Establishing Software-Only Root of Trust on Embedded Systems: Facts and Fiction

Yanlin Li[1], Yueqiang Cheng[1], Virgil Gligor[1](✉), and Adrian Perrig[2]

[1] CyLab, Carnegie Mellon University, Pittsburgh, PA, USA
gligor@cmu.edu
[2] Computer Science Department, ETH Zurich, Zürich, Switzerland

Abstract. Establishing SoftWare-Only Root of Trust (SWORT) on a system comprises the attestation of the system's malware-free state and loading of an authentic trusted-code image in that state, without allowing exploitable time gaps between the attestation, authenticity measurement, and load operations. In this paper, we present facts and fiction of SWORT protocol design on new embedded-systems architectures, discuss some previously unknown pitfalls of software-based attestation, and propose three new attacks. We describe the implementation of the first attack on a popular embedded-system platform (i.e., on the Gumstix FireStorm COM), establish the feasibility of the second, and argue the practicality of the third. We outline several challenges of attack countermeasures and argue that countermeasures must compose to achieve SWORT protocol security.

1 Introduction

An adversary who can insert malware into a system poses a persistent threat. Malware can survive across repeated boot operations and can be remotely activated at the adversary's discretion. Attempts to detect persistent malware in a system usually require off-line forensic analysis and hence do not offer timely recourse after a successful attack. In contrast, on-line detection of adversary presence in a system can be fast (i.e., a matter of minutes), but typically requires some form of hardware- or software-based attestation by an external device that the system state (e.g., RAM, CPU and I/O registers, device-controller memory) is malware-free. Attestation of malware freedom is particularly important in establishing *SoftWare-Only Root of Trust* (SWORT) since the loading of a trusted-code image in the presence of malware would compromise trust assurances.

When strong guarantees are sought in attestation despite malware presence, designers usually rely on secrets protected in tamper-resistant hardware and standard cryptography; e.g., the private keys of a Trusted Platform Module (TPM) [29]. In contrast, SoftWare-based ATTestation (SWATT) aims to avoid management of secret keys and their protection in hardware. Nevertheless, some SWATT approaches that attempt to provide security assurance still use *some* secrets. For example, early research suggests that, to obtain guarantees of untampered code execution on an adversary-controlled machine, one should use

© Springer International Publishing Switzerland 2015
B. Christianson et al. (Eds.): Security Protocols 2015, LNCS 9379, pp. 50–68, 2015.
DOI: 10.1007/978-3-319-26096-9_7

software agents that encapsulate *hidden* random addresses at which the checksum execution is initiated [7,14]. Other work requires the use of *obfuscated* checksum code [17,24]. More recent research shows that if a remote system can be initialized to a *large enough secret*, malware cannot leak much of the secret before it is changed by a remote verifier, and malware-free code can be loaded on that system [32].

Facts and Fiction. Although tempting, attestation based on secrets is fraught with risk when performed in the presence of a skillful and persistent adversary. Hardware-protected private keys can still be successfully attacked by exploiting compelled/stolen/forged certificates corresponding to these keys [11,19], side channels [30], and padding oracles [3]. Equally important, managing hardware-based attestation (e.g., TPM-based) poses significant usability challenges; e.g., the Cukoo attack [18]. Using a software-protected secret (e.g., checksum obfuscation technique) is particularly dangerous since the secret may be implicitly used for verifying millions of user machines and its discovery may affect many sensitive applications. Annoyingly, secrets can be subpoenaed by oppressive authorities leading to loss of privacy precisely where one needs it most; i.e., in censored computing environments. Hence, attestation based on secrets combines facts, such as the strong cryptographic guarantees available for trusted-image authenticity, with fiction; e.g., long-term protection of secrets can be assured despite advanced persistent threats.

In contrast, traditional SWATT protocols [2,14,20–23] do *not* need secrets for SWORT establishment. These protocols use external system verifiers, which are assumed to be free of malware, to challenge adversary-controlled systems with the execution of checksum functions whose output is verified and execution time is measured. Hence, these protocols must assure that their checksum functions have *accurately measurable* execution-time bounds. Inaccurate verifier measurements would allow an adversary to exploit the time gap between the verifier's expected measurements and the adversary's lower actual execution time.

In practice, however, accurate measurements of checksum execution times are more fiction than fact. For example, to avoid numerous false alarms on realistic system configurations, a verifier's attack-detection threshold must be extended past the average execution-time measurement to account for a checksum's execution-time jitter. This includes clock variation due to static skew and dynamic (e.g., peak-to-peak) jitter, each of which can easily extend a processor's clock period by 3–8 % [27], and execution-time jitter due to hard-to-predict Translation Lookaside Buffer (TLB) and cache behavior. Since threshold extensions can lead to successful attacks, previous SWATT approaches set low values for them; e.g., less than 1.7 % over the average execution time in the no-attack mode of operation in [14,22]. However, low values will cause false-positive malware detection on new embedded-system platforms, which SWATT aims to avoid[1]. Extending the threshold

[1] Repeating the SWATT protocol only a half a dozen times to identify and disregard false positives would be unrealistic for embedded-system platforms such as the Gumstix FireStorm Com where a single checksum execution takes about thirteen minutes; viz., Sect. 4.1.

to only 3 % over the average execution time to avoid false positives would certainly introduce added vulnerabilities (viz., Sect. 3.2), and thus SWATT would have to counter new attacks on these platforms.

Furthermore, modern embedded platforms have became increasingly complex and diverse. Thus the belief that a traditional checksum designs (reviewed in Sect. 6), perhaps with provable properties [2], will suffice for different architectures and scale is based on more fiction than fact.

An additional fact, which is sometimes ignored by past research, is that SWATT must be uninterruptably linked to other functions to be useful in SWORT. Otherwise, SWORT becomes pure fiction: an adversary could pre-plan unpleasant surprises for a verifier *after* successful SWATT and *before* system boot. For example, the adversary could violate the authenticity of the trusted image by returning a correct image-integrity measurement and yet load an adversary-modified image in a malware-free state; viz., Sect. 3.1. This type of attack is enabled by hardware features on modern embedded platforms that enable attackers to create a time gap between SWATT and authenticity measurement of the loaded image.

Contributions. In this paper, we show that the new embedded system architectures pose significant challenges to software-only root of trust (SWORT) protocols by enabling new attacks launched primarily against software-based attestation (SWATT). In particular, we make the following three contributions.

