Secure and Efficient Software-based Attestation for Industrial Control Devices with ARM Processors

Binbin Chen
Advanced Digital Sciences Center
binbin.chen@adsc.com.sg

Xinshu Dong
Advanced Digital Sciences Center
xinshu.dong@adsc.com.sg

Guangdong Bai
Singapore Institute of Technology
guangdong.bai@singaporetech.edu.sg

Sumeet Jauhar
Advanced Digital Sciences Center
sumeet.jauhar@adsc.com.sg

Yueqiang Cheng
Baidu USA XLab
chengyueqiang@baidu.com

ABSTRACT

For industrial control systems, ensuring the software integrity of their devices is a key security requirement. A pure software-based attestation solution is highly desirable for protecting legacy field devices that lack hardware root of trust (e.g., Trusted Platform Module). However, for the large population of field devices with ARM processors, existing software-based attestation schemes either incur long attestation time or are insecure. In this paper, we design a novel *memory stride* technique that significantly reduces the attestation time while remaining secure against known attacks and their advanced variants on ARM platform. We analyze the scheme's security and performance based on the formal framework proposed by Armknecht et al. [7] (with a necessary change to ensure its applicability in practical settings). We also implement memory stride on two models of real-world power grid devices that are widely deployed today, and demonstrate its superior performance.

KEYWORDS

Software-based attestation, industrial control devices, ARM Processors, memory stride

ACM Reference Format:
Binbin Chen, Xinshu Dong, Guangdong Bai, Sumeet Jauhar, and Yueqiang Cheng. 2017. Secure and Efficient Software-based Attestation for Industrial Control Devices with ARM Processors. In *Proceedings of ACSAC 2017*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3134600.3134621

1 INTRODUCTION

Industrial control systems (ICS) monitor and operate critical infrastructures like power grids and metro systems, which are of paramount significance to national security and people's daily lives. High-profile attacks (e.g., Stuxnet [1] and Ukraine Power Outage [9]) have well demonstrated the susceptibility of ICS to cyber attacks, and how stealthy such attacks can be. The use of malware on ICS devices plays a critical role in many of these attacks, as well as in new emerging forms of attacks demonstrated by researchers [3, 11, 14, 32, 39, 40]. In particular, in many ICS attacks (e.g., Ukraine Power Outage [9]), the attackers hide the malware for an extended period in the system in order to maximize the damage they can cause. Hence, ensuring the software integrity of ICS devices against various (especially stealthy) attacks is a foundational requirement for enhancing ICS security. Otherwise, all other mechanisms, e.g., secure protocols, can be subverted by malware on the devices.

Verifying the software integrity, i.e., *attestation*, is conducted based on some form of *root of trust* on the devices under attestation. This, together with the fact that ICS devices execute well-specified program logic, allows an attestation-based approach to provide high assurance about the software integrity of the attested device. In comparison, solutions like anti-virus or host-based intrusion detection systems only provide best-effort malware detection, hence do not result in a similar level of trust assurance. Advances in secure hardware make it possible to establish a hardware root-of-trust on a remote device [6, 22], and there are solutions that leverage these technologies to attest the integrity of software [8, 27]. However, a key challenge in applying hardware-based attestation solutions to ICS devices is the necessity of equipping all devices with secure hardware, as long as they can be affected by malware. The inconvenience and additional cost associated with such hardware-based solutions raise practical concerns for their deployment in real-world ICS, even though researchers have made notable progress in minimizing the hardware change requirements [15, 16].

In comparison, a SoftWare-Only Root of Trust (*SWORT*) [23, 25, 36] scheme does not require hardware support. Instead, it uses the following elegant idea: if the running copy of a carefully designed checksum software is unmodified, it would be able to compute a checksum over its own memory footprint faster than any modified versions. Intuitively, the checksum (seeded with some random nonce) is a fingerprint of the authentic executable image, and a modified software needs to do some extra work in order to "hide" its different memory footprint and return the correct checksum. Such a software-based attestation approach is highly desirable for legacy ICS devices that do not have secure hardware.

The need for efficient SWORT schemes in ICS. To use a SWORT scheme for ICS devices, the scheme's efficiency, i.e., the amount of time it requires, is important. There are two reasons behind this.

ACSAC 2017, December 4–8, 2017, San Juan, PR, USA

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134621

Firstly, many mission-critical ICS impose high availability requirement on their devices. For instance, operations in a power grid require real-time monitoring of the voltage, current, and other information across the grid. Hence, devices like phasor measurement units (PMUs) often require message delivery within cycle response time, i.e., 20*ms* for a 50-Hz power grid [2, 19]. We face such a stringent requirement when integrating our SWORT solution into a substation automation system (see Section 6). Since SWORT intrinsically consumes all computational resources of the device under attestation, it needs to strive for short execution duration to reduce its impact on ICS services.

Secondly, SWORT is vulnerable to *proxy attacks*, where the device under attestation asks a computationally more powerful remote device to compute the checksum. Short attestation time helps raise the bar for launching such proxy attacks. For example, if the SWORT process completes within 20ms, a proxy attack behind a slow link (e.g., the cellular backhaul of an electrical substation) would fail.

Let L denote the size of the SWORT code (e.g., $L = 2$ KBytes in our settings, see Section 6), and s denote the size of the fastest memory (usually the RAM) that hosts the SWORT code ($s \geq L$ and s ranges from several KBytes to MBytes or more in field devices). If the checksum needs to be computed over the whole memory (with size s), the attestation can take a long time to complete. This challenge has been reported in [25] for a high-end ARM-based device. As we will show, even for low-end ARM-based power grid devices that we experiment with, which have less than 100KBytes of RAM, walking over the entire RAM can take more than 150ms.

A key approach to reduce the attestation time is to design secure partial memory walk schemes. Prior research has proposed such secure schemes for Intel [36] and MIPS [34, 35] platforms that conduct the memory walk over the checksum code region only (with size L instead of s). Unfortunately, we found that those mechanisms do not apply to ARM devices, which are popular in power grids [21] and other ICS applications [10, 24][1]. Making things worse, malware on such ARM-based field devices has been available for a couple of years [11].

Our contributions. In this work, we focus on low-end ARM-based field devices operating in real-mode, because they are widely deployed in ICS and unlike their high-end cousins they do not have any secure hardware. We propose a novel *memory stride* technique to reduce the root-of-trust establishment time. Our design is secure against all known attacks on software-based attestation and their advanced variants on ARM platform, without having to access every memory word. The basic idea of memory stride design is to conduct the random walk on two subsets of addresses in the memory in an interleaving manner. One subset is the memory area that hosts the actual attestation code (of size L) and the other subset (of size $\frac{s}{L}$) is a group of memory addresses (called *stride addresses*) that span across the whole memory, with two neighboring stride addresses placed L distance apart (see Figure 3 in Section 4).

To understand how such a design can help reduce the attestation overhead, we perform a detailed analysis based on a formal framework adapted from the work by Armknecht et al. [7]. We found the upper bound provided by that generic framework [7] turns out to be too loose when being applied to a practical setting. Hence, we remove one pessimistic assumption made in the original framework (for which we are unaware of any feasible attack). By doing so and by introducing concrete parameter value ranges into our analysis, we significantly simplify the analysis results, and derive an $O(s)$ requirement for a full memory walk solution. Applying this adapted analysis framework to memory stride, we show that memory stride only needs to incur $O\left(L + \frac{s}{L}\right)$ overhead.

[1]For instance, the ARM Cortex-M processor family has been shipped in billions of devices, and claimed by ARM to be "the most popular choice for embedded applications" [5].

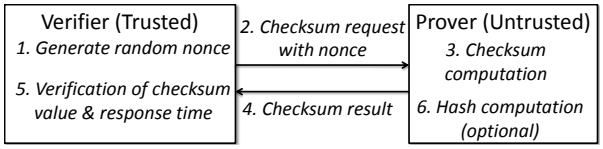

Figure 1: SWORT establishment process in a nutshell.

Last but not least, we implemented memory stride on several ICS boards deployed in real-world power grid devices. Our experiments show that the proposed techniques can reduce the downtime required for attesting the devices to less than 20*ms*, and it achieves $> 20x$ reduction for a device with around 100KByte of RAM.

In the following, Section 2 gives a primer of SWORT and surveys related work. Section 3 presents the threat model. Section 4 presents our novel memory stride technique. Section 5 analyzes the security and overhead of the proposed scheme. Section 6 presents the evaluation results on real-world devices. We conclude in Section 7.

