Supporting Trusted Clock Service in the S/W–based kTPM

Quang Hieu Nguyen, Chanik Park

Computer Science and Engineering Department
Pohang University of Science and Technology, Gyeongbuk, Republic of Korea
hieunq@postech.ac.kr, cipark@postech.ac.kr

Abstract

Trusted Platform Module (TPM) is a hardware chip providing security features related to trusted computing such as encryption, decryption, and key management. It is observed that TPM chips are widely available in most of desktops and laptops. However, TPM chips are not available in mobile devices due to space and cost limitation. Therefore, there have been research on providing TPM functionalities by software in embedded devices. Software–based TPM is more flexible and better performance compare to the original hardware. But the challenges is to provide hardware–level trusted services to S/W–based TPM such as secure storage, secure entropy source, and trusted clock. Our previous work kTPM, which provides TPM functionalities by software in ARM–based embedded devices, supported secure entropy and trusted clock services by cloud. In this paper, we propose a method to support trusted clock service locally in kTPM, thus removing dependency of cloud trusted clock service. We make the trusted clock update integrity by isolating clock update in a secure part of kernel which is served as TCB. This method guarantees the trusted clock is updated periodically even if OS is compromised with the acceptable overhead (10% CPU utilization overhead at maximum).

1. Introduction

Trusted Platform Modules is a coprocessor which provides several security–related services including encrypt/decrypt, key generation, remote attestation. Because of strong security support, more and more devices and applications are running with the help of TPM e.g. BitLocker [7], ELAM [8]. Due to space and cost limitation, it is not practical to see TPM feature on mobile devices. Also, TPM chip is always a low–cost microprocessor which produces a low performance. Nowadays, there are some researches on software–based TPM (S/W–based TPM). fTPM [2] proposes a software implementation of TPM under ARM TrustZone [9] enforcement. Our previous work, kTPM [1] leverages kernel–level privilege to provide isolation execution space for S/W–based TPM. S/W–based TPM provides better speed since the TPM code is run on ARM Cortex Processor which is much better than low–cost TPM chip. The other advantage of S/W–based TPM is the flexibility. Developers can easily customize S/W–based TPM code and add the function they want to TPM.

However, in S/W–based TPM, the most challenge task is to provide hardware–level trusted services to S/W–based TPM including secure entropy source, secure memory, trusted clock. Secure entropy source is a source to generate the random value which can be used in key generation, etc. Secure storage is a storage which is used to stores TPM non–volatile objects. Trusted clock is a monotonic counter and its function is to measure lockout durations (e.g. time periods when TPM refuses services when the password is input wrong) and time–bound authorization (e.g. time to live of a secure token). Also, both secure entropy source and trusted clock must be secured which mean that only TPM can access to them. If not, malicious software can mount some attacks to rollback the clock value, read the random source to guess the key generated. fTPM proposes the implementation of secure storage, secure entropy, and trusted clock. fTPM offers a technique to provide trusted clock service by updating clock value when TPM command is issued. However, attackers can easily block TPM commands to prevent the clock from updating. Our previous work, kTPM depends on the trusted clock provided by the trusted cloud, similar to cTPM [10].

In the scope of this paper, we present the technique to provide local trusted clock service to S/W–based kTPM instead of previous trusted clock service from the cloud. Trusted Clock update’s execution is isolated from the untrusted kernel by mechanism same as Nested Kernel [4] or SKEE [5]. Clock value is written to the Replay Protected Memory Block (RPMB) partition. More details will be described in the following sections.

2. Threat assumptions

In general, attackers attempt to subvert TPM’s clock such as refuse to make clock update, rollback clock value. We
assume that kernel booting is protected by secure boot mechanism such as UEFI [3] so kernel's integrity is guaranteed during the boot process. Commodity OS is vulnerable and can be exploited to read and write kernel data arbitrarily (e.g., CVE-2013-6282 [6]) or to manipulate kernel control flow. Our job here is to guarantee the update of the trusted clock as well as keep clock value integrity.

Another type of attacks need considering is hardware attacks. For example, attackers can try to manipulate hardware clock to write wrong clock value. These attack methods are out of the scope of this paper.

