A Software-Based Root-of-Trust Primitive on Multicore Platforms

Qiang Yan†, Jin Han†, Yingjiu Li†, Robert H. Deng†, Tieyan Li‡
†School of Information Systems, Singapore Management University
‡Institute for Infocomm Research, A*STAR, Singapore
{qiang.yan.2008, jin.han.2007, yjli, robertdeng}@smu.edu.sg,
litieyan@i2r.a-star.edu.sg

ABSTRACT

Software-based root-of-trust has been proposed to overcome the disadvantage of hardware-based root-of-trust, which is the high cost in deployment and upgrade (when vulnerabilities are discovered). However, prior research on software-based root-of-trust only focuses on uniprocessor platforms. The essential security properties of such software-based root-of-trust, as analyzed and demonstrated in our paper, can be violated on multicore platforms. Since multicore processors are becoming increasingly popular, it is imperative to explore the feasibility of software-based root-of-trust on them.

In this paper, we analyze the challenges of designing software-based root-of-trust on multicore platforms and present two practical attacks that utilize the parallel computing capability to break the existing schemes. We then propose a timing-based primitive, called MT-SRoT, as the first step towards software-based root-of-trust on multicore platforms. MT-SRoT is able to ensure untampered execution of a critical security task, such as remote software attestation, on homogeneous shared-memory multicore platforms without the support of tamper-resistant hardware. We implement MT-SRoT and show its effectiveness on both Intel dual-core and quad-core processors.

Categories and Subject Descriptors

K.6.5 [**Operating Systems**]: Security and Protection

General Terms

Experimentation, Security

Keywords

Multicore, Software-based Root-of-trust, Trusted Computing

1. INTRODUCTION

As one of the core concepts of trusted computing, root-of-trust is a set of unconditional trusted functions which are essential to perform tamper-resistant actions. Conceptually, root-of-trust is able to verify the integrity of an application so as to detect whether it is tampered. If untampered, this application can then be used to perform integrity verification for another application. Following this mechanism, a chain of trust can be established so that any tampered application in this trusted chain will be detected. In trusted computing, root-of-trust acts as a local trusted party on a remote untrusted system, which dramatically simplifies the security problems involving remote software attestation [18].

Currently, the most practical way to ensure root-of-trust is to use tamper-resistant hardware, like hardware Trusted Platform Module (TPM) [1]. The major advantage of hardware-based root-of-trust is that it is physically secure. An adversary cannot forge fake root-of-trust without using sophisticated hardware techniques [1], if no design or implementation flaws exist. The major disadvantage of hardware-based root-of-trust lies in the hardness of replacing the vulnerable components or upgrading the outdated ones. The effort on fixing vulnerable hardware-based root-of-trust would be much more than that on fixing the Millennium Bug [2], as replacing hardware is more expensive and time-consuming than replacing software. Such design and implementation defects have indeed been observed on hardware-based security products, such as MiFare Classic RFID smartcard [8] and Intel Trusted Execution Technology [19].

To avoid the disadvantage of hardware-based root-of-trust, researchers attempt to build software-based root-of-trust without relying on tamper-resistant hardware. Unlike the former, the latter is easy to patch when vulnerabilities are discovered. Prior research on software-based root-of-trust [10, 16, 15, 17, 14, 21, 13] only focuses on uniprocessor platforms (more precisely, for devices which have a single processor without multicore and simultaneous multithreading features). Since multicore processors are becoming increasingly popular even for embedded devices (ARM Ltd. has just unveiled its multicore processor design for mobile phones [3]), it is imperative to explore the feasibility of software-based root-of-trust on multicore platforms.

In this paper, we analyze the challenges of building software-based root-of-trust on multicore platforms, and propose MT-SRoT, a timing-based primitive to overcome these challenges. The contributions of this paper are summarized as follows:

- We give a systematic analysis on the existing software-based root-of-trust schemes [10, 16, 15, 17, 14, 21, 13]. We show that their essential security properties can be violated on multicore platforms even after fixing them against the existing attacks [18, 20, 7]. Such vulnerability stems from the inherent availability of parallel computing capability on multicore platforms.
- We present two practical attacks against software-based root-

of-trust on multicore platforms. The first attack, which we call *pipeline parallelism attack*, parallelizes the existing schemes to break their timing property achieved only on uniprocessor platforms. The second attack, which we call *parallel interception attack*, exploits the parallel computing capability to nullify the effect of iterative checksum functions used in the existing schemes. Both of the attacks are able to eliminate the observable timing difference between running legitimate code and running tampered code.

- We propose a timing-based primitive, MT-SRoT, to construct software-based root-of-trust on multicore platforms. MT-SRoT is able to prove its own integrity to a remote verifier without the presence of tamper-resistant hardware. Based on that, it is able to ensure untampered execution of a critical security task on homogeneous shared-memory multicore platforms. We implement MT-SRoT and show its effectiveness on both Intel dual-core and quad-core processors.

Our work shows the feasibility of building software-based root-of-trust on multicore platforms and provides a primitive, MT-SRoT, as the first step towards establishing a tamper-resistant execution environment on multicore platforms, without the presence of tamper-resistant hardware.

2. SOFTWARE-BASED ROOT-OF-TRUST ON UNIPROCESSOR PLATFORMS

In this section, we provide an overview for existing software-based root-of-trust schemes on uniprocessor platforms, including the related definitions, threat model, assumptions, and desired properties. We further analyze the existing techniques for both constructing and attacking software-based root-of-trust. Based on this analysis, we revise the existing timing-based schemes to be resilient against all existing attacks on uniprocessor platforms under given assumptions. This preliminary preparation is necessary and important since the fixed scheme will be used as a building block for our software-based root-of-trust scheme on multicore platforms.

2.1 Definitions

Root-of-trust can be practically defined as a set of trusted functions, whose integrity can be remotely verified. For software-based root-of-trust, this condition is ensured by some challenge-response protocols [10, 16, 15, 17, 14, 21, 13] based on certain computationally hard problems. These problems guarantee that it is computationally difficult for the adversary to calculate the correct response with a tampered implementation of the protocol. The costs of solving the underlying computationally hard problems are significant enough to violate certain constraints, which exposes the adversary to the verifier.

Definition 1: *A software-based root-of-trust scheme is a tuple of algorithms and constraints* $(\mathsf{H}, \mathsf{INT}, \mathsf{EVC})$*, where* H *is an implementation of a chosen set of computationally hard problems,* INT *is a sub-algorithm of extracting integrity information, and* EVC *is a set of constraints for external verifiable measurements that do not rely on the existence of tamper-resistant hardware.*

Software-based root-of-trust can be set up via a challenge-response protocol. In this protocol, a trusted verifier who knows the expected states of a legitimate device intends to ensure untampered execution of a critical security task S on an untrusted device as shown in Figure 1.