1. We present new architecture features of embedded system platforms that pose heretofore unexpected challenges to traditional SWATT protocols and their use in SWORT.
2. We define three new attacks against SWORT protocols that are enabled by both new architecture features (e.g., future-posted events, L1 caches relying on software-only coherence mechanisms) and scalability considerations (e.g., jitter caused by caches during randomized memory walks, clock jitter) on embedded platforms.
3. We present the implementation of the first attack on a popular embedded-system platform (i.e., on the Gumstix FireStorm COM), establish the feasibility of the second, and argue the practicality of the third. We outline several challenges of attack countermeasures, find dependencies among them, and argue that these must compose to achieve SWORT protocol security.

Organization. The remainder of this paper is organized as follows: Sect. 2 provides a brief overview of typical SWORT protocols and known attacks against SWATT designs. Three new attacks against SWORT protocols on modern embedded platforms are described in Sect. 3. Section 4 establishes the feasibility of these attacks, and Sect. 5 illustrates the challenges of effective countermeasures for these attacks. We describe related work in Sect. 6, and present our conclusions in Sect. 7.

2 Software-Only Root of Trust

A SoftWare-Only Root of Trust (SWORT) protocol comprises three distinct steps: (1) the establishment of an untampered execution environment (aka. a malware-free system state) via SoftWare-Based ATTestation (SWATT), (2) the verification of trusted-image authenticity, and (3) the loading of the authentic image onto the malware-free state, without allowing exploitable *time gaps* between the three steps.

2.1 Architecture and Protocol

In SWORT, a verifier program runs on a *trusted* machine and performs SWATT on a prover device using a nonce-response protocol; viz., Steps 1 – 2 of Fig. 1. The verifier program comprises a checksum simulator, a timer, and a cryptographic hash function. The checksum simulator generates pseudo-random *nonces* for attestation, constructs a copy of the device memory contents, and computes the expected response (i.e., checksum result) by simulating the checksum computation on the prover device. The timer measures the elapsed time of the nonce-response reception. An authentic, trusted code image (i.e., a correct, malware-free device image) is available on the verifier machine, and the hash function is used to compute its digest for the integrity measurement of the *device image*.

On the prover device, a prover program is installed and includes a checksum function, a communication function, a hash function, and other functions, such as those of the boot code. The checksum function disables interrupts on the device, resets system configurations to a known state, computes a checksum over the prover program and other critical memory contents (e.g., page table, stack, exception handler table, and communication buffer), and establishes an untampered execution

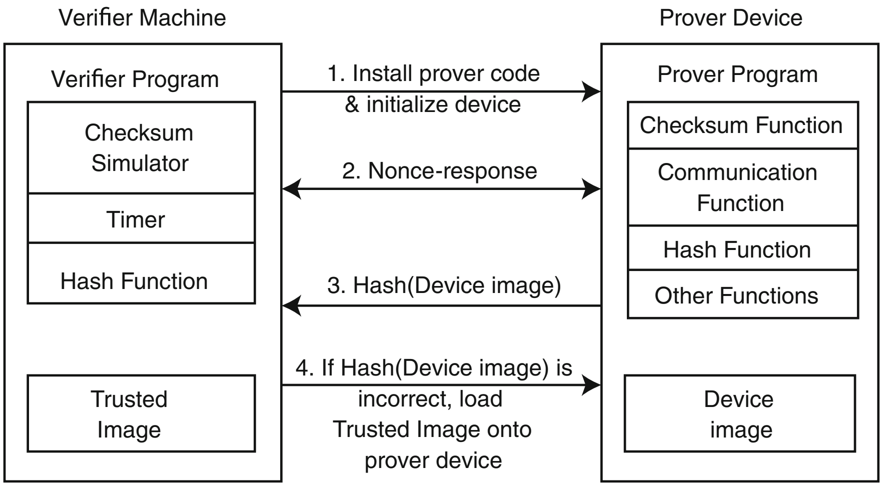

Fig. 1. SWORT system architecture and verification protocol.

environment for the device image. The checksum function must be designed such that modifications of its code or additions of malware instruction would invalidate the checksum result or cause a detectable computation overhead.

The verifier program checks the validity of the prover's response (i.e., the checksum result) and the elapsed time of the nonce-response pair. If the checksum result is correct and the measured time is within a detection threshold, the verifier program obtains the guarantee that an untampered execution environment has been established on the prover device, and subsequent results sent by the prover device are obtained in a malware-free state and are trusted. Please note that SWORT assumes that attackers cannot physically change the hardware configuration of the prover device, such as adding additional memory, replacing the device's CPU with a faster CPU or over-clocking the device's CPU frequency. In addition, the SWATT component (i.e., Steps 1 – 2 of Fig. 1) of the SWORT protocol *assumes* that attackers cannot optimize the checksum function to run it faster or find an alternative algorithm to compute the result faster.

After sending the checksum result to the verifier program, the checksum function calls a hash function to compute an integrity measurement (i.e., cryptographic hash) of the entire device image and sends the measurement to the verifier program. The verifier program checks the integrity of the device image, and if the device image has been changed, the verifier program loads the authentic, trusted image onto the prover device. Subsequent loading of this image in the malware-free memory state establishes the root of trust on the prover device[2].

In SWATT, the checksum function can also fill the spare memory space with pseudo-random values and then compute a checksum over the entire memory contents (instead of only over the prover program and other critical memory contents). In this way, the verifier program can prevent attackers from using the spare memory space on the prover device to perform malicious operations.

2.2 Known Attacks Against SWATT

In the absence of a concrete formal analysis method to evaluate SWATT designs on modern commodity systems (viz., Sect. 6), it is instructive to review the known attacks that must be countered by practical protocols. Several classes of attacks against the SWATT protocol (i.e., Steps 1 – 2 of Fig. 1) have been proposed in the past; i.e., the memory-substitution (aka. memory-copy) attacks [22], proxy attacks [22], split-TLB attacks [31], memory-compression attacks [4], and return-oriented programming attacks [4].

