SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust

Karim El Defrawy
UC Irvine
keldefra@uci.edu

Aurélien Francillon
Institute EURECOM
francill@eurecom.fr

Daniele Perito
INRIA
perito@inrialpes.fr

Gene Tsudik
UC Irvine
gene.tsudik@uci.edu

Abstract

Remote attestation is the process of securely verifying internal state of a remote hardware platform. It can be achieved either statically (at boot time) or dynamically, at run-time in order to establish a dynamic root of trust. The latter allows full isolation of a code region from preexisting software (including the operating system) and guarantees untampered execution of this code. Despite the untrusted state of the overall platform, a dynamic root of trust facilitates execution of critical code. Prior software-based techniques lack concrete security guarantees, while hardware-based approaches involve security co-processors that are too costly for low-end embedded devices.

In this paper, we develop a new primitive (called SMART) based on hardware-software co-design. SMART is a simple, efficient and secure approach for establishing a dynamic root of trust in a remote embedded device. We focus on low-end microcontroller units (MCU) that lack specialized memory management or protection features. SMART requires minimal changes to existing MCUs (while providing concrete security guarantees) and assumes few restrictions on adversarial capabilities. We demonstrate both practicality and feasibility of SMART by implementing it – via hardware modifications – on two common MCU platforms: AVR and MSP430. Results show that SMART implementations require only a few changes to memory bus access logic. We also synthesize both implementations to an 180nm ASIC process to confirm its small impact on MCU size and overall cost.

1. Introduction

Verifying internal state of a remote embedded device is an important task in many scenarios and application settings, e.g., smart meters, implantable medical devices (IMDs) and actuators in industrial control systems that perform critical functions and operate unattended for long periods of time. In addition, increasing adoption of wireless networking prompts concerns about remote exploits of such devices. The recent Stuxnet worm [15] demonstrated the magnitude of damage from attacks on embedded devices. Stuxnet infected Programmable Logic Controllers (PLC) used in industrial control systems and caused considerable physical damage by modifying their control software. Embedded devices are also sometimes placed in physically inaccessible locations, e.g., IMDs, military or industrial sensors and actuators. In such settings, it is hard to physically connect to an external interface to verify the state of a target device.

The discussion above motivates the need for *attestation* techniques to detect, and possibly disable, malicious code prior to performing critical operations. Current attestation methods fall somewhat short of meeting requirements for a wide range of embedded devices. They generally fall into two extremes on the design spectrum: *hardware-* and *software-based* techniques. The former rely on specialized hardware (e.g., a TPM [6]) or on the availability of special CPU instructions [24] to perform attestation, either statically (at boot time) or dynamically, during normal run-time operation. These techniques have attracted a lot of attention from both the research community and industry. They are best-suited for higher-end devices, such as laptops and smart-phones. Experimental devices that include a full TPM as a separate chip have been constructed [23]. However, this approach cannot provide a dynamic root of trust and is expensive [1] for low-end devices.

Several software-based attestation methods have been proposed for commodity [37] and embedded

1. The cost of a TPM chip is close to that of a low-end MCU.

devices [38], [36], [25], [35], [47]. However, they generally offer uncertain security guarantees [41] and some have been subject to attacks [8]. Furthermore, all current software-based techniques involve restrictive assumptions on adversarial capabilities that make them unsuitable for many realistic applications. In particular, they typically assume "adversarial silence", meaning that, during each attestation process, only the intended prover (device being attested) is communicating with the verifier (entity that performs attestation). In other words, even though the prover might have malware installed, it is not aided – or impersonated – by any external party during attestation. The same assumption is sometimes referred to as "no collusion". Any attestation technique that makes this assumption is limited to close-range (one-hop) communication between the prover and the verifier and its security often relies on strict round-trip time measurements. It is easy to see that the adversarial silence assumption is necessary as long as no secret information can be maintained on the prover. Maintaining secrets, however, requires secure storage, which, in turn, prompts the need for hardware support.

Software-based attestation also assumes that the adversary impersonating (or colluding with) the prover must use the same hardware as the genuine prover. While this assumption might hold in a few specific settings, it is unrealistic for many applications.