In this paradigm, the implementation H (referred to as *verification code* H) of the chosen computationally hard problems and the critical security task S can be sent along with challenge c or

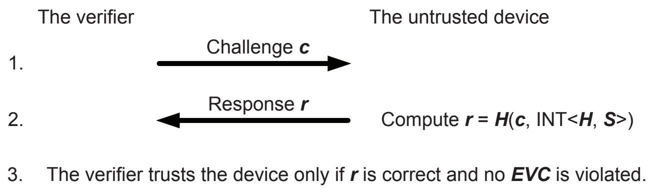

The verifier
The untrusted device
1.
Challenge c
2.
Response r
Compute r = H(c, INT<H, S>)
3. The verifier trusts the device only if r is correct and no EVC is violated.

Figure 1: The challenge-response paradigm used in the software-based root-of-trust schemes

sent beforehand. *INT*$\langle H, S\rangle$ is the integrity information (usually a checksum) derived from the verification code H and the critical security task S. After H is executed, an execution commitment is established so that the untampered execution of H will guarantee the untampered execution of S. The verification code H is currently implemented in the form of an *iterative checksum function*, where a *verification loop* is used to iteratively calculate the checksum values from the memory data picked from pseudorandom positions. If the adversary is forced to involve a constant overhead per iteration, this design is able to amplify the overhead of the adversary to the extent violating certain *EVC*. Until now, only the elapsed time t between sending a challenge and receiving its response has been used as *EVC* [10, 16, 15, 17, 14, 21, 13], where t should be within a certain threshold T. The correctness of root-of-trust setup is ensured by the chosen computationally hard problems. The verifier will trust the untrusted device only if the correct response r is calculated within the thresholds as specified by *EVC*. The execution commitment further ensures untampered execution of S.

2.2 Threat Model

In the threat models of existing software-based root-of-trust schemes [10, 16, 15, 17, 14, 21, 13], *the capabilities of an adversary are restricted to full control of the software systems on the untrusted device*. The adversary cannot modify the hardware specifications or other external configurations which may affect the measurement of external verifiable constraints. The goal of the adversary is to tamper with the execution of a critical security task after the "successful" setup of root-of-trust. In this way, a remote verifier is cheated by the adversary who claims that the critical security task is untamperedly executed but it is actually not. We will use this threat model in our following discussion for both uniprocessor platforms and multicore platforms.

2.3 Assumptions

A well-controlled measurement environment is essential for current software-based root-of-trust schemes to limit the capabilities of the adversary to full control of the software systems only. Therefore, the following assumptions are used for software-based root-of-trust schemes: a) *An adversary cannot add, remove, or change any hardware components of the untrusted device.* b) *The untrusted device must be a uniprocessor device* so that isolated execution ensured by a non-preemptive execution environment with Direct Memory Access (DMA) disabled is sufficient to guarantee the execution commitment without relying on extra hardware support. However, the verifier usually cannot ensure that DMAs from the peripherals are disabled. Thus, it is also assumed that no DMA-based attacks are performed [15]. c) When elapsed time between sending the challenge and receiving the result is used as an external verifiable constraint, *communication latency between the remote verifier and the untrusted device can be estimated as a constant value* so that the threshold for maximum allowed time can be determined. When tight bounds of elapsed time are used as the constraints, it is further assumed that the processor is not overclocked. d) *The untrusted device is only able to communicate with the remote verifier during the whole process of establishing the root-of-trust.*

In real scenarios, these assumptions are practically true for uniprocessor embedded devices in a local network. In our following discussion, we will *keep* the above assumptions and *relax* the uniprocessor assumption by allowing multicore devices with the hardware feature of *isolated execution* [4, 5]. Isolated execution is to ensure that the measured application is the actual application running without tamper, after the software-based root-of-trust is set up on multicore platforms.

2.4 Desired Properties

Since the above assumptions guarantee the availability of a well-controlled measurement environment, the remaining design challenge for software-based root-of-trust is to design the algorithm for *INT*, the computationally hard problems for H, and external verifiable measurements for *EVC*. *INT* can be implemented by well-designed checksum functions [16, 15], while the key idea of designing H and *EVC* is to force the adversary to incur significant overhead in the execution of the tampered verification code. Therefore, the essential security properties for a secure software-based root-of-trust scheme can be summarized as follows:

- **Asymmetric property**: It is unavoidable for an adversary to solve the chosen set of computationally hard problems used in H so as to calculate the correct response with tampered verification code; while the legitimate verification code does not need to solve these hard problems.
- **Observable property**: The costs of solving the chosen set of computationally hard problems used in H should be significant enough to violate the constraints *EVC* for the external verifiable measurements. And the process of calculating these constraints should be deterministic.

2.5 Review of Existing Schemes

The choice of the computationally hard problems is critical in the design of software-based root-of-trust schemes. Several computationally hard problems have been proposed in the existing constructions, which are based on a) slow simulation of side effects of hardware architecture [10], b) difficulty of recovering the seed from pseudorandom sequences [21], c) difficulty of static analysis of self-modifying code [17], and d) difficulty of keeping the execution speed when extra operations are involved [16, 15, 14, 13], respectively. Unfortunately, it was later found [18, 7] that all of them do not satisfy the asymmetric property, except the last one. That is, the adversary is able to calculate the correct response without solving the first three hard problems, while hiding the malicious data. Hence, we focus on the schemes based on the last computationally hard problem in the following discussion. This class of schemes, which is referred to as *timing-based schemes*, can be described as follows:

- **Instances:** SWATT [16], Pioneer [15], and Indisputable Code Execution (ICE) [14, 13]
- **Hard Problem:** It is hard to run the tampered verification code as fast as the time-optimal implementation of the legitimate code when the tampered code involves extra actions to hide malicious data.
- **EVC:** Tight bounds for elapsed time between sending a challenge and receiving its response.

The computationally hard problem used in timing-based schemes inherently achieves the asymmetric property if the following two conditions are satisfied: 1) **Time-optimal condition:** The time-optimal implementation is available for the legitimate verification code. 2) **Full-occupation condition:** No computation resources are left idle during the execution of the time-optimal verification code. When these two conditions are met, it is logically impossible for an adversary to hide the time cost introduced by the extra actions of obscuring the malicious data.

The extra time cost of the tampered code depends on the specific actions introduced by an adversary. Since this construction technique does not produce a significant time gap between measuring tampered code and legitimate code, tight timing bounds are used based on prudent analysis [16, 15]. Although the tight bounds may narrow the application area of this method, and it is challenging to determine the proper timing bounds as required by the observable property, it is the only existing technique that satisfies all essential security properties on uniprocessor platforms.