2 SWORT: A PRIMER & PRIOR RESEARCH

In this section, we will first provide a brief primer of SWORT. We then discuss related prior research, focusing on secure partial memory walk schemes.

A primer of SWORT. Figure 1 illustrates the basic steps of conducting attestation based on SWORT establishment between a trusted *verifier* device and an untrusted *prover* device, who proves to the verifier that its software image is genuine.

The process starts when the verifier requests the prover to prepare an expected environment for attestation. This usually involves filling unpredictable or empty regions in the prover's memory with values generated by a pseudorandom number generator, which is sent by the verifier to the prover. This step is carried out by the prover without any time constraint.

Afterwards, the verifier sends a randomly generated nonce to the prover, and starts its clock (**Steps 1 & 2** in Figure 1). The prover uses the nonce to perform iterations of random walk over its memory, and updates a checksum value based on the content of randomly accessed memory footprint as well as the current running status (e.g., program counter value, memory address, and status registers) (**Step 3**). The number of iterations is determined by the verifier to ensure a sufficiently small probability that a malicious prover (with modified memory) can return a correct checksum within expected time (see details in Section 5). If *1)* the received checksum result is as expected, and *2)* it is received within the pre-defined time constraint, the verifier can be assured of the integrity of the software image on the prover device. This essentially establishes a root of trust on the prover (**Step 5**). Otherwise, this checksum step fails, and the prover can optionally re-initiate a new checksum request.

In such a scheme, a crucial property is that the checksum function is designed with time optimality running over the genuine software code. Any variant of it, e.g., memory copy attacks [36], must take longer time to compute the correct checksum result.

The checksum function needs to at least cover its own code and some small amount of other code that it uses (e.g., the code needed for communication). After the root of trust has been established, it can further compute a hash over other memory / storage region of interest (e.g., a boot image), but the computation of hash does not need to be timed (**Step 6**).

Prior work on SWORT schemes and their analysis. Significant research efforts have been devoted to the design of time-optimal and secure checksum functions for software-based attestation (e.g., [35–37]). One can find surveys of important work in this area, and debates / reflections on some key issues in [13, 25, 33].

Recent efforts also attempt to formalize the requirements and security guarantees of SWORT schemes. For instance, Armknecht et al. propose a formal framework to analyze the security goals, attacker models in software attestation, as well as key system and design parameters [7]. One key question they attempt to answer is on the number of iterations needed to ensure a sufficiently small probability for a prover with modified memory to cheat the verifier and *win* the attestation. The folklore answer to this question has been based on solving a coupon collector problem. Specifically, if one conducts a random walk over the whole memory (with size s), $O(s \ln(s))$ iterations of memory accesses are necessary to ensure a target constant probability that every word in the prover's memory is visited at least once [25]. In comparison, the formal framework in [7] provides a generic (and rather sophisticated) analytical solution for computing the required number of iterations. In Section 5, we will provide a detailed analysis about the number of iterations required based on the framework from [7] and propose a necessary change to apply it in realistic settings.

Secure partial memory walk. Compared to a full memory walk solution that requires visiting the entire memory (of size s), one can reduce the attestation time by conducting the memory walk over the SWORT code region (with size L) only.

Prior research has proposed secure partial memory walk schemes on the Intel [36] and MIPS [34, 35] platforms. In both works, the design of the schemes leverage the underlying platforms' architecture features. In particular, Pioneer [36] leverages Intel's variant instruction lengths to impose a different code size on memory copy attack (to be described in Section 4.2), making a modified checksum code jump into non-executable parts in the code block. Since all instructions on ARM platform are of fixed length, this defense does not apply to the ARM platform. For a MIPS-based platform MSP430, the SCUBA & ICE scheme [34, 35] defeats attacks that involve immediate value, by using the MSP430 feature that an operation over immediate values takes extra CPU cycles. However, as we will discuss in Section 4.2, ARM-friendly immediate values can be used without any overhead on ARM platforms.

Franklin et al. [18] discuss about a partial memory walk solution for ARM platform, and recognize that the attackers may be able to launch attacks by using instructions with immediate values. Although they discuss the restrictions in the range of immediate values, they do not explicitly propose defenses against advanced memory copy attacks that we will present in Section 4.2. Another prior work [38] mentions that certain memory copy attacks can be prevented by "a self-modifying checksum function". However, they "do not use a self-modifying checksum function in the current implementation". We are not aware of existing work on self-modifying checksum function for SWORT, although self-modifying code has been used in other applications [17]. Related to it, obfuscated checksum function may help raise the bar for memory copy attacks with partial memory walk [26], but does not eliminate it.

3 THREAT MODEL

In this paper, we focus on SWORT establishment for ICS devices with low-end ARM processors. These devices are usually deployed in the field (e.g., inside an electrical substation) to serve a long period of time (10 years or more). They do not have any secure hardware, and upgrading them with secure hardware is unlikely to happen soon. We assume there is a trusted attestation verifier that locates in the same local area network (LAN) as the device under attestation. For an electrical substation, the verifier can be its secure gateway or the on-site Human Machine Interface (HMI) machine. The trust of this verifier can be established, e.g., by secure hardware or other means like its compliance with common criteria. We assume the communication delay between the verifier and the prover is short and stable (we will examine this assumption in Section 6). We also trust the prevalent stringent physical access control to the ICS environment, and thus any form of physical attack inside the environment (e.g., physically changing the device hardware or adding an extra device inside the LAN) is out of scope. Albeit the above assumptions, malware can still enter the closed ICS network via various means, e.g., an innocent operator may swap-in a replacement board with malware, where the malware may come from an infected computer that is used to burn the firmware onto the board. Or, malware can be introduced from a maintenance computer that is directly connected to the LAN during diagnosis. It can also be an occasional violation of security policies by operators or vendors who plug the USB drives into an in-network computer. In principle, existing techniques that penetrate the "air-gap" in ICS networks constitute the attack vectors here. However, we assume any attack vector that penetrates the "air-gap" does not persist, and is disabled during the attestation process via procedural means (e.g., by not granting physical access to any personnel during certain periods).

As stated in Section 1, we require a secure attestation scheme to complete as soon as possible, so as to satisfy the availability requirements stipulated by ICS and to increase the bar for attackers to launch proxy attacks. As a concrete benchmark, we use $20ms$, which is required by time-critical power grid applications [2, 19]. We also assume $20ms$ is short enough to make proxy attacks from external network (e.g., from outside of a substation) impossible.

In general, one may perform attestation regularly to detect malware hidden in the system. We do not address malware that launches outright denial-of-service attacks, which are easily detectable. Instead, we focus on attesting devices for detecting surreptitious malware. For example, in the Ukraine power system attack [9], the attackers penetrated into the systems and implanted malware a few months before they launched the actual attack around Christmas. With regular attestation of ICS devices, the implanted malware can

be detected before actual damage happens. In addition, attestation can be performed on demand. For instance, consider the scenario when a critical software vulnerability is identified on some models of ICS devices, the ICS operator can first conduct the attestation before applying a patched version to fix the vulnerability.

Instead of modifying the code, an attacker may also change the configuration of an ICS device, i.e., attack the data. If the code can be attested to be genuine, we can then trust the code to check the integrity of the device configurations assuming it can access a cryptographically signed copy of the genuine configuration.

4 SWORT USING MEMORY STRIDE

This section discusses the overhead of a full memory walk scheme, then presents our memory stride technique for ARM-based devices.

4.1 The Overhead of Full Memory Walk

As presented in Section 2, the main body of a checksum code is to execute N iterations of random memory walk, in each iteration, it generates a new random memory address, reads the memory content in the generated address, and uses the memory content to update a checksum value. The overhead of the scheme, hence depends on the number of iterations N and the overhead of each iteration. For the overhead per iteration, as we will show in Section 4.4, it needs to be around 30 CPU cycles even for the low-end ARM devices that we experiment with. A large part of these CPU cycles is used to ensure all registers in the CPU are occupied in each iteration. For N, we will conduct a detailed analysis in Section 5 by applying the formal framework from [7]. As the main conclusion from our analysis in Section 5, $O(s)$ iterations of memory access / checksum update are needed for a full memory walk over a memory size of s words. Specifically, given a target probability, denoted by $Pr[win]$, that a malicious prover with modified memory state can cheat the verifier and win the attestation, one requires to conduct $N = s \times ln\left(\frac{1}{Pr[Win]}\right)$ iterations (see Equation (5) in Section 5).