Finally, there are some proprietary techniques for embedded processors currently on the market. For example, ARM TrustZone [3] provides an additional – secure – processor mode of execution. It includes a new set of *shadow* registers, a few KBytes of on-chip SRAM [2], and allows controlling access to peripherals by the operating system. Though TrustZone inherently relies on secure boot, it can be used to provide a dynamic root of trust [26], [12]. However, it targets more powerful devices than those considered in this paper[3].

The problem with the static root of trust is that, in general, it does not offer any guarantees about the *current* state of a device, since adversarial exploits can occur post-boot. Even worse, a static root of trust (e.g., TPM v1.1 or Secure Boot) is unsuitable for detecting a powerful attack class based on Return-Oriented Programming (ROP) [39]. ROP allows execution of an arbitrary return-oriented program by merely manipulating the return addresses on the stack, i.e., without changing code. In order to detect such attacks, techniques that do not rely on the code isolation provided by a dynamic root of trust have to check areas of memory that are highly volatile, e.g., stack and heap.

2. In contrast with our target MCUs, most devices with TrustZone do not all include RAM and Flash on chip.

3. TrustZone is available on the high-end ARM processors (ARM11 and Cortex-AX series). However, to the best of our knowledge, it is unavailable for low-end ARM devices that correspond to MCU-s we focus on, e.g., ARM Cortex-M1. Low-end ARM cores with security extensions are known as SecureCore. However, no detailed information is publicly available about them [4].

1.1. Roadmap

In this paper, we stay clear of both efficient-but-limited software-based techniques and heavy-weight TPM-based approaches to attestation. We focus on the design space area that has not been previously explored by utilizing a software/hardware co-design approach to architect an attestation mechanism with minimal hardware requirements.

Our main design guideline is to carefully justify each component necessary to achieve secure establishment of a dynamic root of trust in a remote embedded device. Following this guideline leads us to an approach – called: SMART: **S**ecure and **M**inimal **A**rchitecture for (Establishing a Dynamic) **R**oot of **T**rust – that entails *minimal* hardware modifications to current embedded MCU-s. To the best of our knowledge, this represents the first minimal hardware solution for establishing a dynamic root of trust in low-end embedded devices. We implemented it on two widely available low-cost MCU platforms: Atmel AVR and Texas Instruments MSP430, by modifying open-source implementations of these MCU-s in VHDL and Verilog. (Both obtained from the OpenCores Project [31]).

Organization: Section 2 discusses our goals and building blocks. Then, Section 3 presents SMART details and features. Security issues are addressed in Section 4. Next, Section 5 presents several concrete protocols utilizing SMART as a primitive. Implementation details are discussed in Section 6 and related work in Section 7.

2. Goals and Design Elements

The main result of this paper is the development of a new primitive called: SMART: **S**ecure and **M**inimal **A**rchitecture for (Establishing a Dynamic) **R**oot of **T**rust. SMART is executed by the prover, $\mathcal{PRV}$, and, in doing so, attests a region of code and jumps to it. A proof of execution is computed and sent to the verifier, $\mathcal{VRF}$. SMART guarantees that attested code is executed even if the entire prover system is

compromised (except SMART ROM code). In the rest of this section, we describe our security objectives, adversarial assumptions and SMART's main building blocks.

2.1. Security Objectives

SMART has three security objectives based upon successful completion of the attestation protocol:

- *Prover Authentication:* $\mathcal{VRF}$ obtains entity authentication of $\mathcal{PRV}$.
- *External Verification:* $\mathcal{VRF}$ is assured that memory segment $[a, b]$ on $\mathcal{PRV}$ contains the expected content.
- *Guaranteed Execution:* $\mathcal{VRF}$ is assured that code at location x was executed by $\mathcal{PRV}$.

2.2. Adversarial Assumptions

We assume that the adversary, $\mathcal{ADV}$, has complete control over the software state, code and data of $\mathcal{PRV}$ before and after SMART execution. In particular, $\mathcal{ADV}$ can modify any writable code on $\mathcal{PRV}$ and learn any secret that is not explicitly protected by the MCU on $\mathcal{PRV}$. Furthermore, $\mathcal{ADV}$ has complete control over the communication channel and – during the protocol – can use multiple colluding devices in order to pass or subvert attestation.

