding TIE. AA first loads the TML, calculates the hash value of firefox-bin and then compares the hash with related entry in the TML. If the values match, firefox-bin is loaded and assigned with a $UCON_{RA}$ tag representing its embracing TIE. Otherwise firefox-bin will be loaded as a normal program. Every time when firefox-bin is trying to access entities with no tag assigned, a measurement obligation of $UCON_{RA}$ is executed by the Measurement Agent to attest to the entities against TML and to assign those expected ones with the same tag. For those illegal ones, a different tag is assigned, avoiding repeatedly attestation.

The TIE can resist following kinds of threats in our case: a) replacing the shared libraries. Any access request to those libraries from programs outside the TIE is denied by the TIE isolation policy (Definition 8). The tag value is calculated with a nonce every time when a new TIE is constructed, hence cannot be predicted by malicious programs. For files can be shared among the system, each TIE keeps a dedicated copy. Hence changes to the origin ones cannot tamper with the TIE. b) Instructing Firefox to load unexpected plugins. Before firefox-bin loads the new plugin (untagged), the TIE entrance attestation obligation (Definition 9) is executed, which hashes the plugin and searches the hash value in TML. When there is nothing found, the request is denied, and the access decision is cached. c) Instructing Firefox to connect to a new URL. Since the specific state of firefox-bin can be monitored (by Definition 10 and 11 respectively), any connection to URLs other than those specified in the TML are denied. More sophisticated monitor rules and monitor enforcement components can be designed by software issuer or negotiated with the challenger.

6   Discussion

Because the semantics of a TIE are implemented as an MAC model, the TIE isolation mechanism is easy to be integrated into exist systems. Only a few modules are needed to be added to the existing systems, with most derived from SELinux [15]. The Attestation Authority (AA) can be hosted in a tiny-hypervisor as Secvisor [11] to

achieve high security guarantee, which introduces no more than 2,000 lines of codes and can be efficient. The main overheads for TIE are the MAC enforcement and decision. They are well studied and can be under good control. Moreover, because only target's dependencies are measured, both measurement and verification efforts are reduced. Although the target platform has to perform extra verifications, they are related lightweight, compared to those saved measurement efforts.

Our scheme does not eliminate software vulnerabilities. Instead, we minimized the ways for malicious entities to utilize vulnerabilities of programs in the trust chain by isolating them in a Trusted Isolation Environment (TIE). Our architecture can guarantee the challenger that the target application will behave as the way it is designed, and the challenger can then decide its trustworthiness according to extra information, e.g. reported vulnerabilities, or specify monitors for filtering and introspecting.

$UCON_{RA}$ only concerns about constructing and maintaining the TIE. Pre-authorizations control communications between entities inside the TIE and entities outside. Entrance obligations deal with importing entities into the TIE. Monitor obligations utilize the feature of continuously for dynamic introspecting. However, since our model is derived from usage control model, which can implement the semantics of most prevalent access control models (e.g. Role Based Access Control) by specifying particular authorizations [6], access control inside a TIE can easily be supported.

The Trusted Measurement List (TML) is specified by the issuer of the target application and can be customized by the challenger through the TML negotiation phase (step 6 in Section 4.3). Hence the dependencies can be clearly defined. Meanwhile, they can also be identified with both static program analysis and runtime monitoring [8]. We also considered many ways to reduce the length of a TML, e.g. TML can be referenced and entities can be copied and dedicated to a particular TIE as the xorg and libfreebl3.so case in previous section respectively. For complicated applications with various dependencies, the TML may become very large or may refer to many other TMLs on the target platform. However, the complexity is still relative low comparing to the known-good measurement list for these kinds of applications. We will investigate further on reducing the size of TML.

7   Related Work

Terra [7] uses a Trusted Virtual Machine Monitor (TVMM) that provides the semantics of either an "open box" or a "closed box". The hardware and TVMM can act as a trusted party to allow closed-box to cryptographically identify the software they run to remote parties. However, administrators have to pay extra management overheads to deploy the closed box for each target application. In our scheme, a TIE is constructed at runtime, and administrators only need to register the TIE Entrance (TIEE) and initiate the TIEE. The TIE semantics are implemented as an MAC model; hence they can easily be tailored or enhanced and can be integrated into existing systems with only few efforts, gaining scalable and lightweight. As a result, we avoid the overheads brought by extra virtualization layer, e.g. the hypervisor.

Policy-Reduced Integrity Measurement Architecture (PRIMA) [3] enables the challengers to only verify the applications which are permitted by the policies to interact with the target. Model-based behavior attestation (MBA) [16] attests to the behavior of a model, which is associated with different components of a policy model

and can be attested to separately at runtime. However, as the dimension of the entities scales, the management complexities for MAC policies introduced by them escalate rapidly. The main concern of UCON$_{RA}$ is to construct and maintain an isolated environment. Therefore, only a few polices are needed (Section 3). Tags are specified as a side-effect of measurement, and the measurement is taken automatically as an obligation when an unauthorized access request occurs. Moreover, obligation actions for attestation and monitors are specified in a TML, which are customizable and can be negotiated with the remote party.