Several attacks [18, 20, 7] have been proposed against the implementations of the timing-based schemes for software-based root-of-trust. These attacks include: a) most significant bit (MSB) cancellation attack [16] that exploits the cancellation property of the checksum function, b) hardware-based memory mapping attack [20] that exploits the virtual memory mapping of modern processors, c) return-oriented programming based attack [7], and d) threshold overestimation attack [7]. The first two attacks redirect the memory mapping such that tampered code is able to efficiently calculate the correct response by reading untampered data without significant time overhead. These two attacks have been addressed and solved in the latest implementation [15] of timing-based schemes. The last two attacks are recently discovered [7]. The third attack crafts a return stack that gains the control of code execution from return instructions, which defeats the schemes designed solely against code injection. The remedy for this attack is to simply avoid any indirect jump instructions with unverified jump targets; that is, the verification process should call the function implementing the critical security task before it is returned. The last attack shows the possibility of overestimating the detection thresholds by designing a stronger adversary than that given in the original paper [16]. However, it can be easily solved as new thresholds can be updated accordingly after new attacks are found. It will be more and more difficult for the adversary to develop a stronger new attack. Therefore, the timing threshold will converge to its theoretically best value after the timing-based schemes are deployed for a period of time.

To summarize, the timing-based schemes are robust against all existing attacks on uniprocessor platforms under given assumptions as justified in their original papers [16, 15]. We refer to these timing based schemes as instances of *T-SRoT*, which is defined as an abstract Timing-based scheme for Software-based Root-of-Trust on uniprocessor platforms. We will use it as a building block in our design of software-based root-of-trust for multicore platforms in Section 4.

3. CHALLENGES FROM MULTICORE PLATFORMS

The most significant difference between uniprocessor platforms and multicore platforms is that parallel computing units are naturally available for an adversary on multicore platforms. *The usage status (e.g. disabled or active) of these computing units cannot be remotely verified without tamper-resistant hardware [4, 5]. Thus, the adversary could use unoccupied computing units to break timing-based schemes for software-based root-of-trust.* We present two types of attacks to T-SRoT when implemented on multicore platforms, depending on whether the parallel computing units work cooperatively or alone. In analysis, we assume that T-SRoT is used as the target software-based root-of-trust scheme deployed on one computing unit, while the remaining computing units are left idle for the use by the adversary.

For ease of reference, Table 1 lists the terminologies and experimental settings used in this paper. Note that in our analysis and experiments, we always use multicore desktop PCs, even for testing the schemes that are mainly designed for embedded devices. The major reason is that multicore embedded devices are currently not publicly available in the market. However, using desktop PCs does not affect the effectiveness of any conclusion drawn from our experiments. All involved schemes are implemented in C and inline assembly codes, which are tuned to approximate the optimal implementation as required in the original papers [16, 15].

Table 1: Terminologies and Experimental Settings

Multicore platform	A multicore platform uses current homogeneous shared-memory multicore architecture, where each core on the platform has the same capabilities and can access to the memory used by another core. We only consider the homogeneous shared-memory multicore platforms in the construction of MT-SRoT, and leave the other multicore platforms for our future work.
Computing unit	A physical computing unit is able to perform computation task simultaneously with the other physical computing units. Each computing unit has the same capabilities on homogeneous shared-memory multicore platforms. For example, one core of multicore processor without simultaneous multithreading feature is a computing unit.
Dual-core machine settings	Intel Core 2 Duo 2.53GHz CPU E7200, 4GB DDR2 Memory, Ubuntu Server 9.10 (64-bit).
Quad-core machine settings	Intel I7 2.8GHz CPU 860 with Turbo Boost and simultaneous multithreading disabled to avoid the variation of CPU frequency, 4GB DDR3 Memory, Ubuntu Server 9.10 (64-bit).

3.1 Pipeline Parallelism Attack

Figure 2(a) illustrates our first attack, which we call *pipeline parallelism attack*. This attack exploits a vulnerability in all existing software-based root-of-trust schemes [10, 16, 15, 17, 14, 21, 13], in which pipeline parallelism is not considered in formulating the non-parallelism property. The basic idea of pipeline parallelism attack is to partition the verification loop into several stages and execute each stage in parallel on different computing unit. Provided that the data dependency between stages can be decoupled, each stage can execute the next iteration without waiting for the output from its next stage. This attack is practical on multicore platforms due to the low latency in communications among the computing units. Pipeline parallelism attack can be used to violate the time-optimal condition of timing-based scheme because the pipelined implementation on multiple computing units may execute faster than the time-optimal implementation on one computing unit.

The effect of pipeline parallelism attack depends on the partition of stages and the inter-stage communication cost. If the execution time of implementing T-SRoT on one computing unit is T, the execution time for its pipelined implementation is $T_i + C_i$, where T_i and C_i are the execution time and communication time of the *longest* stage i, respectively. As T_i is less than T, the pipelined implementation will run faster than the original implementation when the communication cost satisfies $C_i < T - T_i$.

To illustrate the process of pipeline parallelism attack, we use SWATT [16] as an example for T-SRoT. The original SWATT code can be partitioned into two pipeline stages that run on two computing units, as shown in Figure 2(b). These two stages are referred to as *stage A* and *stage B*, respectively. In SWATT, there is a natural partition point since the execution of RC4 pseudorandom number generator is not relied on the output of checksum computation.

We test the effectiveness of pipeline parallelism attack on our dual-core machine. To minimize the communication cost, we use the latest cache-optimized lock-free message queue implementation [9]. This message queue is used to hand over the pseudorandom numbers generated from the first stage to the second stage, which calculates the checksum based on these pseudorandom numbers. Figure 3 gives the speedup ratio of pipelined SWATT code compared to the original SWATT code. For 64KB target memory, the pipelined code runs 1.8% faster than the original code after 2,500,000 iterations. This is a significant improvement since the time-optimal Pioneer is only 1.3% faster than its theoretically best adversary after the same number of iterations on modern x64 platforms [15].

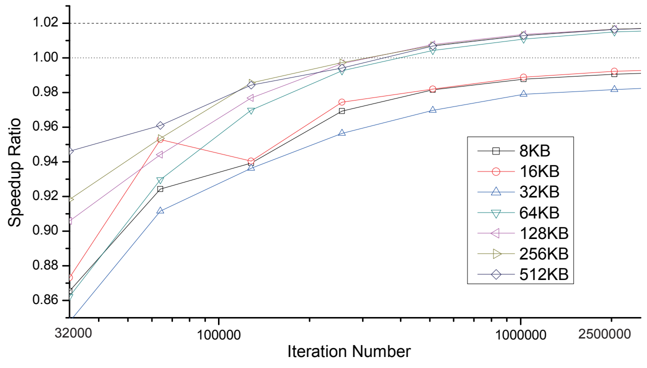


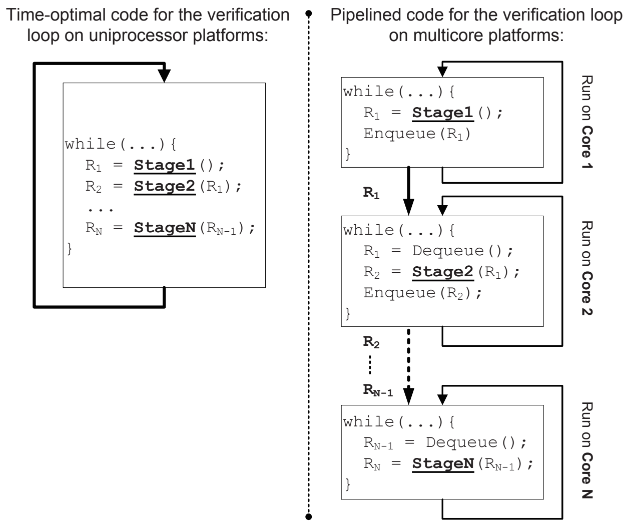
Figure 3: Speedup ratio of pipelined SWATT code compared to the original SWATT code for different memory sizes on the dual-core machine. The original SWATT is reimplemented for x64 architecture.