To illustrate the overhead of a full memory walk scheme in a concrete ICS device setting, we use a low-end remote terminal unit (RTU) with a NXP LPC2292 processing board as an example. This series of devices have been deployed widely in the power grids in South East Asia region. LPC2292 includes an ARM7TDMI-S CPU that runs at 60MHz and 16KB on-chip RAM. Our results in Section 6 show that it takes 52.1ms to achieve a $Pr[win] = 10^{-10}$. (more than 2× higher than our 20ms target despite of its small RAM size). As will become clearer by Section 4.2, full memory walk is actually the only existing SWORT solution for ARM that is secure under advanced ARM-specific attacks that we are going to discuss. Since N needs to grow linearly with the RAM size s, the attestation time will further increase to 156.9ms when the RAM sizes increases to 58KBytes, as in the case of a LPC2362 board, which is in the same series as LPC2292, but has an ARM7 72MHz CPU and a 58KB RAM. Even when one substantially reduces the guarantee to $Pr[win] = 10^{-5}$, full memory walk on RAM of these boards still take 26.1ms and 78.5ms respectively (see details in Section 6).

In practice, there are ICS devices with significantly larger RAM size than what we have experimented. For example, [25] reports their attestation experiment on an ARM-based board with a few hundred megabytes of RAM, for which, a full memory walk will take several minutes to complete.

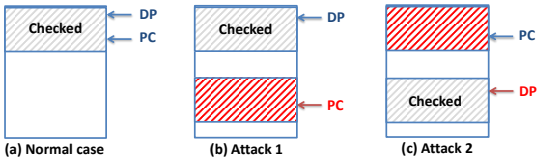

Figure 2: Memory copy attacks with partial memory walk. PC: Program Counter, DP: *Conceptual* Data Pointer to memory access location.

4.2 Challenges for Partial Memory Walk

Unlike full memory walk, there are unchecked memory regions in a partial memory walk, which can be used by an attacker to launch *memory copy attacks* [36].

Memory copy attacks. As illustrated in Figure 2, there are two main types of memory copy attacks that correspond to two important pointers used in SWORT checksum calculation: the program counter (PC) and a conceptual data pointer (DP), which is the memory address taken by the random memory walk. *Attack 1* is to preserve the location and content of the original code, i.e., unmodified data pointer value, while having the PC point to a different region. On the other hand, *Attack 2* preserves the PC values, but maintains a clean copy of the original code in another part of the memory[2]. Hence its DP value changes. Since both PC and DP values are computed into the checksum as the program runs, these attacks need to restore their original value every time they are used in the checksum. Naively doing so will incur overheads. However, attackers can exploit platform-specific features to hide such overheads. We will discuss such features available at ARM platforms in the following.

Memory copy attack's potential leverage on ARM. ARM is a reduced instruction set computer (RISC) architecture, with a uniform length of instructions, e.g., 32 bits in 32-bit mode (ARM mode). There are several features on ARM[3] that can be leveraged by attackers to hide the overhead of launching Attacks 1 & 2.

"Free" Piggybacked Shift: Various logical and arithmetic operations allow for a shift operation for up to 32 bits to be "piggybacked" for its second operand [4]. For instance, instead of using both `LSR r1, r1, 2` and `ADD r0, r0, r1`, we can have a single instruction of `ADD r0, r0, r1 LSR 2` that takes 1 CPU cycle.

"Free" Offset/Shift with `LDR`: The `LDR` pseudo-instruction can take an immediate value as constant in loading the content of a memory address into a register. For example, `LDR r0, [r1, #960]` loads the content of the address 960 bytes above that stored in $r1$ into $r0$. `LDR` also supports shift operations that do not require additional CPU cycles, e.g., `LDR, r0, [r1, r2, LSL #2]`.

"Free" ARM-Friendly Immediate Value: Many ARM instructions support the use of an immediate value as its operand. It incurs no overhead (in terms of both the instruction execution time and the instruction length) compared to operating over registers. Since all ARM-mode instructions need to be kept 32-bit long, only part of

[2] An attacker can also modify both the PC and DP. However, it incurs more overhead since both need to be restored.

[3] We have confirmed these features on both ARM7 and Cortex-M3.

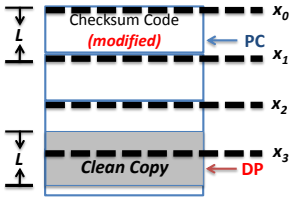

Figure 3: Illustration of stride addresses and the collision between one stride address with a shifted code region.

it can be used to express immediate values. In particular, ARM's encoding scheme for immediate values embedded in an instruction uses 8 bits to store a value, and another 4 bits to store the rotation operations [28]. This encompasses most of "useful" 32-bit values that do not have 0s and 1s spread beyond 8 bits. Besides, some instructions also allow another encoding scheme of a 12-bit value, ranging from 0 to 4095. Depending on which encoding schemes are used, those encodable values are called *ARM-friendly immediate values*.

Zero-overhead memory copy attacks on ARM. With these ARM-specific features in their disposal, attackers can develop advanced variants of memory copy attacks.

Attack 1A: Hardcoded PC values. As noted also in previous research [18, 38], the attacker can use a hardcoded immediate value to restore the PC to the expected value, when a malicious copy of the code is executed. The defense against this is straightforward. The checksum code should be loaded into memory address ranges where the expected PC values are not ARM-friendly.

Attack 2A: Offset-based memory access. The attacker can modify the memory access instruction during memory walk from `LDR r0, r1` to `LDR r0, r1, #0x1000`, where `0x1000` is the starting address of the clean copy of the original code. Similarly, it can redirect the memory access based on the current PC value, as `LDR r0, r15, #0x1000`. Unlike on MIPS [34, 35], such offset values with `LDR` instructions do not trigger additional CPU cycles on ARM.

For Attack 2A, prior solutions for the generic family of attack 2 on other architectures are inapplicable here. In particular, as a RISC architecture, ARM instructions, including `LDR` instructions with offset values, have the same length. This makes solutions that detect such offset inclusion based on varying instruction lengths ineffective [36].

In the following, we will propose a new solution that only conducts partial memory walk over the RAM, while preventing such free-offset attacks with the `LDR` instruction.

4.3 Memory Stride Design

To defeat Attack 2A without conducting memory walk over all the memory addresses, we conduct two types of memory walks over two subsets of the memory space in an interleaving manner:

- The first is a full memory walk over the entire region of SWORT code (with size of L memory words, which is less than the full memory size of s).
- The second is a partial memory walk over a set of disjoint memory addresses in the RAM. We call these addresses *stride addresses*, as they form a set of equally spaced strides in the RAM with a distance of L between neighboring stride addresses. An example is illustrated in Figure 3, where the four stride addresses are x_0, x_1, x_2, x_3. These stride addresses together divide the whole RAM into small and fixed size partitions. For instances, with an interval of L = 1KB between the strides and a RAM starting address at `0x40000000`, the following are stride addresses: `0x40000400`, `0x40000800`, `0x400000C00`, ... For a memory with s words, there are $\frac{s}{L}$ stride addresses.

The stride memory walk and the code memory walk interleave with each other, i.e., one stride memory walk is followed by a code memory walk, then back to stride memory walk, so on and so forth. This is to prevent the attacker from changing the memory layout in between. Since the total number of memory addresses that need to be covered, i.e., the code region and the stride addresses, $O\left(L + \frac{s}{L}\right)$ is only a fraction of the total memory size s, the number of memory access iterations needed by the checksum code can be significantly reduced, hence saving the attestation time.

There is one further design to make our solution secure: we generate random contents to be stored in stride addresses (except for the one that overlaps with the code region), and use a verifier to ensure that these random values do not accidentally match any word in the SWORT code region. Note that, we do not need to change or access the rest of the memory. These random contents do not need to be kept in secret, nor do they need to be changed across devices or over time.

The security of our memory stride technique depends on the following simple observation. Recall that in ARM, an attacker can have "free" offset with LDR. Hence, it can potentially shift either the SWORT code region or the stride addresses around the RAM without increasing the checksum computation time. The only constraint for the attacker is that it has to shift the whole code region as a continuous whole and move all the stride addresses with the same offset, otherwise some conditional check will be needed by the attacker, which incurs additional CPU cycles.