In a *memory-substitution attack*, an adversary runs a modified checksum function in the correct location and saves a correct copy of the original checksum function in spare memory. During the checksum computation, the modified checksum function checks every memory address to read and redirects the memory read to the correct copy when the modified memory contents are read. Two

[2] We assume that other SWATT techniques, such as the ones in VIPER [16] are employed to assure malware-free state of I/O device controllers, including NICs, GPUs, and disk, keyboard, and printer controllers.

types of such attacks have been identified in the past. In the first type, attackers run a modified checksum function at the correct memory location, and save a correct copy of the original checksum function in spare memory space. During the checksum computation, the malicious checksum function computes the checksum over the correct copy. In the second type, attackers load the original checksum function to the correct memory location, but run a malicious checksum function in another memory location that computes the checksum over the original copy. To defend against these attacks, the Program Counter (PC) and Data Pointer (DP) values (the memory address to read) are incorporated into the checksum computation. Thus, the malicious checksum function has to forge both the correct PC and DP values to compute the expected checksum, thereby causing a computation overhead.

In a *proxy attack*, the prover device asks a remote faster computer (a proxy) to compute the expected checksum. The proxy attack can be detected if the user monitors all the communication channels of the prover device. For example, the user can use a Radio Frequency analyzer (e.g., RF-Analyzer HF35C) to detect any wireless communications of the prover device, thus detecting wireless proxy attacks.

In a *split-TLB attack*, an adversary configures the Instruction Translation Lookaside Buffer (I-TLB) and the Data Translation Lookaside Buffer (D-TLB) such that the entries for the checksum function memory pages point to different physical addresses in the I-TLB and the D-TLB. Thus, the adversary can execute a malicious checksum function, but compute the checksum over the correct copy of the checksum code. However, the adversary must guarantee that the carefully configured entries in the D-TLB and the I-TLB are preserved during the checksum computation.

When the checksum function fills the spare memory space with pseudorandom values and computes a checksum over the entire memory contents, attackers cannot find the free memory space for a memory-copy or memory-substitution attack or a split-TLB attack. In this case, attackers might be able to perform a *memory-compression attack* whereby a malicious checksum compresses the memory contents of the prover program (to get spare memory space), and then decompress the compressed content to get the expected value when the compressed memory content is checksumed.

In a *return-oriented programming attack*, an adversary modifies the stack contents to break the control flow integrity of the prover program. Because the stack contents are incorporated into the checksum, the adversary has to perform additional operations (e.g., a memory-copy attack) to compute the expected checksum result.

3 New Attacks Against the SWORT Protocol

New attacks against the SWORT protocol include those that create and exploit timing gaps between correct execution of SWATT and subsequent integrity measurements; i.e., between the correct completion of Step 2 and the execution of

Step 3 in Fig. 1. New attacks also include those that exploit vulnerabilities of traditional SWATT introduced by new architecture features of embedded-system platforms. The balance of this section illustrates both types of new attacks.

3.1 Future-Posted Event Attacks

Some modern embedded-system platforms allow the configuration of *future-posted* events. These events can be set during system configuration (e.g., during Step 1 of Fig. 1) and trigger at a future time when the system runs (e.g., after Step 2 of Fig. 1). Leveraging the *future-posted* events, attackers can create a timing gap between the SWATT steps and subsequent integrity measurements. An example of such an event is the *future-posted Watch-Dog Timer (WDT) reset*. Other examples include the future-posted DMA transfers.

On some embedded platforms, attackers can configure the WDT to reset the device after a specific timer period. For example, on TI DM3730-based platforms, the CPU [28] supports the future-posted WDT reset, and the specific time period to reset the device can be configured as between about 62.5 μs and 74 h and 56 min. As a result, attackers can perform *future-posted WDT reset attacks*.

Figure 2 shows the timeline of this attack. Suppose that malware controls the platform during the installation of the prover code (i.e., Step 1 of Fig. 1) and configures the WDT to reset the device after the *correct checksum result* is sent to the verifier program; i.e., after Step 2 of Fig. 1. Then the malware erases itself from memory and invokes the prover program. During the SWATT steps, the prover program calls the checksum function to compute a checksum over the memory contents based on the nonce from the verifier program, and then sends the correct checksum result to the verifier program. After the checksum result is sent, the WDT reset event is triggered and the platform boots from an adversary-modified device image. After reboot, the malware of the device image controls the platform and sends a forged hash result (i.e., integrity measurement of the device image) to the verifier program.

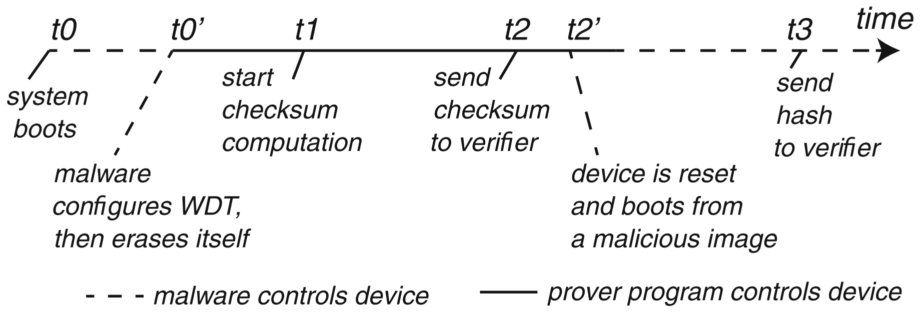
t0
t0'
t1
t2
t2'
t3
time
system boots
malware configures WDT, then erases itself
start checksum computation
send checksum to verifier
device is reset and boots from a malicious image
send hash to verifier
- - - malware controls device
—— prover program controls device

Fig. 2. Timeline of a WDT reset attack.

3.2 Attacks Exploiting High Execution-Time Variance

In SWATT protocols, the measured time of one nonce-response round is utilized to detect malicious operations on the prover device; e.g., malicious operations to forge correct checksum results. Typically the attack-detection threshold is set based on the overhead caused by possible attacks. If a measured time exceeds the threshold, it is highly likely that malicious operations are performed on the prover device. However, the measured time may exhibit significant variance caused by CPU clock variance, Translation Look-aside Buffer (TLB) misses, and possibly cache misses. Hence, to avoid false-positive malware detection, the attack-detection threshold must be extended over the maximal value of the measured time in normal (no-attack) conditions.

Recent research [27] shows that the modern CPU clock variance can be up to 3–8% and it increases with program execution times. Furthermore, because the traditional checksum functions read the memory contents in a pseudo-random pattern, the resulting TLB misses could increase the measured execution time significantly. To account for these types of execution-time jitter, the maximal value of the measured time in normal (no-attack) conditions must be extended significantly; e.g., by nearly 3 % of the average execution time.