Figure 3 also shows two interesting results of pipeline parallelism attack: 1) *The pipelined code does not run faster than the original code until the number of iterations is large enough.* This slowdown is caused by the warmup procedure of the pipelined code that fills the message queue. When the message queue is filled, each stage can continue execution without waiting for the outputs from the previous stage. 2) *The speedup ratios are much lower than expected as only several percentage points are achieved by doubling the number of computing units.* This inefficiency is caused by relatively high communication cost between two consecutive stages.

In our experiments, the average communication cost including waiting time is 26 cycles per iteration for *stage A* and 9 cycles per iteration for *stage B*, while the original code takes 37 cycles per iteration. It shows that the communication cost is significant on current multicore machines when the code size of the verification loop is small. Nonetheless, 1.8% time advantage is significant enough to show the potential of using multiple computing units to break the time-optimal condition of T-SRoT on multicore platforms.

This vulnerability of SWATT [16] also exists in other software-based root-of-trust schemes [10, 15, 17, 14, 21, 13]. In the verification code of these schemes, the lightweight pseudorandom number generators also take no output from checksum computation. It is thus feasible to parallelize the time-optimal implementations of these schemes on multicore platforms.

Our remedy for this vulnerability is to introduce a backward data dependency from the end of the verification loop to its beginning. This can be implemented by using a parameterized pseudorandom number generator that takes the current checksum value as its input so that only one stage can execute at any time while the other


Time-optimal code for the verification loop on uniprocessor platforms:
Pipelined code for the verification loop on multicore platforms:
R1
R2
RN-1
Run on Core 1
Run on Core 2
Run on Core N

```
while(...){
  R1 = Stage1();
  R2 = Stage2(R1);
  ...
  RN = StageN(RN-1);
}
```

```
while(...){
  R1 = Stage1();
  Enqueue(R1)
}
```

```
while(...){
  R1 = Dequeue();
  R2 = Stage2(R1);
  Enqueue(R2);
}
```

```
while(...){
  RN-1 = Dequeue();
  RN = StageN(RN-1);
}
```

(a) Concept

Generate i-th member of random sequence using RC4 cycles		Cycles in the Simulation[16]	
zh ← 2	ldi r31, 0x02	1	**Stage A (13 cycles)**
r15 ← *(x++)	ld r15, x+	2	
yl ← yl + r15	add r28, r15	1	
zl ← *y	ld r30, y	2	
*y ← r15	st y, r15	2	
*x ← zl	st x, r30	2	
zl ← zl + r15	add r30, r15	1	
zh ← *z	ld r31, z	2	
Generate 16-bit memory address			**Stage B (10 cycles)**
zl ← r6	mov r30, r6	1	
Load byte from memory and compute transformation			
r0 ← *z	lpm r0, z	3	
r0 ← r0 ⊕ r13	xor r0, r13	1	
r0 ← r0 + r4	add r0, r4	1	
Incorporate output of transformation into checksum			
r7 ← r7 + r0	add r7, r0	1	
r7 ← r7 ≪ 1	lsl r7	1	
r7 ← r7 + carry bit	adc r7, r5	1	
r4 ← zh	mov r4, r31	1	

(b) SWATT code and pipeline partitions

Figure 2: Pipeline Parallelism Attack

stage has to wait for its output. The parameterized pseudorandom number generator can be implemented based on two lightweight pseudorandom number generators. The current checksum value is used as a dice to select one of the two generators to generate the next pseudorandom number.

3.2 Parallel Interception Attack

Figure 4 illustrates our second attack, which we call *parallel interception attack*. This attack can be launched by an idle computing unit alone. It exploits resource sharing of multicore platforms to intercept the initialization process of the secure execution environment which cannot be finished in one atomic operation on current hardware implementations [4, 5]. For example, the `senter` instruction of Intel Trusted Execution Technology can be successfully invoked only after an initialization process according to its official developers' manuals [4]. This process consists of at least 410 instructions estimated from the size of the last initialization method `txt_launch_environment` in its official open source implementation (Trusted Boot [6]). It is thus reasonable to estimate that the time cost of the initialization process is at least 200 cycles. We will use 200 cycles to represent the cost of the initialization of the secure execution environment in our experiments.

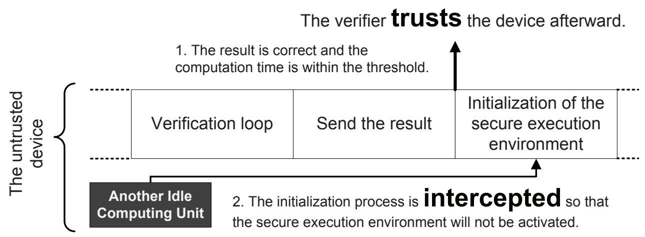

The verifier trusts the device afterward.
1. The result is correct and the computation time is within the threshold.
The untrusted device
Verification loop
Send the result
Initialization of the secure execution environment
Another Idle Computing Unit
2. The initialization process is intercepted so that the secure execution environment will not be activated.

Figure 4: The basic idea of parallel interception attack is to intercept the initialization process of the secure execution environment after the legitimate verification code sends the correct report to the verifier. The verifier is cheated to believe the secure execution environment is untamperedly launched while it is actually not.

Although this attack seems straightforward compared to pipeline parallelism attack, it is more serious as it only requires the availability of an idle computing unit for a short time period. This attack is able to violate the asymmetric property required for software-based root-of-trust. The adversary is now able to gain the control of the execution after the legitimate code sends the correct report without solving the underlying computationally hard problems. This attack is also able to violate the observable property as it takes almost zero cost to finish the interception operation and this cost cannot be amplified by the iterative checksum design. Thus, it will be very difficult to determine any significant thresholds for attack detection.

A simple technique to perform the interception attack is to change the instruction memory. This technique will not fail as long as the memory, which stores the last instruction of the initialization process of the secure execution environment, can be changed at the time between the end of report-sending instructions and the end of the initialization process. To decide the time point when to perform the interception operation, the adversary can estimate the execution time between the start of verification code and the end of report-sending instructions. And this execution time period is estimable because the verifier also uses it to decide the timing thresholds.