3. Design and Implementation
3.1 Architecture

The Trusted Clock implementation inside a secure Interrupt Service Routine (ISR). ISR is moved to an isolated trusted part called the inner kernel same as Nested Kernel (Fig. 1). This modification ensures that the commodity untrusted kernel cannot have the access to or violate the execution of ISR. Clock persistent commands are issued from the trusted ISR and then forwarded to the RPMB block which resides in the untrusted outer kernel. Our extended secure ISR is now in charge of two jobs. The first is reading clock value from I/O timer and then periodically saving the clock value to secure storage (RPMB block).

3.2 Secure Interrupt Service Routine

During kernel execution, a “tick event” is the event which is generated frequently. The purpose of the tick event is to serve the process scheduling. The tick event periodically happens every few milliseconds (5–10 ms) and should not be bypassed or the system will collapse. Therefore we exploit this tick event to serve clock update. One of the ways attackers subvert the trusted clock is to try to skip clock update. Attackers can violate kernel–flow integrity and skip clock update execution. To prevent this attack method, we put the trusted clock update to the inner kernel. The inner–trusted kernel is a kernel de–privilege method similar to Nested Kernel [1] and SKEE [2] which isolate kernel Memory Management Unit, a part of kernel code and data from commodity kernel. The untrusted kernel cannot have any access to kernel code or data inside this area. By this approach, code execution and kernel data related to the trusted clock is protected from the untrusted commodity kernel. So, during the context of secure ISR, it executes two tasks consequently, the first task is the original task – scheduling related task and the additional task is updating the trusted clock value. So attackers can try to manipulate control–flow integrity to skip the second task. However, this approach is not possible with our system. To enter or exit the trusted interrupt service routine, it must bypass the entry or exit gate to gain the access permission to the inner kernel. So, attackers cannot arbitrarily jump into the middle of ISR and skip some ISR’s task. Putting ISR inside trusted–inner kernel makes sure all ISR tasks is done in a secure manner.

The trusted clock update needs to refer to a clock source, so we make the reference base on clock source I/O memory. Normally, a clock value is written to I/O memory by hardware clock. We do not want a clock value to be written from the commodity–untrusted kernel, so we set this memory area as read–only. Therefore the trusted clock update is safe from the threat coming from the untrusted kernel.

3.3 Secure Clock Persistent

With secure Interrupt Service Routine, protecting the trusted clock is not sufficient, yet. Attackers can just simply reboot the system to roll back the clock value. So the clock value must be periodically persisted to a non–volatile memory. We choose Replay Protected Memory Block (RPMB) partition as a candidate to store clock values. RPMB is a partition attached in embedded Multi–Media Controller that provides data storing in an authenticated manner. Reads and writes to the RPMB block are done by issuing an RPMB command. An RPMB command consists of three components: an authentication key, a write counter, and a nonce. The authentication key is a 32–byte one–time programmable key. This key is used to compute HMAC block to protect data integrity. HMAC is the first block of an RPMB frame which is computed from RPMB command content using a secret authentication key to protect command’s integrity (Fig.2).

On our implementation, the RPMB key which is used by RPMB module, is only visible from the trusted kernel, so the trusted kernel is the only part that can raise a valid RPMB command. Periodically, the clock value is written to the
3. Evaluation

We evaluated how the trusted clock affects the system because the periodically update the clock value can degrade the system. The evaluation environment is an ODROID XU4 board (Samsung Exynos5422 Cortex™-A15 2GHz and Cortex™-A7 Octa core CPUs, RAM 2GB). We measured CPU utilization overhead for 100s in the cases of the trusted clock and no trusted clock with the clock persistent gap is 100ms (Fig. 3). The average CPU utilization was 16% when the trusted clock was used, and 6% when it was not used.

![Figure 3 CPU Utilization in case of Trusted Clock (blue line) vs. W/o Trusted Clock (orange line)](image)

To reduce CPU overhead, the solution is to increase the frequency of the clock persistent. We measured in the case of the trusted clock with the frequency of clock persistent was 1 second. CPU utilization reduces from 16% (above case) to only 10%. However, by increasing the frequency, we must sacrifice the clock’s accuracy. Summary, the frequency of the clock persistent increased, the accuracy of clock increases but consequently leads the overhead increases.

4 Conclusions

This paper demonstrates how to overcome hardware limitations to provide trusted clock service for the S/W-based kTPM. We use inner kernel mechanism to protect the integrity of trusted clock execution and RPMB block to store clock value. Our job shows that this approach is feasible and can be applied to S/W-based kTPM with the acceptable overhead.