Remote Attestation on Function Execution (Work-in-Progress)

Liang Gu [1], Yueqiang Cheng[2] ,Xuhua Ding[2], Robert H. Deng[2], Yao Guo[1], Weizhong Shao[1]

[1]Key Laboratory of High Confidence Software Technologies,
Peking University, Beijing, China
[2]School of Information Systems,
Singapore Management University, Singapore

Abstract. A program is a compound of various subroutines playing different roles. In this paper, we study how to attest the execution of those mission-critical subroutines whose execution is the basis to establish trust. Our results include a new attestation scheme called *function attestation*. Given a function F of a program $\mathcal{P}$, the proposed scheme allows for an efficient and secure attestation by using the debug facility of processors and building a trust chain rooted at TPM. Our scheme is lightweight and easy to deploy. It can also be easily extended to support multiple-threaded programs and data flow attestation with slightly more overhead.

Key words: Trusted computing, remote attestation, mission-critical function

1 Introduction

The concept of remote attestation was originally proposed by Trusted Computing Group (TCG) as a key functionality of the trusted computing architecture using the Trusted Platform Module (TPM) [15]. In TCG's trusted computing specification, remote attestation allows a platform, usually referred to as a *challenger*, to verify the configuration integrity of another platform (i.e. an *attester*). Recent years have witnessed various evolutions out of the basic TCG attestation in many dimensions, e.g. program semantics attestation [4], behavior attestation [17], security policy enforcement [7], and property attestation [1].

In this paper, we study remote attestation on mission-critical function execution[1]. The objective is to verify *whether a subroutine of a program is properly executed.* Such an attestation mechanism has many interesting applications. A content distributor can verify whether the DRM engine within a media player functions as expected. An online auction user can check whether the server handles his bid complying the auction rules. Other applications include account auditing and e-voting system etc. A common feature of these scenarios is that the challenger is concerned with whether some given data are processed as expected by a subroutine in a program. We highlight the difference

[1] The term "function" here refers to the program function or procedure. Throughout the paper, we use function and subroutine interchangeably.

between function attestation and program attestation [3]. The latter checks the entire program execution, i.e. all data and control flows. Inevitably, the program attestation scheme is cumbersome due to the enormous number of objects to measure. Nonetheless, in many applications, a challenger is only concerned with a particular procedure, instead of those routines with less security significance, e.g. error report and GUI. Since the challenger can predict (part of) the input data and only a fraction of the program is checked, function attestation is more flexible and allows an efficient construction.

We propose a function attestation scheme in this paper. The scheme installs an attestation agent running at the attester's platform. Given a program's binary code, the agent first locates the entry address of the function to attest. The invocation of the function triggers the agent to set up a secure environment for the function execution and to measure its initial state including the code and variables. During the function execution, the agent measures all data exchanges. In the end, a set of TPM signed measurements are returned to the challenger as the attestation response. The challenger not only verifies the integrity of the state, but also verifies the presence of the secure environment. Our scheme is lightweight and does not rely on source code and complier support. Therefore, it is readily deployable for most off-the-shelf software packages.

In the rest of the paper, we describe the related work in Section 2. Section 3 provides a brief introduction on the building blocks used by our scheme. An overview of the scheme is presented in Section 4 with the details being elaborated in Section 5. We discuss several extensions in Section 6 and conclude the paper in Section 7.

2 Related Work

TCG attestation was first introduced to attest the integrity of platform configurations [15]. Terra [2] attests the code integrity by using a Trusted Virtual Machine Monitor (TVMM), which provides both the open-box Virtual Machine (VM) and the closed-box VM . Open-box vm supports normal OS and applications, while closed-box VM provides a restricted environment to run specified applications. IMA [13] implements the TCG attestation in Linux as an integrity measurement architecture and it measures all software components on platform at their loading time. IMA was later employed to support access control based on platform configuration integrity [12]. The integrity-based attestation schemes can only attest the integrity of target objects, and the loading time integrity does not guarantee stronger security assurance.