The construction of the memory stride counters such attacks by guaranteeing that regardless of how the attacker moves the code region (to a region outside the correct code region) or moves the stride addresses (while keeping all stride addresses inside the memory), there will always be at least one overlapping word between these two address spaces[4] and our construction ensures that: For the overlapped address, when it is accessed by the full memory walk over the code region, it needs to return the "expected" memory content for the code region; when it is accessed by the memory stride, it needs to return the "expected" memory content for the corresponding stride address. These two "expected" values are different. Since a single address can only store one value, at least one of them will return a wrong value that leads to a wrong checksum, unless the attacker incurs extra overhead to deal with this.

We also extensively study how an attacker may bypass our defense, assuming she knows all the details about our implementation. We are not able to find any way that the attacker can succeed without incurring noticeable extra overhead. One advanced form of attack is to split the code region into multiple disjoint segments to avoid the collision with stride addresses, but since different segments require the addition of different offset values before the

[4]The attacker can use a non-word-aligned offset. However, accessing non-aligned addresses incurs extra memory access overhead.

generated addresses can be used, this requires additional control-flow instructions for every memory access, hence incurs significant overhead. Another potential attack is to move all values at stride addresses into a contiguous memory region and rewrite the checksum computation code to read from that contiguous region. Again, this incurs additional instructions for each memory address computation. Also, since the stride addresses are accessed in a random way and interleaved with code region accesses, we do not see obvious memory access time reduction by putting the stride address content together, as compared to the significant slowdown due to the extra instructions required.

Conservatively speaking, an attack will at least require the attacker to introduce one additional CPU cycle for every step of one type of memory walk. (We do not have a working attack that can achieve so though.) Since each step takes about 30 CPU cycles and there are two types of memory walks, this translates to an extra overhead of $1/(30 \times 2) \approx 1.6\%$. We assume a malicious code needs to incur at least 1.6% of extra computation time than a genuine code. This can serve as a delay budget to accommodate potential communication delay variation.

4.4 Memory Stride Implementation

We have implemented memory stride, and verify our implementation on two popular low-end ARM platforms, i.e., ARM7 and Cortex M3. In our implementation, we further introduce some low-level design techniques as presented in Algorithm 1.

In both platforms, there are 16 directly accessible registers, i.e., $r0$ to $r15$. We use $r0$ to $r12$ to store the random nonce from the verifier, where $r0$ is used to keep the value of a pseudo-random T function for generating memory addresses, and $r1$ to $r12$ keep the checksum value. We re-purpose the stack pointer $r13$ as a loop iterator. Since its value needs to be a multiple of 8, we set $r13$ to be 8× the number of remaining outer loops in our code, and reduce its value by 8 (instead of 1) in each loop (Line 2 in Algorithm 1). We re-purpose the link register $r14$ as a temporary holder for memory address, memory content, and status register loading. The special register $r15$ is the program counter. In our implementation, the whole SWORT code (i.e., checksum computation and communication code) fits into the first 2KB of the RAM. We reduce the number of instructions in address masking by splitting the code region into two of 1KB blocks. As a result, the memory addressing masks (e.g., $0x03FC$) become ARM-friendly immediate values, saving additional instructions (or registers) to compose their values.

As shown, the update of checksum for each of the 12 checksum registers is almost identical except for how they compute the memory walk addresses. They are divided into 3 groups. Registers $r1$ to $r4$ form the first group. $r1$ is updated with a random word from 0-1 KB of the RAM, $r2$ and $r4$ are used for memory stride walk, and are updated with a word from a random stride address from the whole RAM (16 KB in the pseudo code example, but can be replaced with larger value), and $r3$ is updated with a random word from the 1-2 KB of the RAM. The checksum update for registers $r5$-$r8$ and $r9$-$r12$ are the same as the group of $r1$ to $r4$. We now use $r1$ (Lines 3 to 10) to explain the update logic of checksum in one register. The first step is to generate a random number using the T-function (Line 3). We then generate the memory walk address (Line 4). To prevent

Algorithm 1 Memory stride pseudocode

1: **while** $r13 > 0$ **do**
2: $r13 \leftarrow r13 - \texttt{0x08}$;
Update $r1$ *with a random word from* $0-1KB$ *of RAM*
3: $r0 \leftarrow r0 + (r0^2 \vee \texttt{0x05})$;
4: $r14 \leftarrow r0 \wedge \texttt{0x03FC}$ + base-addr;
5: $r1 \leftarrow r1 \oplus r14$;
6: $r14 \leftarrow \text{mem}[r14]$;
7: $r0 \leftarrow r0 \oplus r14$;
8: $r14 \leftarrow$ status register;
9: $r1 \leftarrow$ rotate_right$(r1 + r0 \oplus r14 + r15 \oplus r12 + r11 \oplus r13 + r10 \oplus r9 + r8 \oplus r7 + r6 \oplus r5 + r4 \oplus r3 + r2)$;
10: $r0 \leftarrow r0 \oplus r1$;
Update $r2$ *with a random stride address in RAM*
11: $r0 \leftarrow r0 + (r0^2 \vee \texttt{0x05})$;
12: $r14 \leftarrow r0 \wedge \texttt{0x3FC0}$ + base-addr; // $16KB$ RAM
13: $r2 \leftarrow r2 \oplus r14$;
14: $r14 \leftarrow \text{mem}[r14]$;
15: $r0 \leftarrow r0 \oplus r14$;
16: $r14 \leftarrow$ status register;
17: $r2 \leftarrow$ rotate_right$(r2 + r0 \oplus r14 + r15 \oplus r12 + r11 \oplus r13 + r10 \oplus r9 + r8 \oplus r7 + r6 \oplus r5 + r4 \oplus r3 + r2)$;
18: $r0 \leftarrow r0 \oplus r2$;
Update $r3$ *with a random word from* $1-2KB$ *of RAM*
19: $r0 \leftarrow r0 + (r0^2 \vee \texttt{0x05})$;
20: $r14 \leftarrow r0 \wedge \texttt{0x03FC}$ + base-addr;
21: $r3 \leftarrow r3 \oplus r14$;
22: $r14 \leftarrow \text{mem}[r14 + \texttt{0x0400}]$;
23: $r0 \leftarrow r0 \oplus r14$;
24: $r14 \leftarrow$ status register;
25: $r3 \leftarrow$ rotate_right$(r3 + r0 \oplus r14 + r15 \oplus r12 + r11 \oplus r13 + r10 \oplus r9 + r8 \oplus r7 + r6 \oplus r5 + r4 \oplus r3 + r2)$;
26: $r0 \leftarrow r0 \oplus r3$;
Update $r4$ *like* $r2$ *(with a random stride in RAM)*
Update $r5 - r8$ **&** $r9 - r12$ *as two pairs like* $r1 - r4$
27: **end while**
28: Send the values of $r1$ to $r12$ to the verifier;

attack 2, we incorporate this address into the checksum (Line 5). The value retrieved from the randomly generated memory walk address (line 6) is incorporated into the checksum (Line 7). The values of status register (Line 8) and other directly accessible registers (Line 9) are also incorporated into checksum to prevent an attacker from using them. The status register contains information about the state of the processor(privileged or non privileged mode, interrupt setup of the processor, thumb or arm mode, etc) and status of the program (state of the conditional flags, etc.). We include the status register for each checksum update to ensure that any attempts by the attacker to deviate from the expected program state will result in an incorrect checksum. For Line 9, note that the PC value ($r15$) is included to defend against attack 1A (Section 4). Also, the update is done as a strongly ordered sequence of AND & XOR instructions, which prevents potential speed-up via in-parallel computation. To prevent the T-function values from being computed in parallel we update register $r0$ (i.e, the T-function register) with the updated checksum value (Line 10).

5 SECURITY & PERFORMANCE ANALYSIS

In this section, we conduct detailed analysis of the security and performance properties of attestation schemes, based on the formal framework proposed by Armknecht et.al. [7], referred to as the *ASSW framework* hereafter.

5.1 A Recap of the ASSW Framework [7]

In the following, we give a quick recap of the ASSW framework. We reuse most of the terms from ASSW framework, which are underlined, and we refer the readers to [7] for their detailed definitions. Table 2 in Appendix lists the main notations used in the ASSW framework.

