Remote Attestation on Program Execution

Liang Gu[1,2]*, Xuhua Ding[2], Robert H. Deng[2], Bing Xie[1], and Hong Mei[1]
[1]Key Laboratory of High Confidence Software Technologies,
Peking University, Beijing, China
{guliang05, xiebing, meih}@sei.pku.edu.cn
[2]School of Information Systems,
Singapore Management University, Singapore
{xhding,robertdeng}@smu.edu.sg

ABSTRACT

Remote attestation provides the basis for one platform to establish trusts on another. In this paper, we consider the problem of attesting the correctness of program executions. We propose to measure the target program and all the objects it depends on, with an assumption that the Secure Kernel and the Trusted Platform Module provide a secure execution environment through process separation. The attestation of the target program begins with a program analysis on the source code or the binary code in order to find out the relevant executables and data objects. Whenever such a data object is accessed or a relevant executable is invoked due to the execution of the target program, its state is measured for attestation. Our scheme not only testifies to a program's execution, but also supports fine-granularity attestations and information flow checking.

Categories and Subject Descriptors

D.4.6 [**Operating System**]: Security and Protection; D.2.4 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Security

Keywords

Trusted computing, remote attestation, program dependency

1. INTRODUCTION

Remote attestation was initially introduced as one of the basic functions of the trusted computing model proposed by the Trusted Computing Group (TCG) [22]. According to its specification, a TCG-compliant platform (*the attester*), with an embedded Trusted Platform Module (TPM) chip, attests to a remote platform (*the challenger*) with a digital signature upon the hashes of the states of its software components. The challenger evaluates the trustworthiness of the attester by verifying the signatures.

The challenger may require the attester to testify to a specific security property according to the application. The most primitive property, namely the platform integrity, can be measured by TPM directly. Nonetheless, many applications require security assertions on more sophisticated security properties, for example, program semantics [7], behavior [24], or security policy enforcement [11]. The attestations of these properties make use of the basic TCG attestation as a building block and augment it with other security mechanisms.

In this paper, we deal with how to validate the correctness of a program's output. We design an attestation mechanism whereby a platform attests that the output of a program is the expected result of its execution which is not manipulated or tampered with. To motivate our work, we use the information auditing as an example. One of the main reasons why auditing is expensive in terms of manpower cost is that it requires step-by-step manual checking the entire data process including intermediate steps and results, the inputs and the final outputs. Our scheme allows a platform to attest its data processing to an auditor, which not only significantly speeds up the process, but also reduces the errors due to human negligence. Another application is the distributed computing projects, such as the well-known SETI@Home project [1] and the Great Internet Mersenne Prime Project [5]. These projects usually involve a job supervisor and many participants, usually volunteers on the Internet. The supervisor splits a computation intensive task into sub-tasks and assigns them to the participants. It is desirable for the supervisor to have a mechanism which is able to verify the correctness of the results generated by the participants so that they can be safely used for its scientific purpose.

We emphasize that our objective is not to authenticate the integrity of the output of a program, which can be trivially obtained by using digital signatures. Neither does our scheme attest the security of the platform. We observe that the assurance on a platform's security does not always lead to the assurance of a program's correct execution. Even in a secure platform with a reasonable security policy enforce-

*Liang Gu is a Ph.D student at the Peking University and is currently on attachment to the School of Information Systems, Singapore Management University.

STC'08, October 31, 2008, Fairfax, Virginia, USA.

ment, a malicious administrator is still able to tamper with or manipulate a program execution on the platform under her control.

In this paper, we propose a TCG-based remote attestation for the correctness of the execution of a target program. The basic rationale is as follows. If a program is modeled as a deterministic Turing machine, its execution, or its output, is determined by the sum of two factors: its binary code and the input data. Our attestation scheme measures not only the binary code of the target program, but also all the inputs. Although a program is fed with the input when it is started, it also exchanges data with other processes during the course of the execution. Therefore, our scheme attests the validity of those data as well, which may be the output from another program. In short, with a recursive fashion, the proposed attestation measures all relevant data in the platform that affects the final outcome of a target program.

The organization of the paper is as follows. In the next section, we discuss related work. In Section 3, we briefly explain the background of program dependency, which helps to find all relevant data flow. A synopsis of the proposed attestation scheme is provided in Section 4. Then, we present its details in Section 5 and Section 6. We conclude the paper in Section 7.

2. RELATED WORK

In the literature, a number of attestation mechanisms based on TPM have been proposed. Terra [4] uses a Trusted Virtual Machine Monitor (TVMM) to transform a tamper resistant hardware platform into multiple virtual machines (VMs) that are isolated from each other. With the protection of the trusted hardware, TVMM offers both the open-box VM and the closed-box VM. The attestation in TVMM only measures the programs before their executions and is not able to check their behaviors after attestation. Sailer et al. [15] introduced an integrity measurement architecture (IMA) which employs a loading time integrity measurement mechanism to prove the integrity of a remote system. IMA was the first scheme to extend the TCG specified measurements which include BIOS, the OS loader and the operating system, to programs at the application layer. However, IMA attests the integrity of the entire system and is incapable of testifying to a program's execution. Haldar et al. [7] introduced a semantic attestation mechanism based on the Trusted Virtual Machine (TVM). The TVM based semantic attestation mechanism enables the remote attestation of high-level program properties. However, they only introduced the framework and did not clearly spell out the methods to effectively attest a remote program to guarantee its behaviors. Shi et al. proposed BIND [20] which is a fine-grained attestation mechanism. It provides evaluation interfaces to attest the security-concerned segments of code. Jaeger et al. [11] introduced the Policy-Reduced Integrity Measurement Architecture (PRIMA) based on the information flow integrity checking against the Mandatory Access Control (MAC) policies. However, as Hicks et al. [9] observed that these system level mandatory access controls only seek to enforce security at application granularity and can not monitor the data handled within an application. Therefore, it still does not resolve the problem we target at.