Semantic attestation was introduced based on the Trusted Virtual Machine (TVM) [4] and it takes the advantages of TVM to attest the high-level program properties. However, they only introduced the framework and did not clearly specify the methods to effectively attest a remote program according to specified high-level program properties. Property-based attestation[1, 10, 11] was also introduced as a framework to attest the properties of remote platform without revealing the platform configurations. These existing schemes of Property-based attestation did not specify how to attest a specific property. BIND [14] was later introduced as a fine-grained attestation scheme to attest the security-concerned segments of code.The Policy-Reduced Integrity Measurement Architecture (PRIMA) [8] attests the information flow integrity against the Mandatory Access Control (MAC) policies on remote platform. Program execution attestation in-

troduced in [3] is the closest work to ours. It attests whether a program is executed as expected. The approach in [3] is to analyze all objects in the systems (e.g. related processes, files, network sockets) whose states impact the target program's execution. Then, all those objects are measured dynamically by intercepting the involved system calls. Therefore, it incurs a heavy load of measurements. Note that the Secure Kernel is an indispensable prerequisite for both BIND and program attestation, since both of them rely on it to provide a secure environment.

Flicker [9] was later introduced as an infrastructure for executing security sensitive code in complete isolation. It leverages the Secure Virtual Machine (SVM) of AMD processors and provides fine-grained attestation on program execution. However, Flicker is source code based and requires modifications on source code. In many cases, the source code is not available and it is not possible to carry out the modification on target program. Our solution provides binary code based fine grained attestation on mission-critical function's execution. It also leverages the security feature and the debug feature of commodity processors. Our solution does not require source code and modification on target program.

3 Preliminaries

3.1 Call Stack

During a program execution, the operating system maintains a *call stack* (or shortened as *stack*) in its memory to manage function calls. A call stack stores information about the active functions/procedures of a program. The stack organization is machine dependent. Figure 1 depicts the layout of process memory (left part of the figure) and the call stack (right part) on x86 architecture. Growing downwards, the call stack shows the immediate state information after function *a* has invoked function *b*. A call stack consists of an array of *stack frame*, each of which represents an active function and consists of its local variables, the previous frame pointer, the return address and the function parameters. When function a calls function b, the input parameters for b are first pushed into the stack in the reverse order, followed by the return address and then b's local variables. The return address is the address of the instruction immediately following the function call. Therefore, when function b exits, the return address is popped up and the control returns back to function a. The operations of the stack involves two registers, *stack pointer* and *frame pointer*. The former points to the current top of the stack, whereas the latter points to a fixed point of the frame. In short, the call stack describes the run-time state of a program. By tracing the evolution of stacks, one can monitor the control flow.

3.2 Secure Kernel

Secure Kernel (SK) was introduced as a special security mode of processors. AMD's Secure Execution Mode is one of the examples. SK is a hybrid hardware and software components running at the core privileged ring of a system, lower than the OS kernel. It provides as few as possible software interfaces to upper layers in order to minimize its

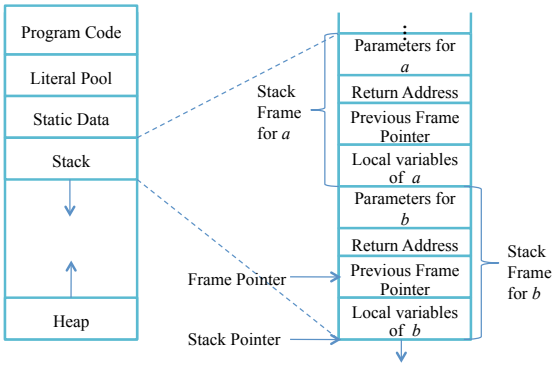

Fig. 1. Layout of process memory and call stack

size. By leveraging the hardware protection mechanisms, it not only secures its own execution, but also protects application software with a secure environment. A full-fledged Secure Kernel prevents illegal access to a process's memory space. It also protects the process's I/O channels, such as paths to network sockets, display and keyboard.