We also assume that $\mathcal{ADV}$ does not perform hardware attacks on $\mathcal{PRV}$. Specifically, it does not alter code stored in ROM, induce hardware faults or retrieve $\mathcal{K}$ using external side-channels. Likewise, $\mathcal{ADV}$ has no means of interrupting execution of ROM-resident code on $\mathcal{PRV}$.

Protection against hardware-based attacks could be added by encasing the MCU in tamper-resistant coating and employing standard techniques to prevent side-channel key leakage. Since our approach is confined to the MCU, employing such techniques is quite natural. Furthermore, hardware attacks could be mitigated using well-known tamper-resistance techniques, such as anomaly detection, internal power regulators and additional metal layers or meshes for tamper detection.

Some processor peripherals might be capable of modifying memory without interaction with the MCU core, e.g., a DMA engine. We assume that such peripherals can be disabled during SMART execution.

Finally, we assume that $\mathcal{PRV}$ and $\mathcal{VRF}$ share a secret key $\mathcal{K}$. This key can be pre-loaded onto $\mathcal{PRV}$ at production time or later. We do not address the details of this procedure.

2.3. Building Blocks

Our design relies on four main components that reside on $\mathcal{PRV}$:

- *Attestation Read-Only Memory:* Memory region in ROM inside the MCU. The key $\mathcal{K}$ can only be accessed from this region.
- *Secure Key Storage:* Memory region inside the CPU; it can be accessed only from SMART code in ROM.
- *MCU Access Controls:* Controls access to $\mathcal{K}$ and prevents non-SMART code from accessing it.
- *Reset and Memory Erasure:* If any error is reported by the above components, a hardware reset of the MCU is performed. Upon reset, hardware enforces a memory cleanup.

We argue that these four components are both necessary and sufficient for building a dynamic root of trust in a low-end embedded system. We detail their purpose in the next section.

2.4. SMART Overview

As discussed above, the central goal of SMART is guaranteed execution of a piece of code on the prover ($\mathcal{PRV}$) to an external verifier ($\mathcal{VRF}$), even when the prover is fully compromised. SMART relies on a challenge-based protocol – initiated by $\mathcal{VRF}$ – that leverages special hardware features of $\mathcal{PRV}$. At the start of SMART (Figure 1), $\mathcal{VRF}$ sends several parameters to $\mathcal{PRV}$: attestation region boundaries a and b; address x where $\mathcal{PRV}$ optionally passes control after attestation if x_{flag} is set; and nonce n to prevent replay attacks. A ROM-resident code segment on $\mathcal{PRV}$ computes a cryptographic checksum C of a region $[a, b]$ in $\mathcal{PRV}$'s memory (using nonce n) and then passes control to x. After execution of code starting at x, $\mathcal{PRV}$ returns C to $\mathcal{VRF}$. The latter verifies correctness of C by re-computing it using the same parameters and $\mathcal{K}$. We refer to ROM-resident code as $\mathcal{RC}$ and code optionally executed thereafter – as $\mathcal{HC}$. The sequence of operations of SMART is shown in Figure 1 and the corresponding pseudo-code of $\mathcal{RC}$ is illustrated in Algorithm 1.

We note that a non-keyed function, such as a cryptographic hash (e.g., SHA-256), is unsuitable for attestation. This is because, without a secret key, anyone can compute a hash of any input and fake a reply by $\mathcal{PRV}$. In particular, malware that infected $\mathcal{PRV}$ can do so. Therefore, our cryptographic checksum is implemented as HMAC keyed with $\mathcal{K}$ that resides

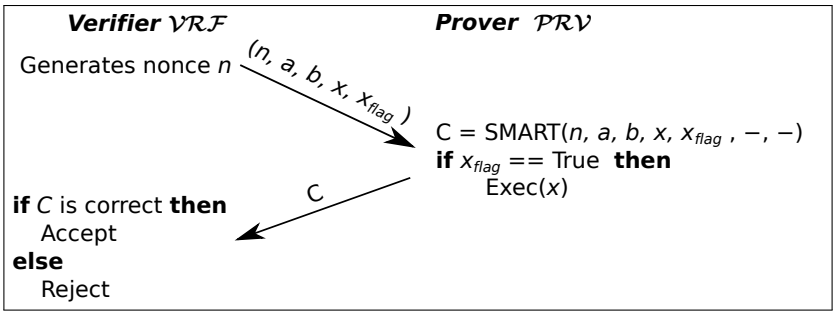


Figure 1: Overview of Protocol Using SMART.