Software based attestation mechanisms do not rely on TPM. Mark et al. [19] proposed to use obfuscation routines to perform static and dynamic analysis for remote attestation of sensor nodes. SWATT [18] performs attestation on embedded devices using a software verification function to verify the memory of embedded devices. Seshadri et al. [17] employed a specially designed checksum function to evaluate the state integrity of the remote programs on an untrusted platform. SCUBA [16] employs attestation to detect compromised nodes without false negatives and repairs the compromised ones with secure code update. Most of the existing software-based attestation mechanisms focus on providing a trustworthy routine to guarantee the trustworthiness of the attestation process. It is a commonly agreed that software-based solutions are more vulnerable to attacks than hardware-based solutions.

3. PRELIMINARY OF PROGRAM DEPENDENCY

In this section, we briefly explain program dependency, which is the cornerstone of our attestation scheme. Informally, dependency describes the relevance between two objects such as instructions, procedures, processes in one platform or even across two platforms. In this paper, we are particularly interested in the dependences at two levels, the program language level and the operating system level. The former refers to the dependences among instructions within a program [3, 10, 12], while the latter refers to the dependences among software components within a system architecture [6]. Although it is the system level dependence that enables us to find those relevant objects which affect the execution of the target program, we need the language level dependences as the starting point to derive them.

The language level dependences of a program are usually represented by the Program Dependence Graph (PDG) [10]. A PDG is a directed graph denoted by $G = \{V, E\}$ where V and E are the node set and the edge set respectively. A PDG visualizes the dependences among instructions of a procedure or a segment of code. A node in G represents one instruction of the program. An edge starting from v_i and ending at v_j means that v_i is dependent on v_j. The dependences among instructions can be either *control dependence* or *data dependence*. *Control dependence* refers to the relationship among instructions that one instruction determines whether another can be executed or not, whereas *data dependence* refers to the relationship among instructions that one's data is used by another.

The System Dependence Graph (SDG) [10] represents the data dependences and the control dependences among procedures of a program. An SDG is constructed by joining all procedures' PDGs at the inter-procedural dependent nodes. These inter-procedural dependences include: 1) parameter-out/write: the data flows from the first party to the second one; 2) parameter-in/read: the data flows from the second party to the first; 3) non-parameter method/function call: the first party calls the execution of the second one, without data exchange. Data dependence in an SDG refers to a program's read or write access to data objects such as configuration files. Throughout this paper, we use $\alpha_w, \alpha_r, \alpha_c$ and β_w, β_r to denote three types of inter-procedural dependences and two types of data dependences, respectively. These dependences depict the information flows between a program and other objects involved in its execution.

4. SYNOPSIS

Our attestation scheme has two participating platforms, the challenger denoted by C and the attester denoted by H. H is equipped with a TPM module coupled with the secure kernel [2]. Among the executables running on H, P is the target program of the attestation. We consider P as a deterministic Turing machine. The objective of the proposed attestation scheme is to allow C to verify the trustworthiness of the outputs of P. The adversary in our model has full privileges in controlling the software system on H, except the secure kernel and TCG software. In other words, the adversary is able to modify and manipulate all the programs and data files on H for her own benefits.

The main idea of our scheme is as follows. Since P is a deterministic program, its state and outputs are determined jointly by its binary code and all inputs. The initial input to P is static data, which are generated before P is invoked. During its execution, P may receive inputs from other programs or data objects. Like the initial input, these runtime inputs also have impacts on P's output. Therefore, our attestation scheme measures not only the binary code of P and its initial input, but also, in a recursive fashion, checks both the correctness and the timing of runtime inputs from relevant sources. In other words, for every data relevant to P's execution, our scheme measures the data itself, its generator and the inputs which result in the data.

We build an attestation agent, denoted as AA, running in the kernel space of H. AA makes use of the OS kernel services such as process management and file system management, to monitor and measure the execution of P and other relevant programs. The functioning of AA relies on the secure kernel which provides a secure execution environment by memory curtaining and process isolation so as to protect the program executions from attacks by malicious processes. The agent AA operates in two consecutive phases: the preparation phase, and subsequently, the measurement phase. Initially, in the preparation phase, AA derives the programs and data files the execution of P depends on, by checking its binary code (or source code if available). It then registers all related system calls according to the discovered dependences. In the measurement phase, it monitors all systems calls. For those system calls which are registered, AA makes an attestation on the relevant states.

In specific, the attestation of P's execution consists of the following steps.

Step I: AA executes in its preparation phase before P is invoked. AA uses the algorithms described in Section 5 to register all relevant system calls into a table. This task is only executed once for P. The table saved in the secure storage protected by TPM. On P's follow-up invocations, AA checks the integrity of the table before loading it.

Step II: On starting P's execution, AA proceeds into its measurement phase. It monitors P's execution and the system calls and makes timely measurements. All measurement results are deposited into the secure storage protected by TPM. The details are shown in Section 6.