3.3 Debug Trap

The Intel IA-64 processor is equipped with debug facility [5, 6] to allow breakpoints in program execution. It has four debug registers (DR0, DR1, DR2 and DR3) to monitor accesses to physical or virtual address. When any address stored in a debug register is accessed, a debug exception is generated and the program control is transferred to a trap handler. We use this trap facility of Intel IA-64 processor to monitor the execution of the mission-critical function: at its entry address and its exit address. Note that similar facilities are also available at other families of processors, e.g. Intel Itanium.

4 Scheme Overview

4.1 Architecture

The proposed attestation scheme involves two entities: an attester executing a program $\mathcal{P}$ and a challenger who requests an attestation on a subroutine F of $\mathcal{P}$. The attester runs an *attestation agent* (denoted by $\mathcal{AA}$) to process all attestation related tasks. Running at the highly-privileged kernel level protected by SK, $\mathcal{AA}$ is the core of our scheme. To ensure the integrity of $\mathcal{AA}$'s execution, our scheme relies on Secure Kernel to provide a secure execution environment. The scheme proceeds in three phases: 1) Preprocessing Phase wherein the entry address of F in $\mathcal{P}$'s address space is located; 2) Runtime Phase wherein the execution of F is monitored and the relevant state information is measured accordingly; 3) Verification Phase wherein a challenger verifies the attestation measurements. The flow chart of our scheme is depicted in Figure 4.1 and the software layout of our scheme is shown in Figure 4.1. $\mathcal{AA}$ consists of two components: a trap handler

and a kernel service. When F is invoked, the trap handler is activated and measures the initial state of F. The kernel service of $\mathcal{AA}$ is to dynamically record the data input and output of F during its execution.

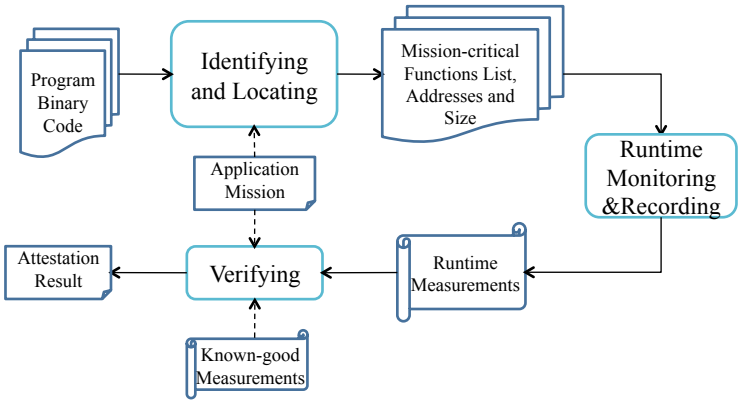

Fig. 2. An overview of the attestation scheme

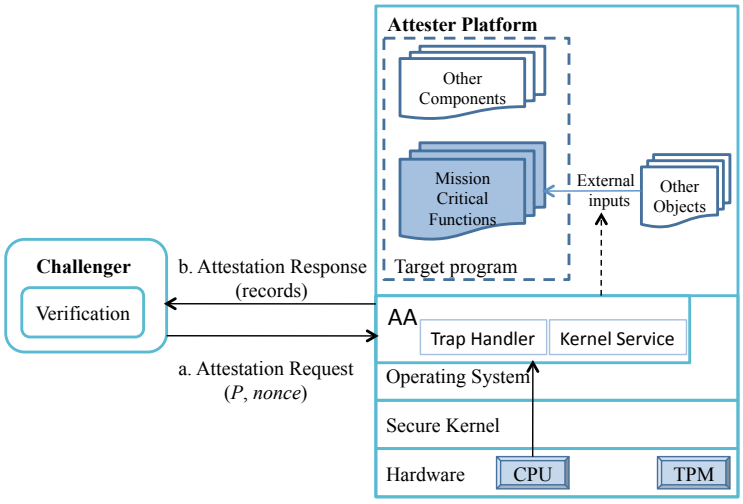

Fig. 3. Software layout of the attestation scheme

CAVEAT. In our scheme, F is chosen by the challenger. How and why a subroutine of $\mathcal{P}$ is selected is orthogonal to our study. Typically, the challenger is concerned with mission-critical functions whose proper execution is of great consequences, e.g.

a password encryption routine. When elaborating our scheme, we only focus on single function attestation for ease of presentation. The scheme can easily be extended to support multiple functions. A detailed discussion is presented in Section 6.