BIND (Binding Instructions aNd Data) [13] attests to only the concerned pieces of codes, which greatly simplifies the verification, instead of the entire memory content. Flicker [14] isolates sensitive code execution with hardware support for late launch and attestation introduced in commodity processors from AMD [10] and Intel [9]. However, BIND and Flicker need developers' supports: only the specially designed application can be supported. In our architecture, all legacy applications are supported.

Property-based attestation (PBA) [4], [5] proposes to attest to specific property of a system without receiving detailed configuration data. However, to identify appropriate properties for every application is a complicated job, which makes the PBA hard to implement. Min Xu et al. [12] presented a tailored UCON model for kernel-integrity protection (UCON$_{KI}$). They regarded OS kernel as a set of Kernel Objects (KO) and protected them by UCON$_{KI}$ rules. While UCON$_{KI}$ mainly considers the kernel integrity protection, our focus is to provide a runtime constructed trusted and isolated environment to facilitate remote attestation. Moreover, UCON$_{KI}$ can be easily integrated into our model to provide a bottom-up protection.

8   Conclusion and Future Work

In this paper, we proposed a conceptual model called the Trusted Isolation Environment (TIE) to facilitate remote attestation. With the TIE, only genuine entities expected by the target application can interact with it. Therefore, challengers only need to attest to the TCB of the target platform, the target, and the target's Trusted Measurement List (TML). We then presented our tailored UCON$_{RA}$ model and properly designed policies. With the continuous and mutable feature and obligation support from UCON$_{RA}$, our architecture can be scalable and lightweight. For future works, we will investigate additional mechanisms to reduce the complexity of the TML.

References

1. Trusted Computing Group (TCG),
   https://www.trustedcomputinggroup.org/
2. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture. In: SSYM 2004: Proceedings of the 13th conference on USENIX Security Symposium, Berkeley, CA, USA, p. 16. USENIX Association (2004)

3. Jaeger, T., Sailer, R., Shankar, U.: PRIMA: Policy-Reduced Integrity Measurement Architecture. In: SACMAT 2006: Proceedings of the eleventh ACM symposium on Access control models and technologies, pp. 19–28. ACM Press, New York (2006)

4. Poritz, J., Schunter, M., Van Herreweghen, E., Waidner, M.: Property attestation—scalable and privacy-friendly security assessment of peer computers. Technical Report RZ 3548, IBM Research (May 2004)

5. Sadeghi, A.R., Stüble, C.: Property-based attestation for computing platforms: Caring about properties, not mechanisms. In: The 2004 New Security Paradigms Workshop, Virginia Beach, VA, USA, ACM SIGSAC, September 2004. ACM SIGSAC, ACM Press (2004)

6. Park, J., Sandhu, R.: The UCON$_{abc}$ usage control model. ACM Transactions on Information and Systems Security 7(1) (February 2004)

7. Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., Boneh, D.: Terra: a virtual machine-based platform for trusted computing. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP 2003), pp. 193–206 (2003)

8. Gu, L., Ding, X., Deng, R.H., Xie, B., Mei, H.: Remote Attestation on Program Execution. In: Proceedings of STC 2008, Virginia, USA (October 2008)

9. Intel Corporation: LaGrande technology preliminary architecture specification. Intel Publication no. D52212, May 2006.

10. Advanced Micro Devices. AMD64 virtualization, C.: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01 (May 2005)

11. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: SOSP 2007 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, Washington, USA (October 2007)

12. Xu, M., Jiang, X., Sandhu, R., Zhang, X.: Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection. In: Proceedings of SACMAT 2007, Sophia Antipolis, France (2007)

13. Shi, E., Perrig, A., Van Doorn, A.: BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In: SP 2005: Proceedings of the 2005 IEEE Symposium on Security and Privacy, pp. 154–168 (2005)

14. McCuney, J.M., Parnoy, B., Perrigy, A., Reiteryz, M.K., Isozaki, H.: Flicker: An Execution Infrastructure for TCB Minimization. In: Proceedings of EuroSys 2008, Glasgow, Scotland, UK (April 2008).

15. Security-Enhanced Linux (SELinux), http://www.nsa.gov/selinux/

16. Alam, M., Zhang, X., Nauman, M., Ali, T., Seifert, J.P.: Model-based Behavioral Attestation. In: SACMAT 2008: Proceedings of the thirteenth ACM symposium on Access control models and technologies, ACM Press, New York (2008)

17. Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., Lepreau, J.: The Flask Security Architecture: System Support for Diverse Security Policies. In: Proceedings of the Eighth USENIX Security Symposium, Aug. 1999, pp. 123–139 (1999)

18. Loscocco, P.A., Wilson, P.W., Aaron Pendergrass, J., McDonell, D.: Linux kernel integrity measurement using contextual inspection. In: Proceedings of the 2007 ACM workshop on Scalable trusted computing, November 02-02, 2007, Alexandria, Virginia, USA (2007)