Step III: On the exit of P's execution, AA requests H's TPM to sign all measurements. When the challenger C requests an attestation on P's execution, the signatures and the measurements are reported to C as in a normal remote attestation.

Since the last step is a standard remote attestation process, the rest of the paper only focuses on the first two steps in order to avoid verbosity. The system architecture of our framework is depicted in Figure 1.

5. THE PREPARATION PHASE

In this section, we first define a set of rules of program dependence, which form the theoretic basis of our algorithms. Then, we show the details of the algorithm executed by AA during the preparation phase.

5.1 Program Dependence

In the following, we use T_p to denote the inter-procedural dependence type set $\{\alpha_w, \alpha_r, \alpha_c\}$ and use T_d to denote the data dependence type set $\{\beta_w, \beta_r\}$. We say $P_i \xrightarrow{t} P_j$, when a programs P_i has a t-type inter-procedural dependence on another program P_j, $t \in T_p$. Similarly, we say $P_i \xrightarrow{t} F_j$, when P_i has t-type data dependence on a data file F_j, $t \in T_d$. A program P_i is treated as a set of instructions. We use C_i to denote a subset of P_i. Similar to the dependences among programs, we use $C_i \xrightarrow{t} C_j$ and $C_j \xrightarrow{t} F_j$ to denote C_i's dependence on C_j and F_j respectively, where $t \in T_p \cup T_d$.

Recall that P's System Dependence Graph (SDG) described in Section 3 depicts all dependences among instructions, some of which invoke relevant processes or read data from files. In order to testify to the inputs from relevant processes, it is not sufficient to only check P's instructions. Therefore, we define a set of rules below to derive the dependence relations among executables from the dependences among instructions.

Clearly, if C_i of P_i has a dependence on C_j of P_j, P_i has the same dependence on P_j. This rule also applies to dependences on file objects. We summarize them in Rule 1 and Rule 2, respectively.

RULE 1. $\forall C_i \subset P_i, C_j \subset P_j, t_p \in T_P,\ C_i \xrightarrow{t_p} C_j \Longrightarrow P_i \xrightarrow{t_p} P_j$.

RULE 2. $\forall C_i \subset P_i, t_d \in T_d,\ C_i \xrightarrow{t_d} F_j \Longrightarrow P_i \xrightarrow{t_d} F_j$.

The above rules only capture the dependences from the direct links in an SDG. Another type of dependence is due to sharing a common resource. Program P_i is indirectly dependent on program P_j if P_j's execution affects the input to P_i. We use Rule 3 and Rule 4 to reflect this.

RULE 3. $\forall C_i \subset P_i, C_j \subset P_j, C_k \subset P_k, \forall t_p \in \{\alpha_r, \alpha_c\}$, $C_i \xrightarrow{\alpha_r} C_j \wedge C_j \xrightarrow{t_p} C_k \Longrightarrow P_i \xrightarrow{\alpha_r} P_k$.

RULE 4. $\forall C_i \subset P_i, F_j, C_k \subset P_k,\ C_i \xrightarrow{\beta_r} F_j \wedge C_k \xrightarrow{\beta_w} F_j \Longrightarrow P_i \xrightarrow{\alpha_r} P_k$.

Rule 3 above shows that program P_k affects P_i whenever it affects P_j's execution which P_i depends on. Rule 4 shows P_k affects P_i whenever it produces any data which P_i relies on. Similarly, one program's execution may impact another program's execution through the chain of invocation. We express this with Rule 5 and Rule 6.

RULE 5. $\forall C_i \subset P_i, C_j \subset P_j, C_k \subset P_k, t_p \in \{\alpha_r, \alpha_c\}, C_i \xrightarrow{\alpha_c} C_j \wedge C_j \xrightarrow{t_p} C_k \Longrightarrow P_i \xrightarrow{\alpha_c} P_j$.

RULE 6. $\forall C_i \subset P_i, C_j \subset P_j, F_k, C_i \xrightarrow{\alpha_c} C_j \wedge C_j \xrightarrow{\beta_r} F_k \Longrightarrow P_i \xrightarrow{\beta_r} F_k$.

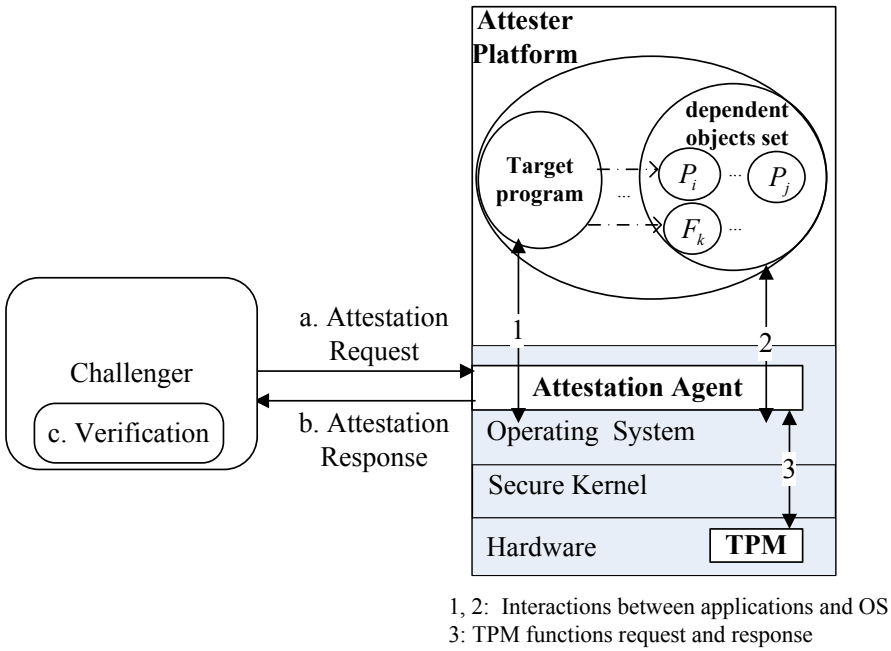

Figure 1: The remote attestation concept model

With Rule 3 and Rule 5, it is straightforward to derive the following transitiveness rule.