The ASSW framework abstracts a full memory walk SWORT scheme as N iterations, where each iteration invokes the following three functions sequentially: *Gen*, *Read*, and *Chk*. The processor executes the memory address generator *Gen* and the compression function *Chk* over the *primary memory* (PM), i.e., the registers, while *Read* uses the addressed generated by *Gen* to access *secondary memory* (SM), i.e., RAM in our setting. $\mathcal{V}$ indicates the verifier, $\mathcal{P}$ the prover and $\widetilde{\mathcal{P}}$ a malicious prover. r_i and r'_i, denote the checksum values after the ith iteration, for $\mathcal{P}$ and $\widetilde{\mathcal{P}}$ respectively.

$\widetilde{\mathcal{P}}$ is said to win the attestation game, iff it can respond $r'_N = r_N$ within time $\delta := N(\delta_{gen} + \delta_{read} + \delta_{chk})$, where δ_{gen}, δ_{read} and δ_{chk} indicate the time for executing *Gen*, *Read* and *Chk* respectively.

The ASSW framework formalizes the following key factors that security of an attestation scheme depends on.

State incompressibility, which requires that the memory state should not be compressible into the PM. It indicates the difference level of the state entries at two arbitrary addresses, and is characterized by $\gamma = \max_{x \in \Sigma} \mathbb{D}_S(x)$, where $\mathbb{D}_S$ denotes the probability distribution of a state S: $\mathbb{D}_S(x) := Pr\Big[x = s | a \xleftarrow{\mathbb{U}} \{0,1\}^{l_a} \wedge s \leftarrow Read(S, a)\Big]$, and $\mathbb{U}$ indicates the uniform distribution.

Time-bounded pseudo-randomness and unpredictability of *Gen*. An address generator has to be sufficiently random such that the malicious prover $\widetilde{\mathcal{P}}$ is not able to use precomputed values. Basically, a truly random *Gen* can generate any address combination $\{a_1, ..., a_N\}$ when it is executed for N times. ASSW framework defines the time-bounded pseudo-randomness of *Gen* with parameters of (t, ϱ), such that for any algorithm *Alg* that $\mathcal{P}$ can execute within time bound t, it holds that

$$\Big| Pr\Big[b = 1 | g_0 \xleftarrow{\mathbb{U}} \{0,1\}^{l_g} \wedge (g_i, a_i) \leftarrow Gen(g_{i-1}) : i \in \{1, ..., N\} \wedge b \leftarrow Alg(a_1, ..., a_N)\Big] - Pr\Big[b = 1 | a_i \xleftarrow{\mathbb{U}} \{0,1\}^{l_a} : i \in \{1, ..., N\} \wedge b \leftarrow Alg(a_1, ..., a_N)\Big]\Big| \leq \varrho.$$

Besides randomness, *Gen* also needs to be time-bounded unpredictable, i.e., for any algorithm $Alg^{\widetilde{Gen}}$ that $\mathcal{P}$ can execute without invoking *Gen*, it holds that

$$\Big| Pr\Big[b = 1 | g \xleftarrow{\mathbb{U}} \{0,1\}^{l_g} \wedge (g', a') \leftarrow Gen(g) \wedge b \leftarrow Alg^{\widetilde{Gen}}(g, g', a')\Big] - Pr\Big[b = 1 | g \xleftarrow{\mathbb{U}} \{0,1\}^{l_g} \wedge (g', a') \xleftarrow{\mathbb{U}} \{0,1\}^{l_g + l_a} \wedge b \leftarrow Alg^{\widetilde{Gen}}(g, g', a')\Big]\Big| \leq \nu_{Gen}.$$

Blind second pre-image resistance and unpredictability of *Chk*. The purpose of *Chk* is to hash the state entries into the response of fixed size. Compared to the common notion of second pre-image resistance requirement for cryptographic hash functions, *Chk* only needs to satisfy a weaker form of blind second pre-image resistance with parameter of ω. Specifically, a hash function $chk : \{0,1\}^{l_r} \times \Sigma \to \{0,1\}^{l_r}$ is ω-blind second pre-image resistant if for any algorithm *Alg* that $\mathcal{P}$ can execute, it holds that

$$Pr\Big[\widetilde{r} = r | r_0 \xleftarrow{\mathbb{U}} \{0,1\}^{l_r} \wedge s_i \xleftarrow{\mathbb{D}_S} \Sigma : i \in \{1, ..., N\} \wedge \widetilde{r} \leftarrow Alg(r_o, (s_j)_{j \in J}) \wedge r \leftarrow Chk^N(r_0, s_1, ..., s_N)\Big] \leq \omega,$$

where $\mathbb{D}_S$ denotes the probability distribution of S.

In addition, Chk^N also need to be unpredictable. Formally, a hash function $Chk : \{0,1\}^{l_r} \times \Sigma \to \{0,1\}^{l_r}$ is ν_{Chk}-unpredictable if for any algorithm $Alg^{\widetilde{Chk^N}}$ that $\mathcal{P}$ can execute without invoking Chk^N, it holds that

$$\Big| Pr\Big[b = 1 | r_0 \xleftarrow{\mathbb{U}} \{0,1\}^{l_r} \wedge s_i \xleftarrow{\mathbb{D}_S} \Sigma : i \in \{1, ..., N\} \wedge r \leftarrow Chk^N(r_0, s_1, ..., s_N) \wedge b \leftarrow Alg^{\widetilde{Chk^N}}(r_0, s_1, ..., s_N, r)\Big] - Pr\Big[[b = 1 | r_0 \xleftarrow{\mathbb{U}} \{0,1\}^{l_r} \wedge s_i \xleftarrow{\mathbb{D}_S} \Sigma : i \in \{1, ..., N\} \wedge r \xleftarrow{\mathbb{U}} \{0,1\}^{l_r} \wedge b \leftarrow Alg^{\widetilde{Chk^N}}(r_0, s_1, ..., s_N, r)\Big]\Big| \leq \nu_{Chk}.$$

The Generic upper bound on $Pr[Win]$ from [7] (with corrections on two minor typo errors). Recall that *Win* denotes the event that a malicious prover $\widetilde{\mathcal{P}}$ responds a correct checksum r'_N) within the time constraint. [7] derives a generic upper bound for $Pr[Win]$ as follows.

$$Pr[Win] \leq \frac{p+s}{l_r/l_s} \cdot 2^{-(l_g + l_r)} + max\{\omega, \nu_{Chk}\} + \max_{0 \leq M \leq N} \{(\pi(M, ops) + \varrho) \cdot \gamma^{N-M} + \nu_{Gen} \cdot (N - M)\} \quad (1)$$

where p and s denote the number of memory words in PM and SM respectively, l_g, l_r and l_s denote the number of bits in the address generation random seed, checksum response and in a memory word respectively, and

$$\lambda := \Big|\Big\{a \in \{0,1\}^{l_a} | Read(\widetilde{S}, a) = Read(S, a)\Big\}\Big| \cdot 2^{-l_a}$$

denotes the fraction of words that are identical in $\widetilde{S}$ and S. Further,

$$\pi(n,x) := \sum_{j=max\{0,n-2^{l_a}\}}^{n-1} (max\{\lambda^{x+1},\gamma\})^{\frac{n}{x+1}-j} \cdot \binom{n}{j} \cdot \left(\prod_{i=0}^{n-j} \frac{2^{l_a}-i}{2^{l_a}}\right) \cdot \left(\frac{n-j}{2^{l_a}}\right)^j \quad (2)$$

There are two typo errors in the original Theorem 1 in [7], which we corrected when reproducing the results:

- The first typo is in the first term for $Pr[Win]$ upper bound, where the original equation in [7] uses $\frac{p+s}{l_s/l_r}$ for the number of responses the PM and SM can hold. We have corrected it to $\frac{p+s}{l_r/l_s}$.
- The second typo is about the definition of λ, which should denote the similarity of $\widetilde{S}$ and S, instead of "fraction of state entries that are different" as in [7].

Intuition behind the upper bound result for $Pr[Win]$ in [7] and a simplified form under realistic settings. The upper bound for $Pr[Win]$ in Equation (1) is derived from a sequence of games for the malicious prover. We refer the readers to the original paper for detailed analysis.