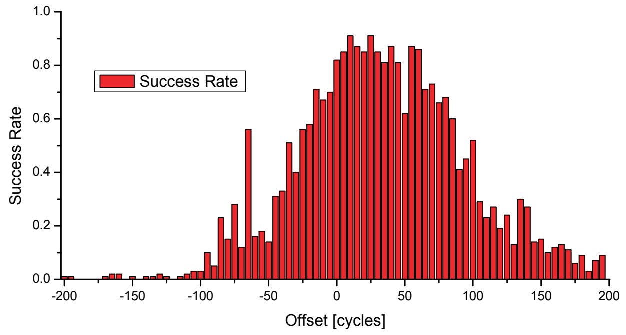


Figure 5: Success rate of one-off parallel interception attack

Figure 5 shows the success rate of one-off interception actions in our experiments, where we use 200 cycles as the estimated time cost for the initialization of the secure execution environment. In our experiments, an adversary estimates the time point to perform the interception operation from different offset, which is based on the estimated cycles between the start of verification code and the end of report-sending instructions. The adversary intends to replace the function pointer at the end of the initialization process of the secure execution environment. It will succeed only when the interception time point is within the 200-cycle gap between the end of report-sending instructions and the end of the initialization process. Each bar in the figure represents the success rate of 100 samples at certain offset. Zero offset represents the estimated time point of the end of report-sending instructions.

The results in Figure 5 show that the adversary is able to achieve over 80% success rate with only one trial to perform interception action if the execution time of legitimate code can be reasonably estimated. Our remedy for this attack is to use two-stage timing report, where the computing units that do not perform the critical security task will be assigned with heavy workloads so that they cannot send any responses before the secure execution environment has been initialized. This will guarantee the successful initialization of the secure execution environment when all the idle computing units are occupied for the computation of their responses. It works because the adversary has to first finish the computation of heavy workloads within the time constraint specified by the verifier.

4. DESIGN OF MT-SROT

MT-SRoT is a software-based root-of-trust scheme designed for multicore platforms. It is a timing-based scheme, able to handle the new challenges from the parallel computing units on multicore platforms as identified in this paper. In the following, we detail the design of MT-SRoT, and analyze why this scheme is able to achieve the two essential security properties of software-based root-of-trust on multicore platforms.

4.1 Conceptual Design

In the design of MT-SRoT, we use T-SRoT as a building block and assume that it is able to resist all existing attacks on a single computing unit, as it has been justified in prior work [16, 15]. Since only abstract operations of T-SRoT are used in MT-SRoT, any T-SRoT instance such as Pioneer [16], SWATT [15], and other existing schemes [14, 13] can be used in MT-SRoT. The appropriate T-SRoT instance should be selected according to the hardware architecture of the target platforms. Figure 6 shows the flow diagram of MT-SRoT.

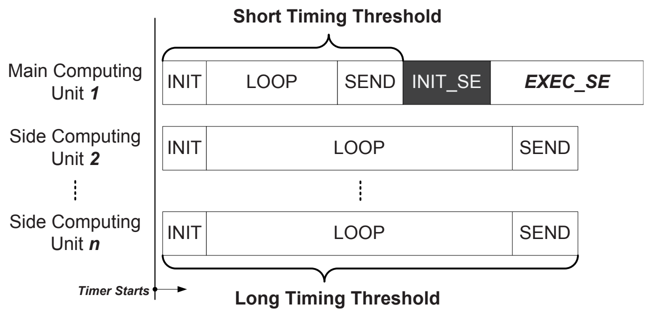


Figure 6: Flow diagram of MT-SRoT

MT-SRoT assigns an instance of *T-SRoT* to each available computing unit, where each instance contains three common abstract operations: *INIT* for initialization, *LOOP* for verification loop, *SEND* for checksum report sending. The computing units are divided into two groups, *side computing units* that only perform the verification task and *main computing units* that also perform the critical security task. The main computing units have two extra abstract operations: *INIT_SE* for the initialization of a secure execution environment, and *EXEC_SE* for the execution of a critical security task under the secure execution environment. Since we assume the availability of isolated execution for the secure execution environment on multicore platforms, the execution of a critical security task *EXEC_SE* is secure after successful invocation of *INIT_SE*. We further assume that the activation of the secure execution environment on one computing unit will not affect the activities on other computing units. These features can be achieved using the improved hardware design proposed in [11].

Given a verifier and an untrusted multicore computing device, MT-SRoT works as follows:

1. The verifier sends the untrusted device a set of challenges $C = \{c_1, c_2, \ldots, c_N\}$ for each *T-SRoT* instance on a multicore platform with N computing units. The code of these instances can be sent beforehand or together with the challenge. After that, the verifier starts timing for each challenge c_i.
2. After receiving the challenge C, the untrusted device deploys an instance of *T-SRoT* for each available computing unit and starts running them as early as possible. The instance of *T-SRoT* on computing unit i uses c_i as an parameter to perform *INIT* operation, which includes setting seed for pseudorandom number generator, and initializing control variables.
3. Each instance of *T-SRoT* performs *LOOP* operation to calculate the checksum of a single memory partition which is coded into the implementation of *T-SRoT*. The memory region consisting of all memory partitions should cover the entire target memory region, including the verification code, the critical security task, and other contents to be verified.
4. After a computing unit finishes its LOOP operation, it performs SEND operation to send the response $(c_i, csum_i)$ back to the verifier, and continue to do *INIT_SE* and *EXEC_SE* operations if it is a main computing unit.
5. After receiving the response $(c_i, csum_i)$, the verifier stops its timer for challenge c_i and checks the correctness of checksum $csum_i$. It also checks whether the elapsed time t_i for c_i is within the timing threshold T_i. Different timing thresholds are used for main computing units and side computing units. This is because heavy workloads (e.g., by increasing the iteration number of verification loop) are assigned to side computing units so as to keep them busy before *INIT_SE* operations are finished by the main computing units. The verifier will trust that the security task has already been launched under a secure execution environment only if all the checksums are correctly calculated within the respective timing thresholds.

To minimize the interference among the computing units during the verification process, each memory partition should be aligned to the cache line boundaries and not be intersected with each other so that coherence cache misses and other non-compulsory cache misses will be minimized. Besides these restrictions, each memory partition should be a continuous memory region; otherwise, the existence of discrete memory segments may degrade the efficiency of cache system. An adversary may exploit this inefficiency to break the time-optimal condition by rearranging the memory segments. For modern processors with separated local cache for each computing unit, the size of each memory partition should be smaller than the size of the local cache such that no cache misses occur after warmup. This is also required in certain instances [15] of

T-SRoT, which will minimize the impact of variance of memory access latency.

The verification process of MT-SRoT is not exactly the same as T-SRoT, since the target memory region is now divided into several partitions and verified by multiple computing units in parallel. On uniprocessor platforms, it is vulnerable to divide the target memory region into partitions and verify them separately [21]. This is because an adversary may move the tampered data from one partition that is currently being verified to another partition that is not being verified. However, this attack does not work for MT-SRoT because any tampered datum within the target memory region will be verified by one of the computing units at any time (unpredictable to adversary) before the end of verification process. Therefore, MT-SRoT provides the same integrity guarantee as provided by the instances of T-SRoT in MT-SRoT.