RULE 7 (TRANSITIVENESS OF α_r AND α_c). $\forall C_i \subset P_i, C_j \subset P_j, C_k \subset P_k, \forall t_1, t_2 \in \{\alpha_r, \alpha_c\},\ C_i \xrightarrow{t_1} C_j \wedge C_j \xrightarrow{t_2} C_k \Longrightarrow P_i \xrightarrow{t_1} P_j$.

This rule claims that both α_c-type and α_r-type dependences are transitive relations with respect to themselves. As shown later, this rule allows us to construct a dependence-closure set of objects the target program depends on.

5.2 Constructing the Set of Dependences and System Calls

To facilitate the discussion, we first explain the notations used in this section. We write $\langle P_i \xrightarrow{t_p} P_j, O\rangle$ or $\langle P_i \xrightarrow{t_d} F_k, O\rangle$, if O is the system calls set which establishes P_i's dependence on P_j or F_k. For example, suppose that P_i reads file F_k using the system call sys_read(). Then we have $\langle P \xrightarrow{t_d} F_k, \{\text{sys\_read}\}\rangle$. For easiness of presentation, we call $\langle P_i \xrightarrow{t_p} P_j, O\rangle$ a *dependence tuple*. We define $\mathcal{L}_P$ as the set of dependence tuples which affect P's execution. Obviously, $\mathcal{L}_P$ includes those objects P directly interacts with. According to the rules above, an object may indirectly affect P. Therefore, those objects are enclosed in $\mathcal{L}_P$ as well. Similarly, for a component C, its dependence tuple set is denoted by $\mathcal{L}_C$.

Given $\mathcal{L}_P$, AA is able to check all data exchanges and process executions affecting P. In the preparation phase, AA's mission is to compute $\mathcal{L}_P$, so that it is able perform measurement when P starts to run. The construction of $\mathcal{L}_P$ relies on the SDG described in Section 3, which requires code analysis on P. Therefore, we consider two possible scenarios: with the source code of P and only the binary code of P is available. Obviously, with the source code, the analysis of P is more efficient and precise. In the following, we define $\mathcal{D}_P^p$ and $\mathcal{D}_P^d$ as the executable object set and data object set that P's execution depends on, respectively. More formally, $\mathcal{D}_P^p = \{(\bar{P}, t) \mid t \in T_p, P \xrightarrow{t} \bar{P}\}$ and $\mathcal{D}_P^d = \{(\bar{F}, t) \mid t \in T_d, P \xrightarrow{t} \bar{F}\}$. Let $\mathcal{D}_P = \mathcal{D}_P^p \cup \mathcal{D}_P^d$.

Source code based approach.

Given the source code of P, the objects which P directly depends on can be easily obtained via a static program analysis. These objects are inserted to $\mathcal{D}_P$ and these direct dependences are inserted into $\mathcal{L}_P$. For every newly inserted object in $\mathcal{D}_P$, we then analyze its dependence and find out the objects it depends on. Among them, some objects which have relations with P according to the aforementioned rules are inserted into $\mathcal{D}_P$ and $\mathcal{L}_P$ as well. With this recursive approach, $\mathcal{D}_P$ and $\mathcal{L}_P$ are completed until no new object is inserted. We present an outline of the steps below.

- Step(1): Use the target program P's source code to generate the Abstract Syntax Tree (AST) of P with the approach proposed in [13]. Then use the resulting AST to construct the program dependence graph of all P's procedures [12, 8]. Connect these PDGs to construct the System Dependence Graph (SDG) of P, $\mathbb{G}_P = \{V, E\}$ using the method introduced in [10];

- Step(2): Compute the direct dependent objects set of P by executing Algorithm 1, i.e. $(\mathcal{D}_P, \mathcal{L}_P) \leftarrow$ Direct_Dependent_Object_Set$(r, \mathbb{G}_P)$, where r is the root of P's AST. Following Rule 1-7 introduced in Section 5.1, Algorithm 1 checks $\mathbb{G}_P$ to determine P's direct dependences on other executable and data objects.

- Step(3): For every executable objects P_i newly inserted into $\mathcal{D}_P$, generate $\mathbb{G}_{P_i}$ recursively. Following

Rule 3-7, add P's indirected dependent objects and dependences into $\mathcal{D}_P$ and $\mathcal{L}_P$ accordingly.