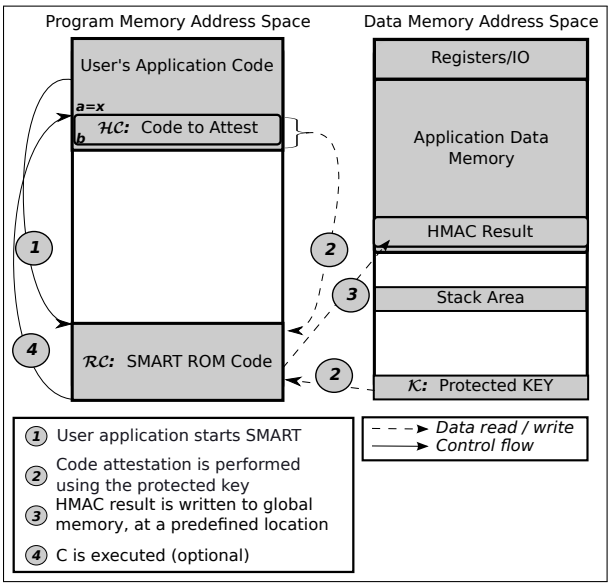


Figure 2: SMART Operation Overview.

in secure storage on $\mathcal{PRV}$'s MCU [4]. Usage of, and access to, $\mathcal{K}$ is restricted by the MCU such that only (trusted and immutable) $\mathcal{RC}$ is allowed to use it. For its part, $\mathcal{RC}$ only uses $\mathcal{K}$ to compute HMAC and then passes control to $\mathcal{HC}$. $\mathcal{RC}$ is instrumented using both static and dynamic analysis tools to prevent accidental leakage of $\mathcal{K}$.

In addition, when x_{flag} is set, interrupts remain disabled after execution of $\mathcal{RC}$. This is to ensure that $\mathcal{HC}$ is subsequently executed, and to prevent *Time-of-check-to-time-of-use* (TOCTTOU) attacks. A TOCTTOU attack could entail installing a malicious interrupt handler and scheduling an interrupt (e.g., a timer) to occur during the first instructions of $\mathcal{HC}$. Such an interrupt handler could allow reading or writing memory between executions of $\mathcal{RC}$ and $\mathcal{HC}$. Furthermore, hardware modifications to the MCU are added to avoid code reuse attacks.

4. If minimality was not a primary goal, public key cryptography could be used to improve key management.

3. SMART in Detail

This section describes, in detail, features and components of SMART.

3.1. Attestation ROM

ROM is a standard feature in many commodity MCUs. Generally, it incurs very little overhead in the design and construction of the MCU, since it constitutes a cheap form of storage. Typically, ROM is hardwired during manufacturing, rendering it immutable. What makes our attestation ROM special is its exclusive hardware-enforced ability to access $\mathcal{K}$.

ROM Code. $\mathcal{RC}$ must guarantee the following properties:

1) *Key Isolation:* $\mathcal{K}$ must not be leaked from ROM.
2) *Memory Safety:* Software bugs should not allow temporary memory exposure or $\mathcal{K}$ leakage.
3) *Atomic Execution:* ROM code must be executed atomically and cannot be invoked partially.

Property (3) is guaranteed by the MCU, as discussed in Section 3.3 below. Properties (1) and (2) are guaranteed by using two code instrumentation tools: `CQUAL` [19] and `Deputy` [11].

As shown in Algorithm 1, SMART computes an HMAC of a particular memory segment and then jumps – without being interrupted – to a verifier-specified address within that segment. The implementation consists of approximately 500 lines of C code, which makes checking its correctness both feasible and relatively easy.