Intuitively, the first term in Equation (1), i.e., $\frac{p+s}{l_r/l_s} \cdot 2^{-(l_g+l_r)}$ captures the probability that an attacker pre-computes a subset of possible responses and stores them in the PM and SM, hence, when the selected random input happens to fall in the pre-computed subset, an attacker can win. Note that, in our setting $l_g = 32$, $l_r = 32 \times 12$, making this term very small (e.g., even when $p+s$ contains $100G$ memory words, this term is as small as 5.28×10^{-116}), hence we ignore this term hereafter.

For the second term, it captures the probability that an attacker may be able to guess the final checksum without knowing all necessary inputs, due to the blind second pre-image resistance and unpredictability of Chk, which are captured by ω and v_{Chk} respectively. While the Chk function being used in practice is likely imperfect, there is no reported attack that can make any effective guess. Hence, as in the numerical example given in Appendix B of [7], we assume $\omega = 2^{-l_r}$ and $v_{chk} = 0$, making this term again very small, i.e., 2.53×10^{-116} in our setting. Hence, we ignore this term hereafter as well.

Now let us look at the last (and the most complicated) term in Equation (1), i.e., $\max_{0 \le M \le N}\{(\pi(M, ops) + \varrho) \cdot \gamma^{N-M} + v_{Gen} \cdot (N - M)\}$. M here captures the attack strategy that a malicious prover follows the procedure for the first M iterations, then switches to guessing the generated addresses in the following $N - M$ iterations. Again, while the address generation function used in practice (predominantly the T-function) is imperfect, there is no reported attack that can make effective guess yet. Hence, we follow the numerical example given in Appendix B of [7] and assume $v_{Gen} = 0$. Under this setting, the optimal strategy is to choose $M = N$ so the attacker never risks to guess the address. Also, since there is no reported attack to the randomness of the Gen function used in practice, we follow the numerical example given in Appendix B of [7] and assume $\varrho = 0$.

Based on the above discussions, Equation (1) can be simplified to

$$Pr[Win] \le \pi(N, ops) \quad (3)$$

For π as defined in Equation (2), it captures the advanced strategy of an attacker to optimally allocate time among different iterations so as to maximize its winning probability. The complexity of the equation mainly comes from the fact that the ASSW framework intends to capture the impact of any memory addresses that have been visited in some prior iterations, or so called *collision addresses*. The ASSW framework makes a rather strong and pessimistic assumption about access to collision addresses, i.e., the attacker can win that iteration without using any time. The j in Equation (2) captures the possible number of iterations with collision addresses, and the term $\binom{n}{j} \cdot \left(\prod_{i=0}^{n-j} \frac{2^{l_a}-i}{2^{l_a}}\right) \cdot \left(\frac{n-j}{2^{l_a}}\right)^j$ is the probability to have exactly j our of a total of n iterations that access collision addresses. Given there are exactly j iterations with collision addresses, the ASSW framework derives that the conditional probability for an attacker to win the attestation is $max\{\lambda^{x+1},\gamma\})^{\frac{n}{x+1}-j}$.

5.2 Applying the ASSW Framework (with a Necessary Change for Practical Settings)

In the following, we will first present why the pessimistic assumption in the ASSW framework about the collision addresses should be removed to make the ASSW framework applicable in practical settings. We then propose our modification to the framework and apply the modified framework to analyze memory stride.

A review on the collision address assumption in ASSW framework, and the rationale for its removal. When deriving Equation (2), the ASSW framework makes two strong and pessimistic assumptions: 1) once an iteration accesses a collision address, the attacker has 100% chance to win, i.e., compute the correct checksum for that iteration while using 0 time, and 2) the attacker has 100% chance to win one iteration, if the attacker spends $(ops + 1)$ (instead of $ops = \delta_{Gen} + \delta_{Read} + \delta_{Chk}$) time for one iteration.

We now show that these two assumptions together make the upper bound derived from the ASSW framework too loose for meaningful use in practice. We discuss the following two cases of N.

1) When $N \ge 2^{l_a} \times (1 + 1/ops)$, since the number of collision addresses $N_{coll} \ge N - 2^{l_a}$ (recall that there are a total of 2^{l_a} addresses in SM), we have $N_{coll} \ge N/(ops + 1)$. Hence $N - N_{coll} \le N \times ops/(ops + 1)$. So, for the $N - N_{coll}$ non-colliding memory addresses, the attacker can afford to let each of them use $ops + 1$ operations while keeping the total time within the allowed time of $N \times ops$. As a result, by the second assumption as made in the ASSW framework, all of these $N - N_{coll}$ iterations succeed with 100% chance, same for all the N_{coll} with collision addresses. In summary, these two strong assumptions give a trivial upper bound of 1 for $N \ge 2^{l_a} \times (1 + 1/ops)$.

2) When $N < 2^{l_a} \times (1 + 1/ops)$, we consider the following simple attack strategy, where the attacker follows the exact checksum computation process and reads every memory word as decided by Gen. In this case, $Pr[Win]$ is equal to the probability that the memory word that contains different value in S and $\tilde{S}$ is not accessed. Under this strategy,

$$Pr[Win] = \lambda^N > (1 - \frac{1}{2^{l_a}})^{2^{l_a} \times (1+1/ops)} \approx \frac{1}{e}^{1+1/ops}$$

Note that, for λ, it captures the similarity between $\widetilde{S}$ and S. As correctly pointed out in the Remark 2 of [7], "a prover $\widetilde{\mathcal{P}}$ is already considered to be malicious even if its state differs by only one state entry (memory word) from S", hence we set $\lambda = 1 - \frac{1}{2^{l_a}}$ above. For $2^{l_a} > 256$ and $ops = 30$, $Pr[Win] > 0.35$. Even if one changes 2^{l_a} and ops to other possible values, $Pr[Win]$ always remains high, i.e., > 10%. Such a high probability is non-acceptable in practice for software-based attestation. In summary, the two pessimistic assumptions made by the ASSW framework together lead to the fact that regardless of how one designs the attestation scheme, the upper bound for the attacker's success rate is too loose to be of practical use.

Note that, the original paper [7] does not realize this issue, because they consider a constant value of $\lambda = 0.8$ when carrying out the numerical calculation. In comparison, we are considering the more realistic setting of having $\lambda = 1 - \frac{1}{2^{l_a}}$.

Our proposed change to the ASSW framework. To fix this problem, we re-examine a practical attestation setting. Since there is no extra space in the PM to store a value that has been accessed before, we propose to drop ASSW framework's pessimistic assumption about collision addresses. Instead, we assume that a collision address does not help attacker. We still keep all the other assumptions.

With this weaker model, since the total time is fixed at $ops \times N$, it means that if the attacker decides to have N_{more} number of iterations, where they allocate one additional CPU cycle for achieving with 100% success probability for that iteration, there should be correspondingly at least $N_{less} = \frac{N_{more}}{ops}$ iterations, in each of which, the attackers use less than ops time, hence the probability to win each of such iterations is upper bounded by γ. Consider the probability to win all these $N_{more} + N_{less}$ iterations. It is upper bounded by $\gamma^{N_{less}}$. Instead, if the attacker does not perform such rearrangement of time across these $N_{more} + N_{less}$ iterations, its probability to win all of them is upper bounded by $\lambda^{N_{more}+N_{less}} = \lambda^{N_{less}\times(1+ops)}$.

Recall that in a realistic setting, $\lambda = 1 - 2^{l_a}$ $(a > 8)$, γ is small, e.g., $\gamma < 0.01$, $ops \approx 30$, hence $\gamma^{N_{less}} < \lambda^{N_{less}\times(1+ops)}$, making the second strategy, which simply spends ops time in each iteration the optimal one. As a result, the upper bound for $Pr[Win]$ becomes

$$(1 - 2^{l_a})^N \tag{4}$$

Hence, for a target upper bound of $Pr[Win]$, one can achieve this requirement, by setting

$$N = 2^{l_a} \times \ln\left(\frac{1}{Pr[Win]}\right) \tag{5}$$

Note that $s = 2^{l_a}$ is the size of the memory the checksum code needs to check (let us assume it is a full memory walk), so the number of iterations N scales linearly with the size of memory for random walk.

One may wonder why this is smaller than the folklore setting of using $O(s\ln(s))$ iterations, which is derived from the coupon collector problem so one meets the target probability for covering every single memory word. The reason is that, in the ASSW framework and our adapted version, one only cares about the memory word that has different value and does not post requirements about the coverage of all the other words. This weaken requirement is sufficient, if we assume attackers cannot move the different word around as the attestation process goes on. The difference in the target event under the ASSW framework and the coupon collector formulation, i.e., the difference of looking at a single memory address vs looking at all memory addresses, results in the difference of the $\ln(s)$ term.