The detailed description of computing $\mathcal{L}_P$ is given in Algorithm 2, which calls Algorithm 1 to generate the object set a program directly depends on. In the Step (2), challenger first obtains the direct dependent objects set of the target program by Algorithm 1 whose complexity is $\mathbb{O}(N)$, where N is the number of nodes in the target program's SDG graph and is linear with target program's Line-of-code. In practical usages, if the direct dependent object set of a target program is generated and distributed by the program provider, the Algorithm 1 can only be executed for once before all remote attestation processes. Algorithm 2 employs a recursive procedure to get the indirected dependencies of the target program. The complexity of this procedure is determined by two factors: first is the number of programs which the target program and its dependent programs depend on; second is the Line-of-code of these dependent programs. It is the worst situation when the target program depends on all the programs in a system. However, in a multi-task operating system, when independent applications run simultaneously, a target program usually depends on only a part of the system. For example, a distributed client can runs on a personal computers with other independent applications simultaneously, like email clients and browsers, and these independent programs depend on specific parts of the whole system.

In practical usages, the complexity of the whole process for generating all the dependences of different programs can be reduced by following approach: first, for these programs which are commonly used as fundamental services for other programs, like the kernel modules in Linux, their $\mathcal{D}_P$ and $\mathcal{L}_P$ can be first generated and distributed by the program provider; second, in order to get the set of dependences for a specific application, the challenger may simply use these available $\mathcal{D}_P$ and $\mathcal{L}_P$ to generate the target application's set of dependences. For different programs, the average complexity of constructing the sets of dependences can be obviously reduced in this approach.

According to $\mathcal{L}_P$, a system trap table called SysTrap, is built as shown in Table 1. SysTrap contains four columns: *System calls, Caller, Callee*, and *Dependence type*. For every element in $\mathcal{L}_P$, we insert to SysTrap one entry accordingly, or multiple entries if several system calls are used.

Binary Code based Approach.

In remote attestation applications, it is not rare that only the binary code of the target program is available. In this case, reverse engineering is needed to get the binary code's higher level imperative representation [14], which is usually in an assembly language[1]. The assembly code can be treated almost in the same way as a high-level source code program. Algorithm 2 is equivalently applicable to the assembly language for generating the target program's dependence tuple set.

Generating $\mathcal{D}_P$ and $\mathcal{L}_P$ of P by the binary code analysis consists of four major steps:

1. Disassemble the binary code and get the assembly codes of the target program;
2. Construct the assembly code's program dependence graph and system dependence graph;
3. Identify all direct dependent objects set;
4. Identify all indirect dependent objects using Rule 3-7.

[1]For Java bytecode, it can be directly analyzed [25].

6. MEASUREMENT PHASE

The task of AA in its measurement phase is to take a snapshot of data generation by measuring the related objects. The structure of AA is shown in Figure 2. We place AA at those system call hooks, so that AA is able to intercept all system calls in SysTrap table. These hooks call the analyzing procedure in AA and transfer the system call type and parameter information to AA's analyzing procedure. According to the system call information, AA's analyzing procedure checks whether the system call, its caller and callee appear in SysTrap. If so, AA's measuring procedure is activated to perform the measurement. A measurement involves states of the caller and the callee, which can be either an executable object or a data file. Meanwhile, AA also records the time of invocation which is useful for later verification. After the measurement, AA forwards the system call to the OS kernel to resume the normal process and deposits the results into a sealed storage.

6.1 Linux System Call Trapping

We build AA with the Linux Security Module (LSM) [23]. LSM provides a set of hooks to enforce system access control policies for the kernel. For example, SElinux [21] has a typical Mandatory Access Control implementation of LSM, which supports dynamic security polices. In our scheme, all dependence related system calls are classified into the following categories: program execution, file operation, socket operation, inter-process communication, and kernel module operation. For the first four kinds of system calls, AA makes use of existing hook interfaces provided by LSM to trap a call. As LSM does not provide any hooks to monitor kernel modules operation, we implement two new hooks to monitor kernel modules operation.

Program Execution.

User space executables are invoked via the system call execve(). The bprm_alloc_security() hook is located in do_execve(), so the AA's analyzing procedure is called by bprm_alloc_security() to monitor program invocation. The loading of the dynamically linked binaries is transparent to the kernel. However the dynamic linker needs to map the dynamically loadable libraries into the virtual memory via mmap() system call. Therefore, we use file_mmap() to call the analyzing procedure to monitor dynamically loadable libraries. Script programs, such as shell or perl scripts, are executed by the corresponding interpreters. The execution of a script is a result of its interpreter's execution and the script loading is essentially a file loading process. For these program execution related system calls, AA's analyzing procedure needs to identify these with corresponding caller program and called program in **SysTrap**.

As a process is likely to be scheduled off from CPU due to time-sharing, it is desirable to re-measure the process when it is re-loaded to CPU. In that case, we use task_prctl() to call the analyzing procedure to check the states of process. If the process's state is changed, a new measurement will be performed.