Algorithm 1: SMART code in ROM.

input : a, b start/end addresses for attestation
x address to jump to after attestation
x_{flag} jump or not?
n nonce sent by verifier
out output address where to store checksum
in (optional) input parameter
output: HMAC output
begin
  /* Disable interrupts during SMART code execution */
  `DisableIRQ();`
  /* Attestation key K is unlocked automatically by the MCU */
  `InitHmac(`K`);`
  /* Attest all parameters */
  `HmacProcess(`$a||b||x||x_{flag}||n||in||out$`);`
  /* Attest memory region $[a, b]$ */
  for $i \in [a, b]$ **do**
    `HmacProcess(`$Mem[i]$`);`
  end
  $C \longleftarrow$ `FinishHmac();`
  /* Store HMAC result in global variable */
  `Copy(`$*out, C$`);`
  /* Erase temporary variables */
  `ResetMemory();`
  if $x_{flag} = True$ **then**
    /* If execute flag set, exec function at address x */
    `Call(`x, in`);`
  else
    /* Restore interrupts status as before SMART exec*/
    `RestoreIRQ();`
  end
end

Key Secrecy. Upon termination, SMART passes control to the untrusted portion of $\mathcal{PRV}$, where malicious code can sift through memory and search for traces of $\mathcal{K}$ or intermediate states used in HMAC computation. This could lead to disclosure of $\mathcal{K}$. For this reason, we instrumented SMART code with `CQUAL` – a tool that detects information leakage in C programs. Specifically, $\mathcal{K}$ is marked with a `SECRET` type. `CQUAL` propagates this type to each variable that is computed with any involvement of any other variable of type `SECRET`. Each function is equipped with a check for leakage of any `SECRET` variable. `CQUAL` instrumentation is performed off-line; it does not incur any overhead during operation of SMART. The end-result is simple: each variable marked `SECRET` by `CQUAL` is zeroed out at the end of each function. The only variables not erased are the outputs of each function. Also, the memory location of $\mathcal{K}$ is no longer accessible upon completion of SMART.

Memory Safety. Key isolation alone does not prevent key leakage, since our code could contain vulnerabilities that allow $\mathcal{ADV}$ to retrieve $\mathcal{K}$ by running SMART on malicious (or malformed) inputs. Fortunately, SMART involves only around 500 lines of code. This relatively small size allows manual inspection for memory corruption bugs. We also enhance manual inspection using `Deputy` – a C compiler based on GCC, that provides an annotation language for describing memory boundaries in C. For example, a C array can be augmented with information about its size. The compiler adds instructions to check all memory accesses to the array and detects memory corruptions. Once SMART code is reinforced with `Deputy`, whenever a memory corruption is detected, a special reset is performed by `Deputy` instrumentation code. As for other error conditions that could cause a reset, we deal with them by making sure that, at each reset, all memory (stack, heap, and registers) is erased.

Furthermore, the stack and *out* pointers might be controlled by $\mathcal{ADV}$ when SMART code is called. If invalid values are provided [5], memory corruption may occur during SMART execution. This could be exploited by $\mathcal{ADV}$ to abuse SMART, e.g., recover bits of $\mathcal{K}$ or skip execution of important code. Therefore, both stack pointer and *out* pointers are checked at the beginning of SMART code.

Side-Channels. Another avenue for $\mathcal{ADV}$ to extract $\mathcal{K}$ is via side-channel attacks. Since hardware side-channels are out of scope of this work, we focus on software side-channels, i.e., malware on $\mathcal{PRV}$ trying to learn $\mathcal{K}$ by observing SMART execution. Low-end MCUs (such as MSP430 or AVR) do not have caches that could be used for timing attacks based on hits and misses. Also, differences in execution time due to bus contention are data-independent and cannot leak $\mathcal{K}$ [6]. Finally, a software-only timing side-channel attack against HMAC-SHA used in SMART

5. For example, a stack pointer that points to an invalid memory region, such as I/O register space. Or, the *out* pointer pointing to the stack itself, leading to corruption of the stack region used by SMART when the HMAC result is written to it.