Current design of MT-SRoT only considers the homogeneous physical cores in the main CPU of the device. The other computing units, such as GPU and disk controller, may exist on the same device. These computing units usually have different capabilities and work as slaver processors that execute the commands from the main CPU. It is possible for the adversary to schedule these computing units to intercept the establishment of software-based root-of-trust by DMA or similar mechanisms. This attack is difficult to handle, and thus assumed not to happen in the all existing T-SRoT schemes [15]. It may be feasible to solve this problem by crafting challenges for these computing units. This extension requires considerable efforts to handle such heterogeneous distributed-memory many-core systems, which is left as our future work.

4.2 Asymmetric Property

Recall that a timing-based scheme satisfies the asymmetric property if both time-optimal condition and full-occupation condition are fulfilled. However, these conditions can be easily violated on multicore platforms even for the schemes that are secure on uniprocessor platforms. To meet these conditions on multicore platform, we introduce three new properties:

- **Full-parallelism property:** The verification code should be fully parallelized on each computing unit. It can be achieved by assigning each computing unit with an independent task so that each task can be independently optimized without considering the inter-core communication cost.
- **Data-independent property:** The data processed in independent tasks of verification code should be independent so that the adversary cannot use inter-core communication to accelerate the computation (e.g., by reusing the common intermediate results of any two tasks).
- **Full-coverage property:** The execution time of the tasks assigned to the computing units that do not perform the critical security task should be long enough to cover the initialization process of the secure execution environment.

The first two properties are mainly designed to achieve the time-optimal condition. It is well known that it is difficult to achieve the time-optimal implementation for generic computation tasks on multicore platforms. The reason is that the tradeoff between computation and communication costs dramatically expands the search space of possible implementations. Therefore, we introduce full-parallelism property and data-independence property and use these two properties to reduce the search space. We further introduce the following conjecture for current multicore machines to solve the time-optimal problem.

Cooperation Efficiency Conjecture: *For any two computing units with equal computational capability, the efficiency when they work individually to solve two independent* ***computationally intensive non-parallelism*** *tasks (each computing unit solves one task) will* ***not be lower*** *than that when they work cooperatively (together the two units solve two tasks), if the following three conditions are satisfied: 1) they always use time-optimal implementations; 2) the local cache of individual computing units is large enough to hold the entire working set of the assigned task; 3) there are no common sub-tasks in the chosen tasks.*

This conjecture can be realized on current mainstream homogeneous multicore processors, like Intel Core processors. Due to the page limit, we only give the basic idea for the proof of this conjecture: The condition of no common sub-tasks implies that the cooperation strategy will only incur extra communication cost without saving any computation cost. Our experiments of pipeline parallelism attack further show that these communication costs are still significant on current multicore platforms. The major challenge of implementing this conjecture is to ensure the condition of no common sub-tasks. It is difficult to absolutely guarantee, but it can be easily approximated by using different seeds for the same pseudorandom number generator used in the verification code. The probability of the occurrence of a long common pseudorandom number subsequence will be very low when using two different seeds for a non-flawed pseudorandom number generator. Therefore, the probability of the occurrence of a reusable common sub-task, which requires a long common pseudorandom number subsequence to offset the extra communication cost, is also very low.

Under the cooperation efficiency conjecture, the first two properties actually reduce the time-optimal problem on multicore platforms to the time-optimal problem on each computing unit, which is the same as the problem that has been solved in the timing-based schemes for uniprocessor platforms [16, 15]. Figure 7 shows the feasibility of achieving time-optimal condition on multicore platforms. In the experiment of parallel checksum function, each available computing unit is required to simultaneously verify a non-overlapped continuous memory region whose size is the same as that of its counterpart executed on a single computing unit alone. The longest finish time in these computing units is chosen as the result. The total iteration number for each computing unit that verifies M-size memory is $M\ln(M)$ based on the coupon collection problem used in the prior work [16, 15]. The theoretical best time ratio of parallel checksum function is 1.0 when there is zero interference cost between these computing units. Figure 7 shows that our implementation of parallel checksum function almost reaches the theoretical best result. And the interference among multiple computing units is insignificant even when the size of assigned non-overlapped memory partitions is larger than the size of local L1 data cache, 32KB.

Besides the above effects, these two properties also partially solve the full-occupation problem as prior work [15] has shown that the timing-based schemes can be tuned to occupy all the computation resource on a single computing unit. Nonetheless, it is still possible for an adversary to launch parallel interception attack if all of the tasks assigned to a computing unit finish before the end of the initialization process of the secure execution environment. Therefore, we need the last property, full coverage property, to achieve the full-occupation condition by ensuring that all available computing units finish their tasks after the initialization process of the secure execution environment. These tasks are protected by the interrupt-resistant mechanisms introduced in the instances of T-SRoT [15], which guarantee that the current checksum value will be overwritten when an adversary attempts to reschedule any task (by an unexpected software or hardware interrupt). Hence, both time-optimal and full-occupation conditions for our timing-based scheme are sat-

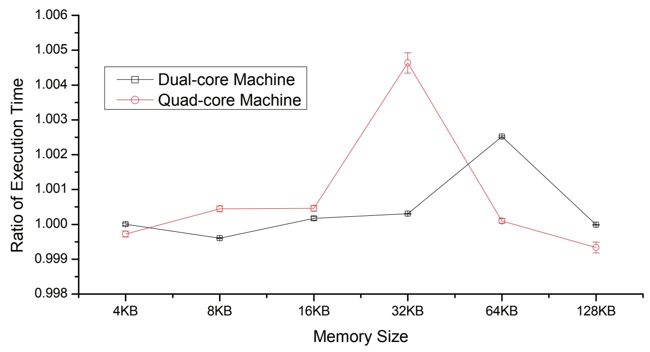


Figure 7: Time ratio between running time-optimal implementation on a single computing unit and running parallel implementation of the same checksum function on multiple computing units

isfied, which leads to the asymmetric property.

4.3 Observable Property

The observable property requires deciding a tight threshold to distinguish between legitimate verification code and tampered verification code. This threshold decision problem on multicore platforms can be reduced to the threshold decision problem on uniprocessor platforms under the cooperation efficiency conjecture. The conjecture implies that cooperative attacks against our scheme by multiple computing units are not as efficient as individual attacks to each computing unit. It is because the communication among multiple computing units does not accelerate the whole verification process if there is no common intermediate result to share in the verification process. So the best strategy for the adversary is to tamper with verification code on the computing units that verify the memory region containing the tampered data.