Algorithm 1 Direct_Dependent_Object_Set($C, T, \mathbb{G}_P$)

Input: C is a component of executable P; T is abstract syntax tree of P; $\mathbb{G}(P)$ is P's SDG;
Output: C's direct dependent objects set $\mathcal{D}_P$=$\mathcal{D}_P^p \bigcup \mathcal{D}_P^d$, and $\mathcal{L}_P$;

if C is the root of T **then** $\mathcal{L}_P = \emptyset$; **endif**
set $\mathcal{D}_C = \{\mathcal{D}_C^p, \mathcal{D}_C^d\} = \{\emptyset, \emptyset\}$; $\mathcal{L}_\mathcal{C} = \emptyset$;
if C has subcomponents in T **then**
  for every subcomponent C_i of P **do**
    $\{\mathcal{D}_{C_i}^p, \mathcal{D}_{C_i}^d, \mathcal{L}_P\}$ =Direct_Dependent_Object_Set($C_i, T, \mathbb{G}_P$);
    For C_i's each program dependence $\langle C_i \xrightarrow{t_p} \overline{P}, O\rangle$ in $\mathcal{D}_{C_i}^p$, add $(\overline{P}, t_p)$ into $\mathcal{D}_C^p$, add $\langle C \xrightarrow{t_p} \overline{P}, O\rangle$ into $\mathcal{L}_C$; /** **Rule** 1 **/
    For C_i's each data file dependence $\langle C_i \xrightarrow{t_d} \overline{F}, O\rangle$ in $\mathcal{D}_{C_i}^d$, add $(\overline{F}, t_p)$ into $\mathcal{D}_C^d$, add $\langle C \xrightarrow{t_d} \overline{F}, O\rangle$ into $\mathcal{L}_C$; /** **Rule** 2 **/
  end for
else
  if C has only an instruction node **then**
    if C has a t_p dependence on program $\overline{P}$'s component $\overline{C}$ by system calls O **then**
      According to **Rule** 1, add $(\overline{P}, t_p)$ into $\mathcal{D}_C^p$, add $\langle C \xrightarrow{t_p} \overline{P}, O\rangle$ and $\langle C \xrightarrow{t_p} \overline{C}, O\rangle$ into $\mathcal{L}_C$, add $\langle P \xrightarrow{t_p} \overline{P}, O\rangle$ into $\mathcal{L}_P$;
    end if
    if C has a t_d dependence on data file $\overline{F}$ by system calls O **then**
      According to **Rule** 2, add $(\overline{F}, t_d)$ into $\mathcal{D}_C^d$, add $\langle C \xrightarrow{t_d} \overline{F}, O\rangle$ into $\mathcal{L}_C$, add $\langle P \xrightarrow{t_d} \overline{F}, O\rangle$ into $\mathcal{L}_P$;
    end if
  end if
end if
return $\{\mathcal{D}_C^p, \mathcal{D}_C^d, \mathcal{L}_P\}$;

Algorithm 2 Get_All_Dependent_Objects(P):

Input: Target program P's source code;
Output: P's dependence set $\mathcal{L}_P$;

Construct P's abstract syntax tree T_P and its dependence graph $\mathbb{G}_P$;
$\{\mathcal{D}_P^p, \mathcal{D}_P^d, \mathcal{L}_P\} = Direct\_Dependent\_Object\_Set(r, T_P, \mathbb{G}_P)$;
for every $(P_i, t_p) \in \mathcal{D}_P^p$ in which P_i has not been analyzed **do**
  Construct P_i's abstract syntax tree T_{P_i} and its dependence graph $\mathbb{G}_{P_i}$;
  $\{\mathcal{D}_{P_i}^p, \mathcal{D}_{P_i}^d, \mathcal{L}_{P_i}\} = Direct\_Dependent\_Object\_Set(r_{P_i}, T_{P_i}, \mathbb{G}_{P_i})$;
  for $\forall \bar{C} \subseteq P_i$ **do**
    if $\exists C \subseteq P, \exists\langle C \xrightarrow{t_p} \bar{C}, O\rangle \in \mathcal{L}_C$ **then**
      According to Rule 3, 5, 6, 7, check all dependences in $\mathcal{L}_{\bar{C}}$, find P's indirected dependent objects and add them into $\mathcal{D}_P^p$ and $\mathcal{D}_P^d$, and add these indirected dependences into $\mathcal{L}_C$; all P_i's direct dependences which contribute to identifying P's indirected dependences should be added into $\mathcal{L}_P$.
    end if
    if $\exists C \subseteq P, \exists\langle C \xrightarrow{\beta_r} F_k, O_1\rangle \in \mathcal{L}_C, \exists\langle \bar{C} \xrightarrow{\beta_w} F_k, O_2\rangle \in \mathcal{L}_{\bar{C}}$ **then**
      According to Rule 4, add (P_i, α_r) into $\mathcal{D}_P^p$, add $\langle P_i \xrightarrow{\beta_w} F_k, O_2\rangle$ into $\mathcal{L}_P$;
    end if
  end for
end for
return $\mathcal{L}_P$;

File Operation.

In order to monitor the runtime dependences among processes and files, file operations are monitored. When a process accesses a file, the LSM hook file_permission() is triggered to check the permissions of specific operation on the target file. We use this hook to call the analyzing procedure.

Inter-Process Communication (IPC).

To monitor inter-process communications, we use IPC related hooks to call the analyzing procedure. The IPC related LSM hooks are ipc_permission(), msg_queue_msgrcv() for system IPC message queues, shm_shmat() for shared memory segments, sem_semctl() for semaphore operations. For asynchronous IPC operations, which are carried out in different parts, the analyzing procedure separately monitors them. If the part of an asynchronous IPC operation matches SysTrap, the analyzing procedure calls the measuring procedure to record the states this IPC operation.

Socket Operation.

The socket objects can be considered as data objects. The reading from and writing to socket should also be monitored. We use two hooks: socket_sendmsg() and socket_recvmsg() to call the analyzing procedure.

System Calls	Caller	Callee	Dependence Type
sys_read	P_1	F_1	β_r
sys_execve	P_2	P_3	α_c
...	...	...	....