Figure 4: An RTU device used in our experiment, which consists of 3 different types of boards.

Analyzing the overhead of the memory stride scheme. As derived above in Equation (5), given a constant target $Pr[Win]$, for a full memory walk over the $s = 2^{l_a}$ memory words in the whole memory, $N = s \times \ln(\frac{1}{Pr[Win]}) = O(s)$. For the memory stride scheme, the security analysis from the adapted ASSW framework largely applies, except that the size of the memory to be walked, i.e., 2^{l_a} now corresponds to two parts: the size of the memory that holds the SWORT code, which is L, and the size of all the memory stride addresses, which is $\frac{s}{L}$. Hence, the incurred overhead is

$$N = O\left(L + \frac{s}{L}\right) \tag{6}$$

which can be significantly smaller than the $O(s)$ overhead incurred by a full memory walk.

6 EVALUATION

We develop and evaluate our memory stride solution (see Algorithm 1) on two models of real-world power grid remote terminal unit (RTU) devices that are widely deployed today. The photo of one RTU is shown in Figure 4. As can be seen, it contains different types of boards, including a main processing board and different types of IO boards. Two types of ARM processors, i.e., ARM7 and Cortex-M3, are used in the different boards of these two RTU models. Our following evaluation will report our implementation and experiments on these two platforms.

Evaluation setup. The main processing boards on both RTU models use the LPC2292 chip with an ARM7 processor [31]. The processing board is capable of handling standard IEC 60870-5-104 communication protocols [20], collecting and processing measurements, as well as issuing various control commands. The I/O boards in the RTU shown in Figure 4 are equipped with the LPC1756 chip with an ARM Cortex-M3 processor [30]. Both chips have 16KB RAM.

Besides the checksum computation, our SWORT code also includes other code that needs to be executed to establish root of trust. In particular, the communication code to send the computed checksum back to the verifier. The communication logic on the processing board is handled by another chip W5100 [41], which implements a hardware network stack. Since it runs no software and claims to be resistant against network attacks, we consider it a trusted device in our experiment. Hence, the communication code is short, which invokes a very thin layer of w5100 device driver to

		10 nines assurance		*5 nines* assurance	
Device	Setting	Full RAM Walk	Memory Stride	Full RAM Walk	Memory Stride
LPC2292	ARM7 (60MHz, 16KB RAM)	52.1ms	13.2ms	26.1ms	6.7ms
LPC2362	ARM7 (72MHz, 58KB RAM)	156.9ms	11.0ms	78.5ms	5.6ms
LPC1756	Cortex M3 (100MHz, 16KB RAM)	27.6ms	7.0ms	13.9ms	3.6ms
LPC1788	Cortex M3 (120MHz, 96KB RAM)	137.0ms	5.9ms	68.6ms	3.0ms

Table 1: Time needed for establishing the Root-of-Trust for the baseline full RAM walk scheme and our memory stride scheme. Results on LPC2292 and LPC1756 (*10 nines*) are based on measurements; the rest are based on estimation. *10 nines* indicate 99.99999999% probability that an attacker will lose, while *5 nines* indicate 99.999%, respectively.

interact with registers provided by this chip. The LPC1756 device communicates with external parties through a CAN controller. The driver code needed to send messages through the CAN controller is also short. The overall code size (including the communication driver) is 1.8KB and 1.9KB on LPC1756 and LPC2292, respectively. The whole SWORT code (including checksum computation and communication) hence fits into the first 2KB of the RAM.

SWORT time reduction with memory stride. To demonstrate the efficiency of our memory stride solution, we compare it with a scheme that needs to conduct random walk over the whole RAM space —- the only solution that can stand previously described attacks on ARM platform (see Section 4.2). We conduct experiments on the LPC 2292 and LPC 1756 boards to determine the time needed for running the 2 schemes on them, and then estimate the time needed for another two ARM-based boards (LPC 2362 and LPC 1788) using their memory sizes and CPU frequency. The results are summarized in Table 1. As can be seen from the table, performing checksum over the entire RAM is significantly more expensive than using our memory stride scheme. In particular, the attestation time needed by full memory walk scheme ranges from 27.6ms to more than 150ms on the different boards for achieving an assurance of 10 nines. Here, *10 nines* indicate 99.99999999% probability that a malicious prover will not win the attestation. In comparison, the memory stride solution uses less than $20ms$ of checksum computation time on all platforms. When the memory size is large, e.g., on LPC1788, which is a variation of LPC1756 and contains $96KB$ of RAM, the gain of our solution over the full RAM scheme can be more than $20x$.

Network delay variance. In the following experiment, we assess the potential impact of network delay variance in a typical power grid substation network setup. Specifically, if the communication between the verifier and a genuine prover is delayed beyond the acceptable response time threshold, the prover cannot be attested successfully and this leads to a false positive result[5]. We evaluate our scheme in a testbed that emulates the environment in a real-world power grid substation that runs IEC 60870-5-104 protocol [20]. In particular, we setup a tested using two models of industrial Ethernet switches, i.e., Belden Spider II 8TX Ethernet switch [12] and Moxa EDS-205 Switch [29]. To emulate a large substation automation system, we use multiple switches and connect them in a daisy chain setting. We use Belden Spider II 8TX Ethernet switches in the first 5 hops and Moxa EDS-205 Switches in the next 5 hops. We also replay background traffic that is generated at a rate 10x faster than a sample IEC 60870-5-104 protocol trace we collected from a real-world electrical substation.

[5]We evaluate in a separate experiment that the computation time for SWORT is stable, with the maximum variation between different runs below 0.1% of the overall time.

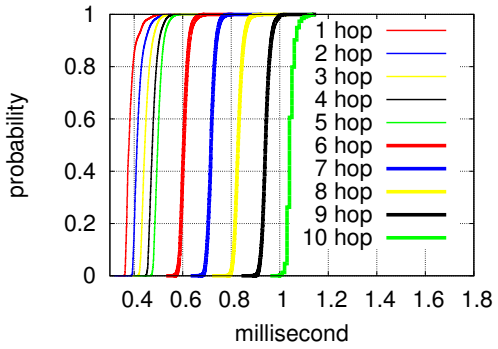

Figure 5: Cumulative Distribution Function (CDF) on round trip communication time in a substation network with up to 10 hops. The first 5 hops use Belden Spider II 8TX Ethernet switches and the last 5 Hops use Moxa EDS-205 switches.

Figure 5 plots the distribution of the end-to-end round trip time needed for the trusted verifier (we use Raspberry Pi in this experiment) and the prover (we use the RTU's main processing board). To focus on the variation in the communication delay, we let the prover immediately respond to the challenge without going through the memory stride phase. For each hop, we repeat the experiment by 500,000 times to get the distribution of the round trip time. As can be seen, the round trip time increases with the number of hops. The Belden switches introduce about 0.03 millisecond of extra delay per hop on average, while the Moxa switches introduce about 0.1 millisecond of extra delay per hop on average.

Recall that under a pessimistic assumption, the best possible attack that one may find against our memory stride scheme incurs 1.6% overhead of the computation time. If the checksum computation time is 12.5 millisecond, this translates to $1.6\% \times 12.5 = 0.2$ millisecond of extra delay allowance (or delay budget) while still allowing the verifier to detect such attacks. As shown in Figure 5, even with a single switch, the communication delay is above 0.3 millisecond, which is greater than the delay budget. Hence, the SWORT scheme needs to take the network topology into consideration and the allowed time for the checksum response should include both the computation time and the minimum communication time given the topology between the verifier and the prover. By doing so, the delay budget can be used to cater for the communication delay variation.

Now we examine the delay variation on each individual hop. For a given hop, we find that the probability for the delay to be 0.2 millisecond above the minimum delay measured in that same distance is less than 0.1%. Hence, with a delay budget of 0.2 millisecond, the false positive rate can be kept lower than 0.1%. The false positive

rate increases with reduced delay budget. For example, when the delay budget is 0.1 millisecond, the false positive rate can go beyond 10%. This implies that we may need to keep the computation time at 12.5ms (by choosing a suitable number of iterations), even if a smaller number of iterations is already sufficient to achieve the 10 nines assurance as shown in the analysis in Section 5 and the experimental results earlier.