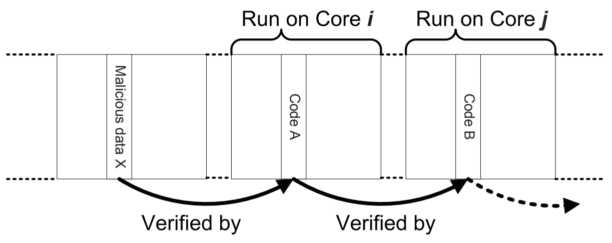


Figure 8: Chain effect caused by taint propagation

To amplify the overhead of the adversary on homogeneous shared-memory multicore platforms, a chain effect can be designed for memory partition scheme so as to force the adversary to tamper with all computing units, even when its purpose is only to tamper with one byte within the target memory without being detected. As illustrated in Figure 8, when the adversary tampers with one part of the verification code, the taint propagation will in turn forces it to tamper with other parts of the verification code. This effect can be implemented by a *check ring*, where the memory partition that contains the verification code for one computing unit is assigned to its next computing unit (the first computing unit is assigned as the next computing unit of the last computing unit). If the verification code executed on one computing unit is tampered, the check ring will force the adversary to tamper with all of the verification codes for the remaining computing units.

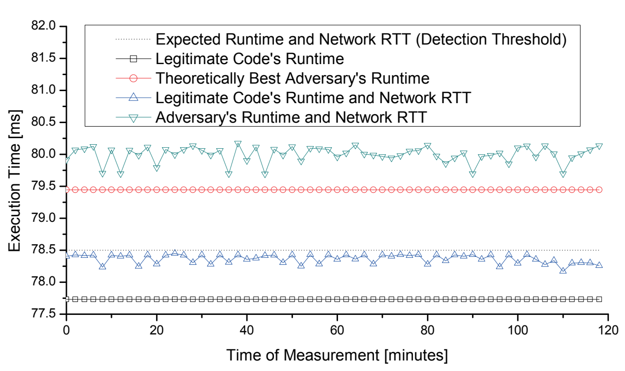
For those computing units that execute tampered verification codes, the minimum time for finishing their tasks alone can be estimated in the same way as estimated on uniprocessor platforms [16, 15]. However, the estimated time cannot be used as the timing thresholds if the adversary is able to accelerate the remaining computation process by using the idle computing units that finish the execution of the legitimate verification code. This unwanted situation is prevented due to the non-parallelism property [15] of T-SRoT with our fixings against pipeline parallelism attack. Therefore, we can reuse the analysis results of the theoretical best adversary [16, 15] to determine the timing thresholds in our scheme so as to satisfy the observable property.

5. IMPLEMENTATION AND EVALUATION

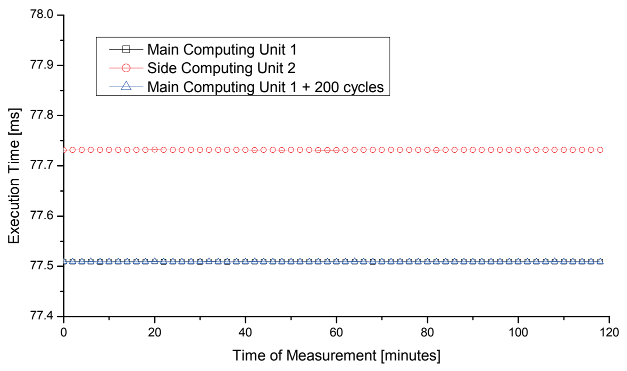
We implement MT-SRoT on our multicore machines of x64 architecture (referring to Table 1 for experimental settings). Pioneer [15] is chosen as the instance of *T-SRoT*. According to the original paper of Pioneer, the target memory region to be verified should include the verification code and a critical security task. Our memory partition scheme evenly divides this memory region into N continuous partitions for a machine with N computing units. To determine the iteration number and timing threshold, the original analysis requires estimating the time overhead and advantage of theoretical best adversary including the time of extra computation, network round trip (RTT) and cache warmup. In our experimental settings, the untrusted devices are one dual-core machine and one quad-core machine located at two different physical locations on our LAN segment. For the dual-core machine, the time overhead o is 1.71 cycles per iteration of the verification loop; the upper bound of RTT time a_{RTT} between the verifier and the untrusted device is 0.755ms; the cache warmup time a_{cache} is 0.00021ms. For the quad-core machine, the time overhead o is 0.93 cycles per iteration of the verification loop; the upper bound of RTT time a_{RTT} between the verifier and the untrusted device is 0.597ms; the cache warmup time a_{cache} is 0.00011ms.

To keep side computing units busy before the end of the initialization of the secure execution environment, we also need to estimate the extra workload assigned to side computing units from the initialization cost $a_{\mathrm{INIT\_SE}}$, which is 200 cycles estimated from the latest official open source implementation of Intel Trusted Execution Technology [6], and the upper bound of the execution time variations of the main computing units $a_{\mathrm{VAR\_MAIN}}$. The required minimum iteration number is $Itn_{\mathrm{main}} = \max((a_{\mathrm{RTT}} + a_{\mathrm{cache}}) * clockSpeed/o, M\ln(M))$ for main computing units and $Itn_{\mathrm{side}} = Itn_{\mathrm{main}} + 2 * \max(a_{\mathrm{INIT\_SE}}, a_{\mathrm{VAR\_MAIN}})$ for side computing units, where $clockSpeed$ is the clock speed of each computing unit, and $\max(a_{\mathrm{INIT\_SE}}, a_{\mathrm{VAR\_MAIN}})$ is doubled to ensure that side computing units cannot finish its workload before the end of *INIT_SE* operation, and M is the size of the assigned memory partition. To prevent any false positives due to RTT variations, we double the minimum iteration number as the original Pioneer does. For 16KB target memory region, the minimum iteration number is 2,400,000 for main computing units and 2,409,520 for side computing units on the dual-core machine, and 3,600,000 for main computing units and 3,612,792 for side computing units on the quad-core machine. The timing threshold for the verifier is set as the sum of the computation time taken by the legitimate verification code for the specified iteration number and the upper bound of RTT time. The verifier considers the root-of-trust setup fails if a response from any computing unit is received after its timing threshold.