Previous software-based attestation schemes did not have to account for *high* execution-time variance in setting the maximal execution time threshold. Their

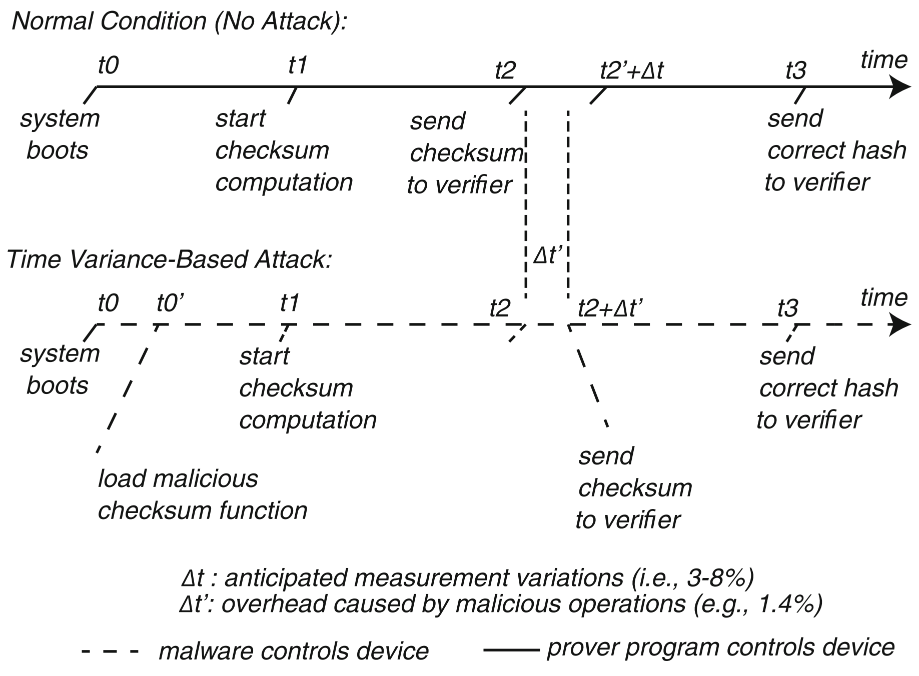
Normal Condition (No Attack):
t0
t1
t2
t2'+Δt
t3
time
system boots
start checksum computation
send checksum to verifier
send correct hash to verifier
Δt'
Time Variance-Based Attack:
t0
t0'
t1
t2
t2+Δt'
t3
time
system boots
start checksum computation
send correct hash to verifier
load malicious checksum function
send checksum to verifier
Δt : anticipated measurement variations (i.e., 3-8%)
Δt': overhead caused by malicious operations (e.g., 1.4%)
- - - malware controls device
—— prover program controls device

Fig. 3. Timeline of normal condition and time-variance based attack.

designs made cache behavior fairly predictable; e.g., the checksums fit into the cache and random access patterns resulted in predictably high overheard. TLBs need not be used since measurements were taken in physical RAM, and clock jitter was small because checksum execution times were short for relatively small memory configurations. In contrast, some of the new embedded-system processors force virtual memory (and hence the TLB) use whenever caches are used, and large memory configurations (i.e., GB size) cause checksum functions to execute for minutes instead of tens of milliseconds. Consequently, attackers can now exploit the high execution-time jitter on embedded systems to launch successful time-variance based attacks with *non-negligible probability*.

Timeline. Figure 3 shows the timeline of a *time-variance based attack*. Here, malware that controls the platform loads a modified checksum function that computes the expected checksum result. To protect the modified contents (i.e., malicious code) from being detected, the modified checksum function performs additional operations to forge the expected checksum, and these operations cause an overhead $\Delta t'$. However, as shown in the figure, under normal (no-attack) conditions, the anticipated measured time variation is Δt (i.e., the timing detection threshold), is larger than $\Delta t'$. Consequently, the verifier receives the correct checksum result within the timing detection threshold, and hence the verifier cannot detect this attack; i.e., a false-negative detection result.

3.3 Attacks Exploiting I-cache Inconsistency

Modern embedded processors have multiple-level caches. However, to save energy, some embedded processors may not have hardware support for cache coherence between Instruction-cache (I-cache) and Data-cache (D-cache), and software has to maintain cache coherence. For example, the ARM Cortex-A8 processor, which is widely deployed on embedded platforms, does not have hardware support for cache coherence between I-cache and D-cache. Software has to use cache maintenance instructions to ensure cache coherence. Therefore, the contents of the I-cache may differ from those of the D-cache, and attackers can leverage this feature to hide malicious instructions (e.g., malicious instructions in the communication function or hash function) in the I-cache without being detected. We call this attack the *I-cache inconsistency attack*.

This attack is similar in spirit to the Split-TLB attacks (Sect. 2.2), where the I-TLB and D-TLB contain inconsistent mappings for the checksum function pages. Experience with those attacks suggests that the *I-cache inconsistency attack* is equally practical, particularly since its setup is simpler.

Timeline. The timeline of an I-cache inconsistency-based attack is shown in Fig. 4. Here, malware first loads malicious instructions into the I-cache, then overwrites the malicious content in memory with the original values to guarantee that only legitimate contents are in the memory during checksum computation. The malicious code needs to comprise only a few instructions of the hash

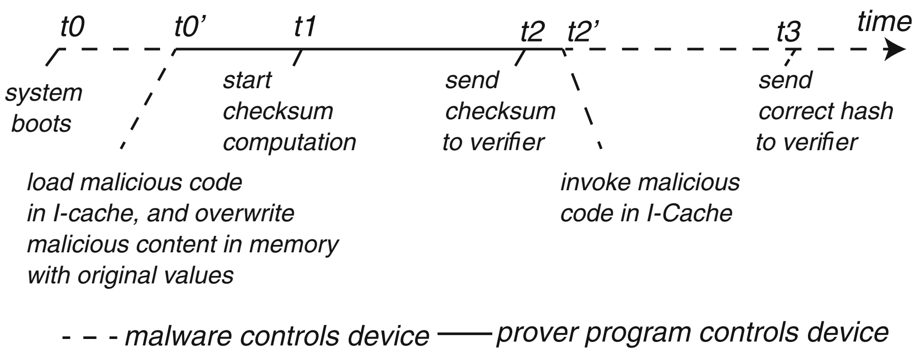
t0
t0'
t1
t2
t2'
t3
time
system boots
start checksum computation
send checksum to verifier
send correct hash to verifier
load malicious code in I-cache, and overwrite malicious content in memory with original values
invoke malicious code in I-Cache
- - - malware controls device
—— prover program controls device

Fig. 4. Timeline of an *I-cache inconsistency attack.*

or communication function. The checksum function is computed over the legitimate memory contents and the correct checksum result is sent to the verifier program. After the checksum result is sent to the verifier program, the malicious instructions in the I-cache are invoked, and then the adversary controls the system.