Table 1: System Trap Table

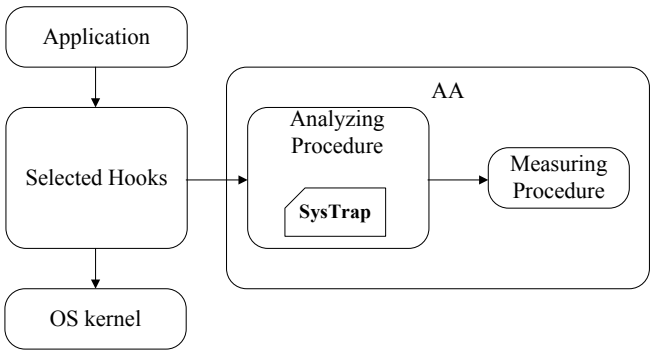

Figure 2: Attestation Agent

Kernel Module Operations.

Some applications involve modules of the operation system. When a kernel module is required, the kernel locates and loads the right module into the memory. As there is no LSM hook for kernel modules operations, we modify insmod() and modeprobe() to invoke the analyzing procedure to monitor the runtime dependences on kernel modules. The analyzing procedure deals with these operations in the same manner as with the program execution system calls.

6.2 Measurement Process

The measurement process is triggered when the trapped system calls match SysTrap table. AA may measure a process, a file or a segment of data. There are two types of data: external data which refers to external data input into the system; inter-process data which refers to the data communicated among objects in the system. The approach to record the state of external data is trivial. Thus, we elaborate the details of process and file measurement as well as the inter-process data among them. Table 2 shows the steps to measure a process Q before its once execution, where $\mathcal{SK}$ denotes the Secure Kernel, r is a random value received from the challenger, and AIK_{pr} is the private AIK of TPM on the target platform. We employ the same cache mechanism in [15] to measure a target process before it executes. Since a process can be offloaded from a CPU due to process scheduling, a cache is used to temporarily keep all the measurements when the process is schedule out. A new measurement is performed only if the target process or data file have been changed before being reloaded. Otherwise, the measuring procedure directly re-uses the measurement stored in the cache. On Q's execution, AA first suspends the process and measures the process. Then AA checks the cache, and records the cached measurement if cache-hit happens; otherwise, AA utilizes the platform configuration registers in TPM to store the results. After the measurement is stored, the Secure Kernel enables the trusted execution mode for Q. When Q's execution stops, the Secure Kernel exits from the trusted execution mode and the operating system regains the control. For data files, AA also employs the cache mechanism to measure them immediately when they are accessed. The steps are outlined in Table 3.

For different dependence types, AA measures different objects accordingly.

- α_c dependence: This type of dependence only involves two processes. Therefore, AA employs the process in Table 2 to measure the states of both the caller and the callee.
- α_r or α_w dependence: These two types of dependences involve data exchange between two processes. Thus, AA measures the states of both processes and the data exchanged between them. AA employs the process in Table 2 to measure the states of processes. For α_w, its exchanged data can be directly recorded. For α_r, the exchanged data is only available when the system call returns. Therefore, for α_r related system calls, we insert a hook at the end of the system call handler to catch the returned data.
- β_r or β_w dependence: These two types of dependences involve file operations. Accordingly, AA measures the state of the process, the state of the file, and the data read from or written to the file, using the algorithms in Table 2 and 3. The data written to the file can be directly recorded. For the data read from the file, we have to handle β_r related system calls to get the returned data immediately after they finish the task and before they return the data to application. If the β_r or β_w dependences are between a process and the socket objects, AA measures the state of the process, the data read from or written to the socket object. The state of the socket object is represented by the data.

Table 2: Process measurement procedure: MEASURE(Q, r)

Steps	Players	Messages and Actions
1	AA	Suspends Q , checks the measurement cache, if cache-hit happens, records the cached measurement and jumps to **step 6**
2	AA $\rightarrow \mathcal{SK}$	Disables interrupt
3	$\mathcal{SK} \rightarrow TPM$	PCR_Reset
4	TPM	PCR_Extend (Address(Q), Q's code)
5	$TPM \rightarrow$ AA	$PCR, Quote = Sig\{PCR, r\}_{AIK_{pr}}$;
6	$\mathcal{SK}$	$\mathcal{SK}$ enables the Trusted Execution Mode, enables interrupt
7	AA $\rightarrow Q$	Q obtains control and resumes execution
8	$\mathcal{SK}$	After Q stops its execution, $\mathcal{SK}$ disables the Trusted Execution Mode
9	OS	OS re-obtains control

Table 3: File measurement procedure : MEASURE(F, r)

Steps	Players	Messages and Actions
1	AA	Checks the measurement cache, if cache-hit happens, records the cached measurement and jumps to **step 6**
2	AA $\rightarrow \mathcal{SK}$	Disables interrupt
3	$\mathcal{SK} \rightarrow TPM$	PCR_Reset
4	TPM	PCR_Extend (Address(F), F's data)
5	$TPM \rightarrow$ AA	$PCR, Quote = Sig\{PCR, r\}_{AIK_{pr}}$;
6	$\mathcal{SK}$	Enables interrupt
9	OS	OS re-obtains control

Our measurement minimizes the gap between time of measure and time of use. The secure kernel provides isolated memory spaces to quarantine the process at runtime when running at the Trusted Execution Mode. It is able to prevent the target process's space from being accessed by other programs when it is being executed. If the target program's execution requires a long duration, the OS may divide the task into several smaller slots. In order to protect the measured program from a malicious OS's modification when it is suspended, we employ the securely restoring execution environment mechanism introduced in [20] to deal with this situation. Therefore, the invoking time measurement guarantees that the measurement indeed reflects the state of the one being executed.