4.2 Trust Chain

We assume that the adversary is not able to compromise the secure environment established by Secure Kernel. In order for the challenger to verify an attestation of F, a trust chain needs to be built up to the attestation agent. The root of trust chain in our attestation scheme is TPM. With the TPM-based authenticated boot-up, the trust chain can be extended to the Secure Kernel if the challenger successfully verifies its integrity at loading time. The authenticated boot-up also attests the loading time integrity of OS kernel and $\mathcal{AA}$. Since $\mathcal{AA}$ runs under the protection of Secure Kernel, the trust chain can be further extended to $\mathcal{AA}$. In our scheme, Secure Kernel protects the memory region of target programs from tampering at runtime.

4.3 Rationale

Before presenting the details of our scheme in the next section, we explain the rationale of our design. We model the function F as a deterministic Turing machine[2]. Its state is determined jointly by its binary code and all data inputs. Therefore, to attest F's execution, it is sufficient to attest which data are used as inputs and what instructions are executed for F. The inputs to F include both the initial inputs which are data generated before F's invocation, as well as runtime inputs which are data exchanged between F and other processes or devices. In our scheme, the initial inputs are measured by the trap handler when F is invoked, while the runtime inputs are measured in real time by the kernel service.

It is more costly to measure the instructions run by F, due to the well-known gap between the time-of-measurement and the time-of-running. We take the same approach as BIND [14], i.e. to employ Secure Kernel for execution protection. In the Secure Kernel protection mode, the execution of F will not be tampered with by the adversary. To measure the code precisely at the loading time, we employ the debug facility provided in x86 processors. Both the entry and exit virtual addresses of F are identified and loaded into two debug registers. A debug trap is raised whenever the instruction at the entry/exit address is loaded, which allows the trap handler to enable/disable the secure execution environment and to measure the binary code together with its initial inputs.

5 Design Details

5.1 Preprocessing Phase

The first main task in the preprocessing phase is to locate the entry of F in binary code so as to prepare for trapping and monitoring [3]. A function can be called with either a

[2] A randomized function F can be treated as a deterministic function with randomized inputs.

[3] In order to locate the mission-critical function, the binary code is required to be compiled with symbol tables.

static linking or a dynamic linking. Depending on the link type, the attester locates the entry address of F in different ways.

STATIC LINKED FUNCTION The entry address of a statically linked function is hard-coded during compilation time and remains unchanged in every loading. The address can be retrieved by looking up the Symbol Table in an ELF file. The symbol of *STT_FUNC* type is related to functions. The attributes of *Elf32_sym* : *st_name*, *st_value* and *st_size* represent the function ID, function address and function size respectively. For example, the function *openssh_RSA_verify()* in ssh-rsa.c of the OpenSSH program is a static function, whose function ID is *sub_351C0* with an offset at 0x000351C0 and has 0x000006C2 bytes. $\mathcal{AA}$ loads 0x000351C0 to a debug register as the entry trap for F.

DYNAMICALLY LINKED FUNCTION If F is a dynamically linked function, its entry address is chosen by the dynamic linker at loading time. ELF employs the Procedure Linkage Table (PLT) to support lazy binding. Lazy binding means that the address relocating only happens when the function is accessed for the first time. The size of F can be extracted from the shared library object containing F. For example, function *_RSA_public_decrypt()* is contained in lib "libcrypto.so.0.9.8" and its size is 0x00000030 bytes. The actual entry address F can be obtained by setting a trap upon the write operation to the corresponding address in the Global Offset Table (GOT). First, $\mathcal{AA}$ looks up the PLT to locate the PLT entry for function F. From the first *jump* instruction in the PLT entry, $\mathcal{AA}$ finds the corresponding GOT entry, where the starting address of F will be stored by the dynamic linker during the first loading. $\mathcal{AA}$ sets a trap on the GOT entry by loading the GOT entry address to a debug register. $\mathcal{AA}$'s trap routing will be activated when the dynamic linker changes the initial value at the GOT entry to the actual entry of F. Therefore, $\mathcal{AA}$ obtains the entry point of F and sets up the trap for F.