4 Checksum and Attack Implementation

We implemented our SWATT protocol checksum and attacks based on future-posted events and execution-time variance using a Gumstix FireStorm COM[3] as the prover device and a HP laptop as the verifier machine. We leave the implementation of the *I-cache inconsistency* attack to future work.

The Gumstix FireStorm COM is a TI DM3730-based platform [28] with 512 MB SDRAM, 64 KB SRAM, and 512 MB NAND Flash. The TI DM3730 Central Processing Unit (CPU) is equipped with an ARM Cortex-A8 core (MPU) [1] running at 1-GHz. The ARM Cortex-A8 core has two levels of caches without hardware-support for cache coherence in the first-level caches. The first level (L1) has a 32 KB I-cache and a 32 KB D-cache while the second level (L2) has a 256 KB unified cache. The Gumstix COM runs a Linux operating system (based on Yocto Poky Dylan 9.0.0[4]) with the Linux kernel version 3.5.7. The HP laptop is connected to the Gumstix COM directly via an Ethernet cable. The laptop has an Intel Quad-Core i5 CPU running at 2534 MHz, 4 GB of RAM, and runs 32-bit Ubuntu 12.04 LTS as the guest OS.

We implemented the prover program as a loadable kernel module, which includes the Ethernet communication function, checksum function, and SHA256 hash function. The checksum implementation for the ARM Cortex-A8 core is based on the checksum function of PRISM [6]. The prover program uses an AES-based Pseudo-Random Number Generator. Using part of the pseudo-random nonce sent by the verifier machine as the seed, the prover program generates

[3] https://store.gumstix.com/.
[4] https://www.yoctoproject.org/.

and fills all spare memory space on the Gumstix COM with pseudo-random values before the checksum computation. On the verifier machine, we run a verifier program that measures the time of one nonce-response round using the *RDTSC* instruction.

In this section, we first describe our checksum function implementation in detail, and then present the implementation of attacks exploiting the Watch-Dog Timer and the execution-time variance of our checksum.

4.1 Checksum Function

Checksum Implementation. We implement the checksum function as 32 code blocks where each code block updates a 32-bit checksum-state variable (out of a total 32 checksum-state variables) comprising 33 ARM instructions that strongly order *AND*, *XOR*, and *SHIFT* operations. Each code block takes as input: the other checksum-state variables, the memory address being read (Data Pointer), memory contents, current processor status (i.e., the Current Processor Status Register (CPSR) value), Program Counter (PC), the pseudo-random numbers generated by a T-function [12], and a counter. The strong ordering of the checksum instructions guarantees that they cannot be executed in parallel.

In the checksum function, the 32-bit T-function is used to build a Pseudo-Random Number Generator (PRNG) to construct the (random) memory addresses used by the *read* instructions. Each code block performs two memory *reads*: one from the 512 MB SDRAM and the other from the 64 KB SRAM. Multiple iterations of checksum code-block executions cover the entire memory content of our system. The checksum function uses all 30 available ARM General Purpose Registers (GPRs) (nb., 2 GPRs in monitor mode are not available to access on ARM Cortex A8) as follows: 25 GPRs are used to save checksum states; r0 stores the pseudo-random value from T-function; r1 and r2 are used as temporary variables; r3 stores the counter value; $r12$ stores the current checksum state. (We also use available Save Processor Status Registers [1] to save checksum states.) Fig. 5 shows the assembly code of a single checksum block.

Checksum Execution Time. To include every randomly picked memory location in the checksum computation at least once with high probability, we set the checksum code-block iteration number (i.e., the number of checksum code blocks that execute on the Gumstix COM) to $0xb0000000$ based on the Coupon Collector's Problem [5]. Our verifier measurements show that a single nonce-response round (i.e., Step 2 of Fig. 1) takes 765.4 s on average with 2.9 s standard deviation over 371 measurements under normal (no-attack) condition. The maximal execution time of the nonce-response round is 768.1 s while the minimum value is 745.0 s. Seven measurements out of the 371 measurements take less than 750 s while the other 364 measurements take between 765 to 769 s. Thus, the measured time variance of the seven measurements is about 2.6 %.

To get consistent timing results using the *RDTSC* instruction, we configure the HP laptop (the verifier machine) running at a constant CPU frequency (2534

$r0$	Pseudo-Random Number (PRN) in T-function
$r1$	tmp and memory address to read
$r2$	tmp
$r3$	counter
$r4$ *to* $r14$	checksum states
C	Carry flag
Assembly Instruction	Explanation
umull r2, r1, r0, r0	$tmp = PRN \times PRN$, T-function computation
orr r1, r2, #0x5	$tmp = tmp \mid 5$
add r0, r0, r1	$PRN = PRN + tmp$
and r2, r0, 0x1FFFFFFC	$offset = PRN \ \& \ mask$
adds r1, r2, 0x80000000	$addr\_sdram = base\_addr + offset,\ update\ C$
ldr r2, [r1]	$tmp = mem[addr\_sdram]$
adcs r12, r12, r2	$r12 = r12 + tmp + C,\ update\ C$
eor r12, r12, r13	$r12 = r12 \oplus r13$
adcs r12, r12, r15	$r12 = r12 + PC + C,\ update\ C$
eor r12, r12, r14	$r12 = r12 \oplus r14$
adcs r12, r12, r4	$r12 = r12 + r4 + C,\ update\ C$
eor r12, r12, r0	$r12 = r12 \oplus PRN$
adcs r12, r12, r5	$r12 = r12 + r5 + C,\ update\ C$
eor r12, r12, r6	$r12 = r12 \oplus r6$
adcs r12, r12, r1	$r12 = r12 + addr\_sdram + C,\ update\ C$
mrs r2, spsr	$tmp = SPSR$
eor r12, r12, r2	$r12 = r12 \oplus tmp$
adcs r2, r12, r0	$tmp = r12 + PRN + C,\ update\ C$
movt r2, #0x4020	$tmp = (tmp \ \& \ 0xFFFF) \mid 0x40200000$
and r1, r2, #0xFFFFFFFC	$addr\_sram = tmp \ \& \ mask$
ldr r2, [r1]	$tmp = mem[addr\_sram]$
adcs r12, r12, r2	$r12 = r12 + tmp + C,\ update\ C$
eor r12, r12, r7	$r12 = r12 \oplus r7$
adcs r12, r12, r8	$r12 = r12 + r8 + C,\ update\ C$
eor r12, r12, r1	$r12 = r12 \oplus addr\_sram$
adcs r12, r12, r9	$r12 = r12 + r9 + C,\ update\ C$
eor r12, r12, r10	$r12 = r12 \oplus r10$
adcs r12, r12, r11	$r12 = r12 + r11 + C,\ update\ C$
mrs r2, cpsr	$tmp = CPSR$
eor r12, r12, r2	$r12 = r12 \oplus tmp$
adcs r12, r3, r12, ROR #1	$tmp = rotation\_shift\_right\_1\_bit(r12)$ $r12 = tmp + counter + C,\ update\ C$
adcs r0, r0, r12	$PRN = PRN + r12 + C,\ update\ C$
sub r3, r3, #1	$counter = counter - 1$