7. CONCLUSION

Before we conclude this paper, we discuss several interesting features of the proposed scheme.

Fine Granularity Fine-granularity attestation is known for its flexibility in software update and light-weight known-good fingerprints management. Our scheme can be adapted to provide fine-granularity, since we use the code to obtain the target program's dependent objects. To support fine-granularity, the programmer or the challenger annotates the sensitive parts of target program. AA only checks the dependences relevant to the annotated part.

Dependent Objects Set Distribution A compiler usually builds AST and the program dependence graph for the optimization purpose. The program developer may generate the software's dependent object set during its compilation. Once being generated, the dependent object set can be distributed by the software vendor together with the released software.

Information Flow Checking As our attestation agent dynamically records the processes and the objects related to the target program, it allows the challenger to verify the interactions between the target program and the objects it depends on. To support information flow checking, we modify the measurement phase to include the security labels of the processes as well.

In short, we propose in this paper a new method for attesting the execution of programs. We design an attestation agent which resides in the attester's platform. The agent analyzes the target program as well as its runtime environment to find out relevant processes and objects which affect the target program's execution. When the target program starts, the agent measures the state of the processes and data by trapping the corresponding system calls, with the assumption that the Secure Kernel and TPM jointly provide a secure computing environment. Our scheme also allows for fine-granularity attestation and information flow checking.

8. ACKNOWLEDGMENTS

This work is partly supported by the Office of Research, Singapore Management University and partly supported by the High-Tech Research and Development Program of China under Grant No. 2007AA010301. We especially thank Prof. Weizhong Shao for his valuable advice in the research. The authors would like to thank the anonymous reviewers for their very valuable comments and their helpful suggestions.

9. REFERENCES

[1] The search for extraterrestrial intelligence project. http://setiathome.berkeley.edu.

[2] AMD platform for trustworthy computing. http://www.microsoft.com/whdc/winhec/papers03.mspx, 2003.

[3] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 342–351, New York, NY, USA, 1992. ACM Press.

[4] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra a virtual machine-based platform for trusted computing. In *SOSP 2003*, Bolton Landing, New York, USA, October, 2003.

[5] GIMPS. The great internet mersenne prime search. http://www.mersenne.org/prime.htm, 2007.

[6] Z. Gu, S. Kodase, S. Wang, and K. G. Shin. A model-based approach to system-level dependency and real-time analysis of embedded software. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, page 78. IEEE Computer Society, 2003.

[7] V. Haldar, D. Chandra, and M. Franz. Semantic remote attestation—a virtual machine directed approach to trusted computing. In *the Third virtual Machine Research and Technology Symposium (VM '04). USENIX.*, 2004.

[8] M. J. Harrold, B. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. *SIGSOFT Software Engineering Notes*, 18(3):160–170, July 1993.

[9] B. Hicks, S. Rueda, T. Jaeger, and P. McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of 2007 USENIX Annual Technical Conference*, page 205ÍC218, 2007.

[10] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 392–411, New York, NY, USA, 1992. ACM Press.

[11] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *SACMAT '06 : Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28, New York, NY, USA, 2006. ACM Press.

[12] R. Johnson and K. Pingali. Dependence-based program analysis. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 78–89, New York, NY, USA, 1993. ACM Press.

[13] J. Jones. Abstract syntax tree implementation idioms. http://jerry.cs.uiuc.edu/ plop/plop2003/Papers/Jones-ImplementingASTs.pdf.

[14] Á. Kiss, J. Jász, G. Lehotai, and T. Gyimóthy. Interprocedural static slicing of binary executables. In *SCAM*, page 118. IEEE Computer Society, 2003.

[15] R. Sailer, X. Zhang, T. Jaeger, and L. v. Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, USA, August, 2004.

[16] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. In *WiSe '06: Proceedings of the 5th ACM workshop on Wireless security*, pages 85–94, New York, NY, USA, 2006. ACM Press.

[17] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. v. Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP'05*, Brighton, United Kingdom, October 23ÍC26, 2005.

[18] A. Seshadri, A. Perrig, L. v. Doorn, and P. Khosla. Swatt: software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*. IEEE, 2004.

[19] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In R. Molva, G. Tsudik, and D. Westhoff, editors, *ESAS*, volume 3813 of *Lecture Notes in Computer Science*, pages 27–41. Springer, 2005.

[20] E. Shi, A. Perrig, and L. V. Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *2005 IEEE Symposium on Security and Privacy*, 2005.

[21] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux security module. Report #01-043, NAI Labs, Dec. 2001. Revised May 2002.

[22] Trusted Computing Group. Trusted platform module main specification. http://www.trustedcomputinggroup.org, October 2003.

[23] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, Aug. 2002.

[24] L. Xiao-Yong, S. Chang-Xiang, and Z. Xiao-Dong. An efficient attestation for trustworthiness of computing platform. In *Proceedings of the 2006 International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP'06)*, 2006.

[25] J. Zhao. Dependence analysis of java bytecode. In *COMPSAC*, pages 486–491. IEEE Computer Society, 2000.