In order to support fine-grained attestation, another task in the preprocessing phase is to identify the objects F's execution depends on. There are three types of object identified by $\mathcal{AA}$.

1. Binary image of F. It includes not only F's binary code, but also the instructions of those subroutines called by F. Their addresses and sizes are recorded by $\mathcal{AA}$ accordingly.
2. Global variables accessed by F. $\mathcal{AA}$ inspects the binary image of F and records the address of those global variables accessed by. All global variables can be located in the *.symtab* section in $\mathcal{P}$'s binary code.
3. System calls for data exchange. $\mathcal{AA}$ examines the binary image of F and records the enclosed data exchange systems calls. Specifically, these system calls are for data object accesses (disk files and sockets) and inter-process communications. $\mathcal{AA}$ locates all the corresponding hooks in Linux Security Module (LSM), including such as *file_ioctl*, *socket_recvmsg*, *ipc_permission*, *msg_queue_msgctl*, and *shm_shmctl*. The enclosed data exchange system calls for F are divided into categories according to these LSM hooks and stored as a config file in format like XML. At runtime, $\mathcal{AA}$ uses this config file for F as an input to monitor and measure F's execution.

5.2 Runtime Phase

The runtime phase begins with the exception triggered by accessing the entry address of F. The main task of this phase is to measure F's execution. The measurement results will then be used for attestation and verified by the challenger. Two types of information are measured. One is the information related to the execution state, such as stack and code. The second type of information is those runtime input/output data [4] to/from F through system calls. Accordingly, the trap handler component of $\mathcal{AA}$ monitors and measures the execution state information, whereas the kernel service of $\mathcal{AA}$ records data exchanges. An illustration of the runtime phase is shown in Figure 4 below. It has three stages: i) activation, ii) execution and iii) exit. The measurement actions in the procedure are summarized in Table 1.

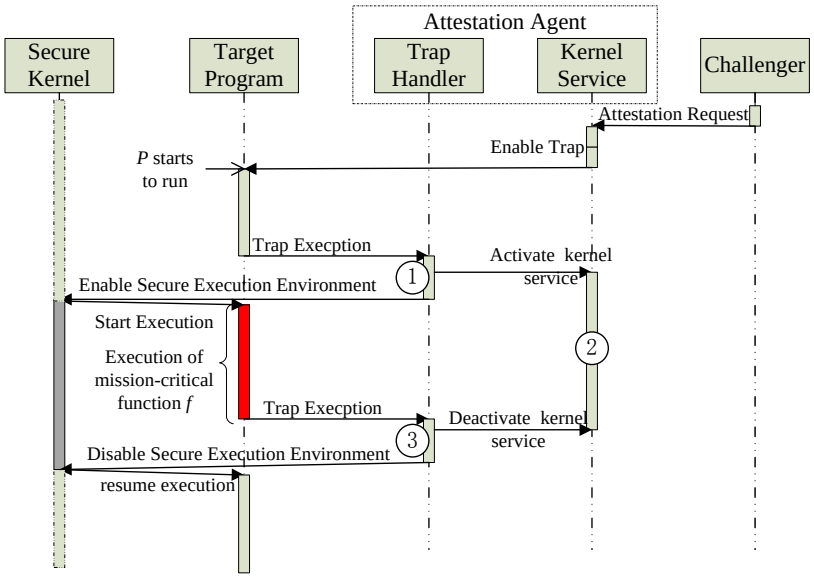

Fig. 4. Runtime phase: measuring execution state and data exchange of F

Activation A debug exception is generated whenever the entry address of F is accessed. The control is then passed to $\mathcal{AA}$'s trap handler. Before the trap handler is launched, a Platform Configuration Register PCR_i within TPM, which is used to extend integrity measurements, will first be reset:$PCR_i\_reset()$. Then TPM extend the

[4] The inputs and outputs of a function may be data structures of arbitrary complexity and nesting levels. We only consider the data of value type in our scheme of this version, like the integer type.

integrity of the trap handler as an extension of authenticated booting. Then, the trap handler performs the following steps.