Fig. 5. Assembly instructions for a single checksum block.

MHz); we also configure the ARM Cortex-A8 processor on the Gumstix COM running at its maximal CPU frequency (1 GHz).

4.2 WDT Reset Attack Implementation

The implementation of the WDT reset attack against our prover code comprises two critical steps: the first is the setting WDT timer to reset the device, and the second is the generation of the correct hash result and its sending to the verifier program when the system boots up.

Before setting the WDT timer, we must wake it up. Specifically, we have a dedicated kernel module *malmod*, which is installed exactly before the attestation kernel module. The *malmod* sets *EN_WDT2* bit (i.e., Bit 5) in the *CM_FCLKEN_WKUP* register and *EN_WDT2* bit (i.e., Bit 5) in the *CM_ICLKEN_WKUP* register to wake up the WDT timer. A WDT timer has a large reset period, from 62.5 µs to 74 h 56 min, and the time of one nonce-response round under normal (no-attack) condition (i.e., 765.4 s) is within the reset period. In addition, the SHA256 hash computation (i.e., the interval $t3 - t2$ in Fig. 2) takes about 76 s. The *malmod* configures the WDT to reset the device between times *t2* and *t3* on the time line of Fig. 2, and then starts the WDT by writing its start/stop register (WSPR) twice using the sequence *0xAAAA* followed by *0x5555* [28]. Before checksum computation starts, the prover kernel module replaces the memory contents of the *malmod* with the required pseudo-random values.

After the correct checksum result is sent to the verifier machine, the Gumstix COM reboots from an attacker-modified device image. In the modified device image, we place a script file in the */etc/init.d/* directory, which automatically executes a malicious program after reboot. In our implementation, the prover kernel module and the verifier program communicate via ICMP packets. The malicious program generates the expected ICMP response message including the correct hash value, and sends the ICMP message to the verifier program.

4.3 Feasibility of the Time-Variance Based Attack

We evaluated the feasibility of time-variance based attacks by measuring the overhead of a *memory-substitution* attack (viz., Sect. 2.2) in which an adversary saves a correct copy of the prover code on the NAND Flash. Then the attacker-modified checksum verifies every randomly generated memory address to be used by the *read* instructions and redirects the *reads* to the original copy on the NAND Flash whenever they attempt to use the attacker-modified checksum contents.

In this attack, the attacker-modified checksum function reads the entire SDRAM (512 MB) in a pseudo-random pattern. Note that the attacker-modified SDRAM content (i.e., the modified checksum code blocks) represents only a small fraction; i.e., about 4 KB or 0.00076 %[5]. Consequently, the main overhead of this attack is caused by the instructions that verify the pseudo-random

[5] $\frac{4}{512 \times 1024} = 0.0000076$.

SDRAM memory address and redirect the *reads* when necessary. In the evaluation, we ignore the small and relatively rare overhead caused by the actual reading the NAND Flash, and evaluate the frequent overhead caused by the additional instructions that check the randomly generated SDRAM address.

We added six instructions in each modified checksum block to verify every pseudo-random SDRAM memory address used by the *read* instructions; i.e., one instruction to jump out of the modified checksum code block to avoid changing the PC value incorporated into the checksum; four instructions to check if the memory address is within the modified memory space and to restore the previous Current Processor Status Register (CPSR) value; and one instruction to jump back to the checksum block. We measured the time of one nonce-response round of the SWATT protocol (i.e., Step 2 of Fig. 1) using the modified checksum function on the Gumstix COM. The measurement results show that the overhead caused by the six additional instructions is only 1.4 %[6], which is much smaller than the time variance under the normal (no-attack) condition; i.e., 2.6 %, as shown in Sect. 4.1. Consequently, this instance of a memory substitution attack will succeed with non-negligible probability.

5 Challenges for Effective Countermeasures

Challenges. To counter attacks exploiting *future-posted events*, the checksum function needs to capture the values of *all* critical configuration registers in the checksum result. Thus the setting of future-posted events would cause an incorrect checksum result. This countermeasure is less straight-forward than it might first appear, for two reasons. First, modern embedded platforms have a large numbers of configuration registers with complex configuration options; e.g., there are thousands of I/O device control registers in the Gumstix ARM Cortex-A8 platform that must be analyzed. Thus, finding *all* critical configuration options that may enable such attacks becomes a non-trivial exercise. Second, the assurance of a correct setting that disables future-posted WDT events depends on effective countermeasures to attacks that exploit high execution-time variance and I-cache inconsistency (discussed below).