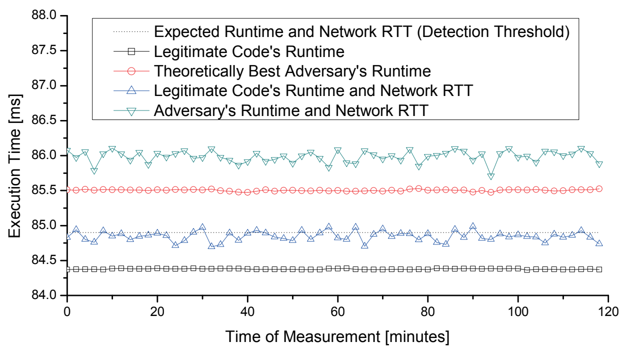
We run our experiment for 2 hours and take a sample every two minutes. For each sample, the verifier invokes two executions of MT-SRoT on the untrusted device by using the legitimate verification code and the tampered verification code of the theoretical best adversary, respectively. Each execution of MT-SRoT includes the



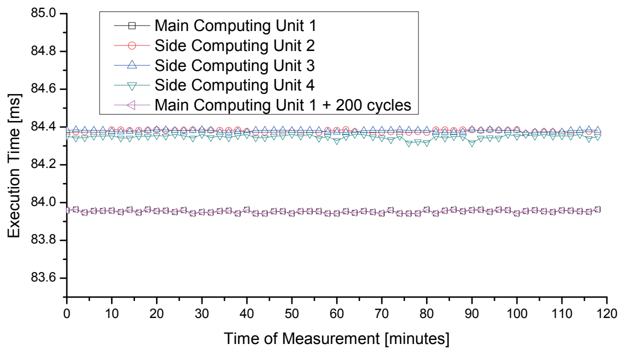
(a) Legitimate code vs. adversary code (dual-core)



(b) Main computing unit vs. side computing unit (dual-core)



(c) Legitimate code vs. adversary code (quad-core)



(d) Main computing unit vs. side computing unit (quad-core)

Figure 9: Significant and stable timing differences exist between legitimate verification code and tampered verification code as in sub-figures (a) and (c). Such significance and stability also exist in the timing differences between main computing unit and side computing units as in sub-figures (b) and (d). Sub-figures (a) and (c) only show the results for the slowest side computing unit (representing the total execution time), as the results for other individual computing units are very similar.

complete challenge-response protocols, where a 50ms spin-loop is used as the symbolic critical security task for the main computing unit. Both the verifier and the untrusted device measure the elapsed time for obtaining the correct responses using the `rdtsc` instruction. The time measured on the untrusted device for the tampered verification code is the checksum computation time of the theoretical best adversary.

Figure 9 shows the results of our experiments, in which we investigate the timing differences between legitimate verification code and tampered verification code, and between main computing unit and side computing units. The experimental results lead to the following conclusion: a) The theoretical best adversary consumes significantly more time than the detection threshold, which yields zero false negative. b) The variance of response computation time is very low, which can be easily determined from a training process. The impacts from inference among multiple computing units and slight difference in clock rates of different computing units are insignificant for our design. c) The variance of RTT time may cause false positive, which can be avoided by the improved estimation method as suggested in [15]. d) The timing difference between main computing units and side computing units is significant, which prevents the parallel interception attack for the initialization of secure execution environment. Overall, our experiments show that *MT-SRoT is able to provide the same functionalities of software-based root-of-trust on multicore platforms as those provided by the existing timing-based schemes on uniprocessor platforms* [16, 15].

6. CONCLUSION AND FUTURE WORK

Due to the fast advent of computing devices with multicores, it is increasingly important to develop software-based root-of-trust on multicore platforms so as to avoid the patching problem in deploying hardware-based root-of-trust, especially in a large scale. Motivated by this, we investigated the feasibility of constructing software-based root-of-trust on multicore platforms. Our analysis showed that it is easy to use the inherent parallel computing capability on multicore platforms to break the existing software-based root-of-trust schemes designed for uniprocessor platforms. To address these attacks, we presented a timing-based primitive called MT-SRoT to solve these challenges, which is able to ensure untampered execution of a critical security task, such as remote software attestation. Our work is the first step towards software-based root-of-trust on multicore platforms that can be further used as the foundation to establish a tamper-resistant execution environment without the presence of tamper-resistant hardware.

Our work extends software-based root-of-trust from uniprocessor platforms to multiprocessor platforms. Our scheme preserves all the advantages of the existing schemes achieved on uniprocessor platforms, while it also inherits most of their disadvantages. The major disadvantages that need to be addressed are: 1) heavy dependence on a well-controlled measurement environment established under strict assumptions, and 2) vulnerable to certain low-level attacks [15], such as overclock-based attack and DMA-based attack. For these low-level attacks, we notice a recent scheme, *PioneerNG* [12], which is able to defend against all these attacks on

AMD Opteron processors except overclock-based attack. Although it is still difficult to design a general scheme to overcome these disadvantages, the situation will change if corresponding hardware features are available and become standardized. Lack of such hardware support is the major obstacle on the current development of software-based root-of-trust. PioneerNG also provides an alternative design to establish software-based root-of-trust on multiprocessor platform by stopping all unoccupied processors from using the `hlt` instruction. It is different from our scheme which allows unoccupied processors becoming instantly available to other tasks after establishing an isolated execution environment on target processors. Either design can be adopted to fit the hardware architecture of the target platforms.

Since software-based root-of-trust has the attractive properties of flexibility and cost-effectiveness for large scale deployment, it could play an important role in the future. We expect that the future development of software-based root-of-trust will incorporate more hardware features to conquer the challenges which are tangled in a software-only approach.

Acknowledgement: This research is supported by A*STAR SERC Grant No. 102 101 0027.

References

[1] Trusted Computing Group, http://www.trustedcomputinggroup.org.

[2] Year 2000 problem, http://en.wikipedia.org/wiki/Year_2000_problem.

[3] ARM: multicore mobiles coming next year, http://www.multicoreinfo.com/2009/10/multicore-mobiles.

[4] Intel Trusted Execution Technology, http://www.intel.com/technology/security.

[5] Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, May 2005.

[6] Trusted Boot, http://sourceforge.net/projects/tboot.

[7] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 400–409, 2009.

[8] N. T. Courtois, K. Nohl, and S. O'Neil. Algebraic attacks on the crypto-1 stream cipher in mifare classic and oyster cards. In *Cryptology ePrint Archive: Report 2008/166*, 2008.

[9] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, page 407, 2007.

[10] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of the 12th conference on USENIX Security Symposium*, pages 21–21, 2003.

[11] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and A. Seshadri. How low can you go?: recommendations for hardware-supported minimal tcb code execution. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 14–25, 2008.

[12] A. Seshadri. *A Software Primitive for Externally-verifiable Untampered Execution and its Applications to Securing Computing Systems*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 2009.

[13] A. Seshadri, M. Luk, and A. Perrig. Sake: Software attestation for key establishment in sensor networks. In *Distributed Computing in Sensor Systems*, pages 372–385, 2008.

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

[15] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–16, 2005.

[16] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–282, 2004.

[17] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim. Remote software-based attestation for wireless sensors. In *Security and Privacy in Ad-hoc and Sensor Networks*, pages 27–41, 2005.

[18] U. Shankar, M. Chew, and J. D. Tygar. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th conference on USENIX Security Symposium*, pages 7–7, 2004.

[19] R. Wojtczuk and J. Rutkowska. Attacking intel trusted execution technology. In *Black Hat DC*, 2009.

[20] G. Wurster, P. C. v. Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 127–138, 2005.

[21] Y. Yang, X. Wang, S. Zhu, and G. Cao. Distributed software-based attestation for node compromise detection in sensor networks. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, pages 219–230, 2007.