Step 1:Initial configuration

The trap handler disables interrupts and DMA to protect its execution. It extracts the return address of F from the stack and sets a debug register with this return address, so that $\mathcal{AA}$ will re-gain the control when F exits. It also activates the kernel service of $\mathcal{AA}$ for monitoring runtime external inputs of F.

Step 2: Measurement (Recording action ① in Figure 4 and Table 1).

The trap handler measures both the execution environment and F's initial state. These measurements are extended with the PCR_Extend.

The execution environment includes the code of Secure Kernel, the code of OS kernel modules containing the kernel service of $\mathcal{AA}$. The initial state of F includes the binary image of F including subroutines called by F, the global variables F accesses, and the local call stack. Note that the addresses of the first two are obtained during the preprocessing phase and the call stack can be located by the stack pointer and the frame pointer. The call stack contains the input arguments of F.

Step 3: Set up the secure environment for F

The trap handler enables the interrupts and then it calls the Secure Kernel to set up a secure environment for F. Secure Kernel establishes a protected environment for $\mathcal{P}$ and transfers $\mathcal{P}$ into the secure domain, including $\mathcal{P}$'s program code, stack, heap and other contents. Then Secure Kernel passes the control to F.

Execution (Recording action ② in Figure 4 and Table 1). The kernel service is implemented in the operating system kernel. The kernel service leverages a set of hooks provided by Linux Security Module (LSM) [16], which is originally used to enforce system access control policies for the kernel. Whenever F invokes a system call, the kernel service is invoked and checks whether the caller or callee is $\mathcal{P}$. If so, the kernel service measures the data in use and related information, such as file name or addresses. These measurements are also extended into the specified PCR with PCR_i_Extend.

Exit (Recording action ③ in Figure 4 and Table 1). When F's execution completes and the processor accesses its return address, a breakpoint trap is generated again. The trap handler disables interrupts, then records F's return results and binds it with the inputs and binary image measurements. In x86 architecture, the return value is passed in register, namely EAX, ECX and EDX. Then the trap handler deactivates the kernel service and calls Secure Kernel to transfer $\mathcal{P}$'s context into the original domain. Secure Kernel terminates the secure domain for $\mathcal{P}$ and resumes the execution of $\mathcal{P}$ in a normal environment.

5.3 Verification Phase

As in all attestation scheme, the challenger communicates with the attester through an authenticated and confidential channel. The challenger receives a set of measurements

Recording Action ID	Executor	Action
①	trap handler	$PCR_i\_Extend(SecureKernel)$;$PCR_i\_Extend$ (OS modules containing the Kernel Service);PCR_i_Extend (f, f's parameters,local variables, global variables,f's dependent functions);
②	Kernel Service	PCR_i_Extend ($\mathcal{P}$'s External Inputs, External Inputs's producer);
③	trap handler	o=f's outputs;PCR_i_Extend (o); Records o and $Sig\{PCR, o\|\|nonce\}_{AIK_{pr}}$

Table 1. Recording actions for monitoring a mission-critical function f's execution(The action IDs are shown in Figure 4)

signed by the attestation key (AIK) of the attester's TPM. Prior to the signature verification, the challenger checks the validity of AIK as specified in the TCG standard.

In order to verify execution of F, the challenger prepares a database of the known-good measurements for $\mathcal{AA}$ Secure Kernel, and F's binary image. For those data accessed by F, e.g. initial parameters, the challenger determines their legitimate domains. We remark that the challenger may have the expected values of certain inputs. For example, F may take as input the data sent by the challenger. The verification procedure consists of three steps. In essence, the challenger ensures that the trust chain rooted at TPM can be properly established.