To counter attacks that exploit *I-cache inconsistency*, a possible approach is to design a checksum function whose size is larger than the I-cache. Thus during checksum execution, the code blocks in the I-cache will be evicted and replaced. However, attackers may be able to compress the checksum code blocks (e.g., by removing duplicate instructions), add malicious instructions and run the compressed checksum code blocks in the I-cache instead of the original, larger blocks. It is extremely challenging to guarantee that a checksum function cannot be compressed to fit into the I-cache. For example, one might include the cache-miss counter, which is used by the system performance monitor, to the checksum computation [10], in an attempt to prevent attackers from running

[6] The primary reason the overheard added by the six instructions is so small is that the instruction which *reads* from a pseudo-random memory address in every code block consumes many more CPU cycles than six instructions.

a smaller checksum function that will change the cache-miss counter. However, this countermeasure would require that the cache replacement policy be deterministic. Otherwise, the verifier program could not predict and verify the checksum results. Unfortunately, on the TI DM3730 platform, the ARM Cortex-A8 processor utilizes a random-replacement policy for the I-cache.

Yet another possible approach to counter attacks that exploit *I-cache inconsistency* is to utilize dynamically modified instructions in the checksum code blocks. Then, one could insert instructions into these code blocks that invalidate all I-cache blocks. For example, the ARM Cortex-A8 instruction set includes an instruction that invalidates all I-cache blocks. However, attackers may perform *read-decode-execute* operations to handle dynamically modified checksum instructions and avoid the invalidation of the I-cache blocks. That is, in a *read-decode-execute* operation, adversary-modified (i.e., malicious) checksum code in the I-cache can read the dynamically modified instructions, decode them, and execute code based on the decoded information. Thus, the modified (i.e., malicious) checksum function can avoid the instruction that invalidates I-cache blocks. Obtaining demonstrable countermeasures for attacks that exploit *I-cache inconsistency* is extra challenging because they depend on other countermeasures; i.e., on those for the time-variance based attacks (discussed below).

To counter attacks that exploit high *execution-time variance*, one could modify the checksum design to force attackers to perform complex operations and cause an unavoidable increase of execution overhead. A possible approach would be to add performance-monitor values (e.g., the executed instruction counter, the TLB-miss counter) in the checksum computation. Unfortunately, this might not increase the attack overhead sufficiently for detection. For example, attackers may need only several additional instructions to calculate the correct instruction counter. In addition, attackers may run the malicious code in the same page with the checksum function, thereby avoiding changes to the TLB-miss counter. The I-cache-miss counter may force attackers to perform a large number of operations to obtain the expected value; e.g., by simulating the I-cache replacement. However, this also requires that the I-cache replacement policy be deterministic, which is not the case for the ARM Cortex-A8.

Effectiveness. To be effective, a countermeasure for a given attack must not only deny the attacker' goal but must also compose with countermeasure for other possible attacks; e.g., a countermeasure can be effective for an attack only if other countermeasures are effective against other attacks [8]. Composition requires that all dependencies between countermeasures be found and cyclic dependencies removed [9].

Countermeasure dependencies exist for our attacks. For example, the countermeasures for the future-posted WDT reset attack depend on the countermeasures for attacks that exploit high execution-time variance and I-cache inconsistency. Unless these two attacks are countered effectively, the adversary can modify the checksum code to erase the instructions that disable future-posted WDT events. Furthermore, a dependency exists between the countermeasures for the I-cache inconsistency attack and that for time-variance attack. The former depends on the latter, since the time-variance attack can be used to modify the checksum code so that the I-cache resetting instructions are erased.

6 Related Work

Reflection [26] is a software-only approach to verify program code running on an untrusted system. *Reflection* fills the spare memory with pseudo-random content, resets the system state, and computes a hash over the entire memory. A verifier machine checks the hash results and the execution time. Genuinity [10] validates the system configuration of a remote machine using software-based mechanisms. Genuinity reads memory in a random pattern to cause a large number of TLB misses, and incorporates the number of TLB misses in the checksum result, preventing attacks based on the observation that simulating the hardware operations (e.g., TLB block replacement) is slower than the actual execution. However, Genuinity is vulnerable to memory-copy attacks [25]; viz., Sect. 2.2.

SWATT [23] performs software-based attestation for embedded systems. In SWATT, a checksum function computes a checksum over the entire memory contents using strongly-ordered *AND*, *ADD*, and *XOR* operations. An external verifier checks the checksum results and the execution time. The main idea was that malicious operations would either invalidate the checksum results or cause detectable timing delays, or both. However, this protocol does not aim to counter attacks caused by future-posted events, cache (in)coherence, or high execution-time variance; e.g., CPU clock variance, cache- and TLB-caused jitter.

Pioneer [22] establishes an untampered execution environment over a small piece of memory on an AMD Opteron K8 architecture platform. The Pioneer checksum disables all interrupts, and computes its result over a small piece of memory. Naturally, the Pioneer design could not anticipate the future hardware features of embedded platforms that we address in this paper. PRISM [6] used software based attestation to establish untampered code execution on embedded ARM platforms. However, PRISM does not address the possible challenges of cache (in)coherence and high execution-time variance presented in this paper.

SBAP [15] verifies the firmware integrity of an Apple aluminum keyboard that has limited resources using software-only approaches. VIPER [16] verifies the integrity of peripherals' firmware on commodity systems using software-based attestation mechanisms. VIPER describes the possible mechanisms to prevent proxy attacks in software-based attestation protocols. Like Pioneer, the VIPER design could not anticipate the impact of future hardware features on software-based attestation. Kovath *et al.* [14] propose a comprehensive timing-based attestation system that verifies the system integrity of machines in enterprise environments. It successfully detects attacks that have only 1.7 % overhead. However, the low detection threshold may cause a significant number false-positive detection of malware because of the CPU clock variance, which can reach 3 % of execution time on some embedded-system platforms.

Armknecht *et al.* [2] propose an abstract security model for design and analysis of software-based attestation protocols. However, this model does not aim to address SWORT protocols nor to offer *concrete* checksum designs that could counter the the attacks of new hardware features described in this paper.

7 Conclusions

Embedded system platforms are used pervasively in security-sensitive applications, and consequently are becoming attractive targets of attack [13]. Yet, existing software-based attestation protocols designed for such applications cannot address the newly introduced complex hardware features that enable new attacks. As a result, attackers can leverage these features to break the security of SWORT protocols. This paper presents new attacks whose countermeasures appear to require a significant redesign of traditional software-based attestation protocols. In particular, we find dependencies among countermeasures, which show that countermeasures must compose to achieve SWORT protocol security.

Acknowledgements. We are grateful to David Brumely, Tom Forest, Di Jin, and Maverick Woo for their comments and suggestions on the research reported herein. This work was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract N66001-13-2-404 and by a grant from the General Motors (GM) Corporation at CyLab, Carnegie Mellon University. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of CMU, GM, DARPA, or the U.S. Government or any of its agencies.