Towards real-world system integration. We have integrated our solution into a substation automation system testbed, which consists of one HMI machine, several RTUs, electrical meters, relay units, and transformers. Meter readings are monitored continuously for real-time automatic control of the relay units and the transformers. The control decisions need to be made in a few AC cycles. This poses strict requirement on attestation time. We use the HMI machine (with a secure hardware) as the trusted verifier to attest the RTUs. We validate that our scheme achieves the same performance results (in term of both computation and communication delay) in this testbed as that reported earlier in our own experimental setup. We are now working with the RTU vendor and a power grid company to conduct trials in selected low-tension electrical substations.

7 CONCLUSION

In this work, we devise a practical software-based solution to attest ARM-based ICS devices. Our memory stride technique provides a secure partial memory walk solution that can significantly reduce the attestation duration on ARM device. We adapt a formal framework proposed by Armknecht et al. [7] to analyze its security and performance. Our experiments demonstrate that our solutions can attest real-world ARM-based ICS devices significantly faster and can meet stringent requirement by some time-critical power grid applications.

ACKNOWLEDGMENT

This research is partly supported by the National Research Foundation, Prime Minister's Office, Singapore under the Energy Programme and administrated by the Energy Market Authority (EP Award No. NRF2014EWT-EIRP002-040) and in part by the research grant for the Human-Centered Cyber-physical Systems Programme at the Advanced Digital Sciences Center from SingaporeâĂŹs Agency for Science, Technology and Research (A*STAR). We also thank Bin Zhou from Advanced Digital Sciences Center and our collaborators from Mirai Electronics Pte. Ltd. for their contributions to this paper by implementing the end-to-end attestation code on Mirai's RTUs.

REFERENCES

[1] Stuxnet. https://en.wikipedia.org/wiki/Stuxnet.
[2] IEEE Standard 1646, Communication Delivery Time Performance Requirements for Electric Power Substation Automation. 2005.
[3] Ali Abbasi and Majid Hashemi. Ghost in the plc: Designing an undetectable programmable logic controller rootkit. In *Black Hat Europe*, 2016.
[4] ARM. ARM Developer Suite Assembler Guide - 4.3.1. Flexible second operand. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/CIHBEAGE.html.
[5] ARM. Cortex-M Processor Family. http://www.arm.com/products/processors/cortex-m/index.php.
[6] ARM. Trustzone. http://www.arm.com/products/processors/technologies/trustzone/.
[7] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. A security framework for the analysis and design of software attestation. ACM CCS, 2013.
[8] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann. SEDA: Scalable Embedded Device Attestation. ACM CCS, 2015.
[9] M. Assante. Confirmation of a Coordinated Attack on the Ukrainian Power Grid. https://ics.sans.org/blog/2016/01/09/confirmation-of-a-coordinated-attack-on-the-ukrainian-power-grid, 2016.
[10] Atmel. SMART ARM Processor Based MCUs. http://www.atmel.com/products/microcontrollers/ARM/default.aspx.
[11] Z. Basnight, J. Butts, J. Lopez Jr., and T. Dube. Firmware modification attacks on programmable logic controllers. 6(2):76–84, 2013.
[12] Belden. SPIDER II Unmanaged Switches. https://www.belden.com/products/industrialnetworking/unmanagedswitches/spider-2.cfm.
[13] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *CCS*, 2009.
[14] A. Cui, M. Costello, and S. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013.
[15] K. E. Defrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. NDSS, 2012.
[16] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik. A minimalist approach to remote attestation. DATE, 2014.
[17] J. Franklin, M. Luk, J. M. McCune, A. Seshadri, A. Perrig, and L. van Doorn. Remote detection of virtual machine monitors with fuzzy benchmarking. *SIGOPS Oper. Syst. Rev.*, 42(3), April 2008.
[18] J. Franklin, M. Luk, A. Seshadri, and A. Perrig. PRISM: Enabling Personal Verification of Code Integrity, Untampered Execution, and Trusted I/O on Legacy Systems or Human-Verifiable Code Execution. Technical Report CMU-CyLab-07-010, CyLab, February 2007.
[19] Delta Group. Svg2000 series. http://www.deltaww.com/products/CategoryListT1.aspx?CID=060503&PID=761&hl=en-us&Name=SVG2000%20Series.
[20] IEC. Internal standard 60870-5-104. https://webstore.iec.ch/preview/info_iec60870-5-104%7Bed2.0%7Den_d.pdf.
[21] Texas Instruments. Smart grid leverages ARM-based solutions to enable intelligent power consumption with a more robust end-to-end communication network.
[22] Intel. Trusted compute pools with intel® trusted execution technology. http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/malware-reduction-general-technology.html.
[23] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. IEEE S&P, 2012.
[24] Sandia National Laboratories. Control System Devices: Architectures and Supply Channels Overview. http://energy.sandia.gov/wp-content/gallery/uploads/JCSW_Report_Final.pdf.
[25] Y. Li, Y. Cheng, V. Gligor, and A. Perrig. Establishing software-only root of trust on commodity systems: Facts and fiction. Cambridge International Workshop on Security Protocols, 2015.
[26] L. Martignoni, R. Paleari, and D. Bruschi. Conqueror: Tamper-proof code execution on legacy systems. DIMVA, 2010.
[27] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. Eurosys, 2008.
[28] A. McDiarmid. ARM immediate value encoding. http://alisdair.mcdiarmid.org/arm-immediate-value-encoding/, 2014.
[29] Moxa. EDS-205/EDS-208 Series. http://www.moxa.com/product/EDS-208205.htm.
[30] NXP. Lpc 1756. www.nxp.com/documents/data_sheet/LPC1759_58_56_54_52_51.pdf.
[31] NXP. Lpc 2292. www.nxp.com/documents/data_sheet/LPC2292_2294.pdf.
[32] D. Peck and D. Peterson. Leveraging ethernet card vulnerabilities in field devices. In *SCADA Security Scientific Symposium*, 2009.
[33] Adrian Perrig and Leendert van Doorn. Refutation of "on the difficulty of software-based attestation of embedded devices", April 2010.
[34] A. Seshadri, M. Luk, A. Perrig, L. Van Doorn, and P. Khosla. Using FIRE and ICE for detecting and recovering compromised nodes in sensor networks. Technical report, Carnegie Mellon University, December 2004.
[35] A. Seshadri, M. Luk, A. Perrig, L. van Doorn, and P. Khosla. Scuba: Secure code update by attestation in sensor networks. WiSe, 2006.
[36] 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. SOSP, 2005.
[37] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: software-based attestation for embedded devices. IEEE S&P, 2004.
[38] A. Shah, A. Perrig, and B. Sinopoli. Mechanisms to Provide Integrity in SCADA and PCS devices. CPS-CA, 2008.
[39] R. Spenneberg, M. Bruggemann, and H. Schwartke. PLC-blaster: A worm living solely in the PLC. In *Black Hat Asia*, 2016.
[40] P. Traynor, K. Butler, W. Enck, P. McDaniel, and K. Borders. Malnets: Largescale malicious networks via compromised wireless access points. 3(2-3):102–113, 2010.
[41] WIZnet. http://www.wiznet.co.kr/product-item/w5100/.

Appendix

	Meaning	Representative Value
p	primary memory size (in words)	16 (directly accessible registers)
s	secondary memory size (in words)	4K words for a 16KByte memory
l_a	length of effective secondary memory address (in bits)	$s = 2^{l_a}$
λ	the fraction of memory words that are identical in memory state S and $\tilde{S}$	$1 - \frac{1}{2^{l_a}}$
l_g	length of pseudo random seed, in bits	32
l_r	length of the checksum, in bits	32×12
l_s	length of a state entry (i.e., a memory word), in bits	32
Σ	the set of possible state entries (memory words)	$O(2^{l_a})$
γ	state-incompressibility parameter	$max_{x \in \Sigma} D_s(x)$
N	number of iterations for memory accesses	$O(s)$ (see Equation 5)
ϱ	time-bounded pseudo-randomness of Gen	0
ν_{Gen} and ν_{Chk}	time-bounded unpredictability of Gen and unpredictability of Chk^N	0
ω	Blind second pre-image resistance for Chk	$\frac{1}{2^{l_r}}$

Table 2: Main notations used in the ASSW framework.