- **Step 1** The challenger verifies all the signatures on the measurements by using the corresponding public attestation key. If any signature is verified false, the attestation is rejected and the challenger aborts. Otherwise, the challenger is ensured about the cryptographic integrity of the measurements.
- **Step 2** The challenger checks the measurement for the attester's authenticated boot-up, including the loading of Secure Kernel, and the measurement of $\mathcal{AA}$'s binary code including both the trap handler and the kernel service component. If any of the measurements does not match the known-good database, the challenger rejects the attestation and aborts. Otherwise, it is ensured about the trustworthiness of the execution environment including both the presence of Secure Kernel and the integrity of $\mathcal{AA}$.
- **Step 3** The challenger first compares the measurement of the binary image of F with the known-good ones. Then, the challenger examines the measured call stack and checks the initial state of F, including F's input parameters and the global variables, as well as F's outputs. The challenger rejects the attestation if any of the data is not expected. In addition, the challenger verifies TPM's signature on the binding between F's output and the execution.

6 Discussions

6.1 Characteristics of Mission Critical Functions

The I/O characteristics of mission critical functions vary from application to application. Typically, a subroutine of a program may invoke file operations or interprocess operations. Therefore, Secure Kernel has to establish a sophisticated security domain to protect the I/O operations. In applications where the mission critical function does not use system calls, a lightweight secure environment is sufficient to protect the execution. For example, cryptographic operations are usually computation intensive and do not require data I/O operations. Hence, a memory curtaining mechanism is strong enough to ensure the secure execution. It can be established by employing the security features of modern processors, e.g. the SKINIT instruction in AMD SVM. Furthermore, the kernel service of $\mathcal{AA}$ is not activated. Thus, the scheme becomes much more lightweight and the trust chain has less links.

6.2 Attest Multiple Functions

A challenger may be concerned with several subroutines of a program and thus demands an attestation on all of them. It is trivial for our scheme to attest less than four functions, as the processor has four debug registers allowing four breakpoints to be set. At the preprocess phase, $\mathcal{AA}$ can load all entry addresses to the debug registers.

By employing the software based breakpoint exception (Interrupt Vector 3) [5], our scheme can be easily extended to support more than four attestation targets. The INT 3 instruction is one byte long and is easy to replace an opcode in a code segment. Specifically, the trap handler saves the opcodes at the entry and exit addresses of the mission-critical functions, and replaces them with an INT 3 instruction. When the processor executes these INT 3 instructions, the trap handler gains the control and restores the original opcode to the program. Upon returning from the trap handler, $\mathcal{P}$'s execution resumes with the original instructions. Though flexible, this software based approach obviously increases the size of TCB and is slower than the hardware based approach.

6.3 Context Switch

There are two types of context switches, process switch and thread switch. With Secure Kernel's protection, a process's integrity can be preserved even though it is switched on and off. It is infeasible for a malicious process to illegally access $\mathcal{P}$'s address space. In contrast, thread switches are possibly exploited by an adversary. Suppose that program $\mathcal{P}$ is executed in multiple threads, with one of them executing the mission-critical function F. Consequently, the thread of F can be suspended or restored (supposing there is one CPU). We argue that an attestation on thread switch is necessary even in presence of Secure Kernel protection. In the proposed scheme, the integrity of the program $\mathcal{P}$ is not verified and Secure Kernel only protects the execution of F instead of the entire execution of $\mathcal{P}$. Hence, the memory curtaining mechanism of the Secure Kernel does not prevent a *tainted* thread of $\mathcal{P}$ from tampering with the thread of F, as the address space is shared. Similar to the one used in BIND[14], a countermeasure is to measure

the thread context. When a thread is suspended, the binary code of the thread and its context (e.g. registers) are measured. When the thread is restored to CPU, a trap will be generated and the handler verifies whether the loaded thread matches the previous measurement. Another thread may change the variables in the shared address space. If F does not have any data dependence on other threads, any changes on the variables accessed by F are regarded as an attack. Otherwise, the changes are measured by the trap handler.