References

1. ARM. Cortex-A8 technical reference manual. Revision:r3p2, May 2010
2. Armknecht, F., Sadeghi, A.-R., Schulz, S., Wachsmann, C.: A security framework for the analysis and design of software attestation. In: Proceedings of ACM Conference on Computer and Communications Security, pp. 1–12 (2013)
3. Bardou, R., Focardi, R., Kawamoto, Y., Simionato, L., Steel, G., Tsay, J.-K.: Efficient padding oracle attacks on cryptographic hardware. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 608–625. Springer, Heidelberg (2012)
4. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the difficulty of software-based attestation of embedded devices. In: Proceedings of the ACM Conference on Computer and Communications Security, November 2009
5. Erdos, P., Renyi, A.: On a classical problem of probability theory. In: Proceedings of Magyar Tudomanyos Akademia Matematikai Kutato Intezetenek Kozlemenyei, pp. 215–220 (1961)
6. Franklin, J., Luk, M., Seshadri, A., Perrig, A.: Prism: enabling personal verification of code integrity, untampered execution, and trusted I/O or human-verifiable code execution. CyLab Lab Technical report CMU-CyLab-07-010, Carnegie Mellon University (2007)
7. Garay, J.A., Huelsbergen, L.: Software integrity protection using timed executable agents. In: Proceedings of ACM Symposium on Information, Computer and Communications Security, pp. 189–200 (2006)

8. Gligor, V.: Dancing with the adversary: a tale of wimps and giants. In: Christianson, B., Malcolm, J., Matyáš, V., Švenda, P., Stajano, F., Anderson, J. (eds.) Security Protocols 2014. LNCS, vol. 8809, pp. 100–115. Springer, Heidelberg (2014)
9. Kailar, R., Gligor, V., Gong, L.: Effectiveness analysis of cryptographic protocols. In: Proceedings of IFIP Conference on Distributed Computing for Critical Applications. Springer, January 1994
10. Kennell, R., Jamieson, L.H.: Establishing the genuinity of remote computer systems. In: Proceedings of the USENIX Security Symposium, pp. 295–308 (2003)
11. Kim, T.H.-J., Huang, L.-S., Perrig, A., Jackson, C., Gligor, V.: Accountable Key Infrastructure (AKI): a proposal for a public-key validation infrastructure. In: Proceedings of International World Wide Web Conference (WWW) (2013)
12. Klimov, A., Shamir, A.: A new class of invertible mappings. In: Kaliski, B.S., Koç, K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523. Springer, Heidelberg (2002)
13. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S.: Experimental security analysis of a modern automobile. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 447–462 (2010)
14. Kovah, X., Kallenberg, C., Weathers, C., Herzog, A., Albin, M., Butterworth, J.: New results for timing-based attestation. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 239–253 (2012)
15. Li, Y., McCune, J.M., Perrig, A.: SBAP: software-based attestation for peripherals. In: Acquisti, A., Smith, S.W., Sadeghi, A.-R. (eds.) TRUST 2010. LNCS, vol. 6101, pp. 16–29. Springer, Heidelberg (2010)
16. Li, Y., McCune, J.M., Perrig, A.: VIPER: verifying the integrity of peripherals' firmware. In: Proceedings of ACM Conference on Computer and Communications Security, pp. 3–16 (2011)
17. Martignoni, L., Paleari, R., Bruschi, D.: Conqueror: tamper-proof code execution on legacy systems. In: Kreibich, C., Jahnke, M. (eds.) DIMVA 2010. LNCS, vol. 6201, pp. 21–40. Springer, Heidelberg (2010)
18. Parno, B., McCune, J.M., Perrig, A.: Bootstrapping Trust in Modern Computers. SpringerBriefs in Computer Science, vol. 10. Springer, New York (2011)
19. Sagoian, C., Stamm, S.: Certified lies: detecting and defeating government interception attacks against SSL. In: Proceedings of ACM Symposium on Operating Systems Principles, pp. 1–18 (2010)
20. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: SCUBA: secure code update by attestation in sensor networks. In: Proceedings of ACM Workshop on Wireless Security, pp. 85–94 (2006)
21. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of ACM Symposium on Operating Systems Principles, pp. 335–350 (2007)
22. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: verifying integrity and guaranteeing execution of code on legacy platforms. In: Proceedings of ACM Symposium on Operating Systems Principles, pp. 1–16, October 2005
23. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: SWATT: software-based attestation for embedded devices. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 272–282 (2004)
24. Shaneck, M., Mahadevan, K., Kher, V., Kim, Y.-D.: Remote software-based attestation for wireless sensors. In: Molva, R., Tsudik, G., Westhoff, D. (eds.) ESAS 2005. LNCS, vol. 3813, pp. 27–41. Springer, Heidelberg (2005)

25. Shankar, U., Chew, M., Tygar, J.: Side effects are not sufficient to authenticate software. In: Proceedings of the USENIX Security Symposium (2004)
26. Spinellis, D.: Reflection as a mechanism for software integrity verification. ACM Trans. Inf. Syst. Secur. **3**(1), 51–62 (2000)
27. Tam, S.: Modern clock distribution systems. In: Xanthopoulos, T. (ed.) Clocking in Modern VLSI Systems, Chap. 2. Integrated Circuits and Systems, pp. 6–95. Springer, USA (2009)
28. Texas Instruments. AM/DM37X multimedia device technical reference manual. Version R, September 2012
29. The Trusted Computing Group. TPM Main specification version 1.2 (revision 116) (2011)
30. Wollinger, T., Guajardo, J., Paar, C.: Security on FPGAs: state-of-the-art implementations and attacks. ACM Trans. Embed. Comput. Syst. (TECS) **3**, 534–574 (2004)
31. Wurster, G., van Oorschot, P., Anil, S.: A generic attack on checksumming-based software tamper resistance. In: Proceedings of IEEE Symposium on Security and Privacy, pp. 127–138 (2005)
32. Zhao, J., Gligor, V., Perrig, A., Newsome, J.: ReDABLS: revisiting device attestation with bounded leakage of secrets. In: Christianson, B., Malcolm, J., Stajano, F., Anderson, J., Bonneau, J. (eds.) Security Protocols 2013. LNCS, vol. 8263, pp. 94–114. Springer, Heidelberg (2013)