6.4 Data Flow Attestation

Function attestation mainly deals with the integrity of the function logic. Unless the challenger has an expected input, the attestation does not ensure the trustworthiness of the input data. However, our scheme can be used as a building block to attest the data flow within a program. Suppose that the challenger is concerned about the trustworthiness of an output X from $\mathcal{P}$. A data dependency analysis on $\mathcal{P}$'s binary code can identify all the functions which X has a data dependence on. By attesting the execution of all dependent functions, the attester proves the integrity of the data flow. Compared with BIND [14] in this regard, our scheme does not require source code of $\mathcal{P}$, nor requires specialized compiler. Nonetheless, BIND can provide with an even more fine-grained attestation as it works with binary segments. We also remark that as in BIND, data flow attestation only ensures the integrity of primitive data [14], e.g. keyboard input, rather than its semantic.

7 Conclusion

In this paper, we propose a function attestation scheme, which allows a challenger to verify the execution of the mission critical functions within a program. Compared with prior work on TCG integrity attestation (efficient but ineffective) and program attestation (effective but inefficient), our scheme strikes the balance between efficiency and effectiveness. It allows for fine-grained attestation without using cumbersome known-good measurement databases. It does not rely on source code or specialized complier. Therefore, it is fully compatible with legacy applications. The proposed scheme can be easily extended to support multi-threaded programs and data flow attestation. We are currently implementing the proposed scheme. The experiment results will soon be shown in a full version of the paper.

Acknowledgements

This work is partly supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703 and partly supported by the Office of Research, Singapore Management University . We especially thank the anonymous reviewers for their very valuable comments and helpful suggestions.

References

1. Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A protocol for property-based attestation. In *STC '06: Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16, New York, NY, USA, 2006. ACM Press.
2. Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra a virtual machine-based platform for trusted computing. In *SOSP 2003*, Bolton Landing, New York, USA, October, 2003.
3. Liang Gu, Xuhua Ding, Robert Huijie Deng, Bing Xie, and Hong Mei. Remote attestation on program execution. In Shouhuai Xu, Cristina Nita-Rotaru, and Jean-Pierre Seifert, editors, *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, STC 2008, Alexandria, VA, USA, October 31, 2008*, pages 11–20. ACM, 2008.
4. Vivek Haldar, Deepak Chandra, and Michael 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.
5. Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual: Volume 1: IA-64 Application Architecture*. Intel Corporation, pub-INTEL:adr, January 2000.
6. Intel Corporation. *Intel IA-64 Architecture Software Developer's Manual: Volume 4: Itanium Processor Programmer's Guide*. Intel Corporation, pub-INTEL:adr, January 2000.
7. Trent Jaeger, Reiner Sailer, and Umesh 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.
8. Trent Jaeger, Reiner Sailer, and Umesh 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.
9. Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki. Flicker: an execution infrastructure for tcb minimization. In Joseph S. Sventek and Steven Hand, editors, *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*, pages 315–328. ACM, 2008.
10. Jonathan Poritz, Matthias Schunter, Els Van Herreweghen, and Michael Waidner. Property attestation—scalable and privacy-friendly security assessment of peer computers. Technical report, IBM Research Report RZ 3548, 2004.
11. Ahmad-Reza Sadeghi and Christian Stble. Property-based attestation for computing platforms: caring about properties, not mechanisms. *New security paradigms*, 2004.
12. Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *CCS 04*, Washington, DC, USA, October 25-29, 2004.
13. Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van 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.
14. Elaine Shi, Adrian Perrig, and Leendert Van Doorn. Bind: A fine-grained attestation service for secure distributed systems. In *2005 IEEE Symposium on Security and Privacy*, 2005.
15. Trusted Computing Group. Trusted platform module main specification. http://www.trustedcomputinggroup.org, October 2003.
16. Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, August 2002.
17. LI Xiao-Yong, SHEN Chang-Xiang, and ZUO 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.