6. There is no bus contention on AVR due to its Harvard architecture. On MSP430, we manually verified rare cases of bus contention. On other processors, wait cycles can be added to address this issue (only needed when executing SMART code).

is not viable. Code used for HMAC computation does not have conditional branching instructions, resulting in constant execution time. Moreover, to the best of our knowledge, no timing attacks have been reported against HMAC-SHA.

3.2. Secure Key Storage

The next question is where to store $\mathcal{K}$ used for computing HMAC. Clearly, it cannot be stored in normal memory, since malware could easily access it and pass attestation. We use a special hardware-controlled memory location to house a single symmetric key, $\mathcal{K}$. This storage must be immune to software attacks. Recall that hardware attacks are out of scope of this work. We also note that hardware attacks require direct physical access or at least very close physical proximity to the target device. This is improbable in many access-restricted settings, e.g., manufacturing plants, utility stations, fabrication labs or implantable medical devices (IMDs).

Although details of $\mathcal{K}$ initialization are not discussed in this paper, there are at least two viable approaches. In the first, $\mathcal{K}$ is hard-coded at production time and never changed again, i.e., in addition to be being access-restricted, $\mathcal{K}$ storage location is read-only. Alternatively, there could be a secure means of *modifying*, but not reading, $\mathcal{K}$ by an authorized party (e.g., the verifier) that would rely on a special authenticated channel.[7]

3.3. MCU Access Controls

Simplicity and minimal cost are some of the primary objectives of SMART. Hardware modifications are limited to memory access checking and availability of ROM. We now describe the hardware modifications necessary to enforce key protection and to restrict execution of $\mathcal{RC}$.

Key Access Controls. To enforce $\mathcal{K}$ secrecy we need to ensure that it can be accessed only when the program counter (PC) is in the $\mathcal{RC}$ memory region. One simple method to enforce this is to connect the data bus to $\mathcal{K}$ memory when the program counter is in ROM range and the data address is pointing to $\mathcal{K}$ address range. The internal reset signal is triggered if $\mathcal{K}$ memory is accessed while the program counter is not in ROM range. Figure 3 shows how access to $\mathcal{K}$ is controlled in the MCU.

7. This topic is deferred to future work.

ROM Execution Control. Since $\mathcal{RC}$ is authorized to access $\mathcal{K}$, its usage must be controlled to prevent recovery by malware. For example, $\mathcal{ADV}$ can attempt to selectively execute portions of $\mathcal{RC}$ by using code reuse techniques (e.g., return to libc [42], borrowed code chunks [27] or Return-Oriented Programming [39], [7], [9]). To prevent such attacks, we provide additional access controls upon $\mathcal{RC}$ entry and exit. The program counter is only allowed to move into ROM starting at SMART initial address. Similarly, the program counter can leave ROM only from the last SMART address. These controls ensure that $\mathcal{RC}$ cannot be invoked partially: once any attempt to do otherwise is detected, the MCU is immediately reset. This necessitates for $\mathcal{RC}$ to be compiled such that any valid termination of SMART execution is ended by a return from the its last instruction address.

3.4. Cleaning Memory on Reset

When an invalid operation takes place, such as an attempt to violate SMART memory access controls, a hardware exception occurs, leading to an immediate MCU reset. However, if SMART code does not terminate properly, it cannot clean up its working memory and keying material could remain in memory after reset. (The situation is similar if a power loss occurs.) This technique was used in several attacks on MCUs to recover keying material or store information across resets [20], [21]. Therefore, it is mandatory to perform memory cleanup upon each reset. In SMART, memory cleanup is performed by processor logic triggered upon every boot or reset.

We note that the aforementioned phenomenon is similar to cold boot attacks [22] whereby a computer is stopped during execution and its memory is removed in order to recover keying material. However, since a typical MCU features processor and memory in a single "package", the latter cannot be accessed directly. If debugging interfaces are permanently deactivated and memory is freed upon each reset, only hardware attacks (that are out of scope of SMART) would allow recovery of parts of memory.

4. Security Analysis

Our present security argument is informal. A more substantial argument (or a proof) would require formal analysis and verification of SMART code, which is planned as part of future work. The security argument is based on the following assertions:

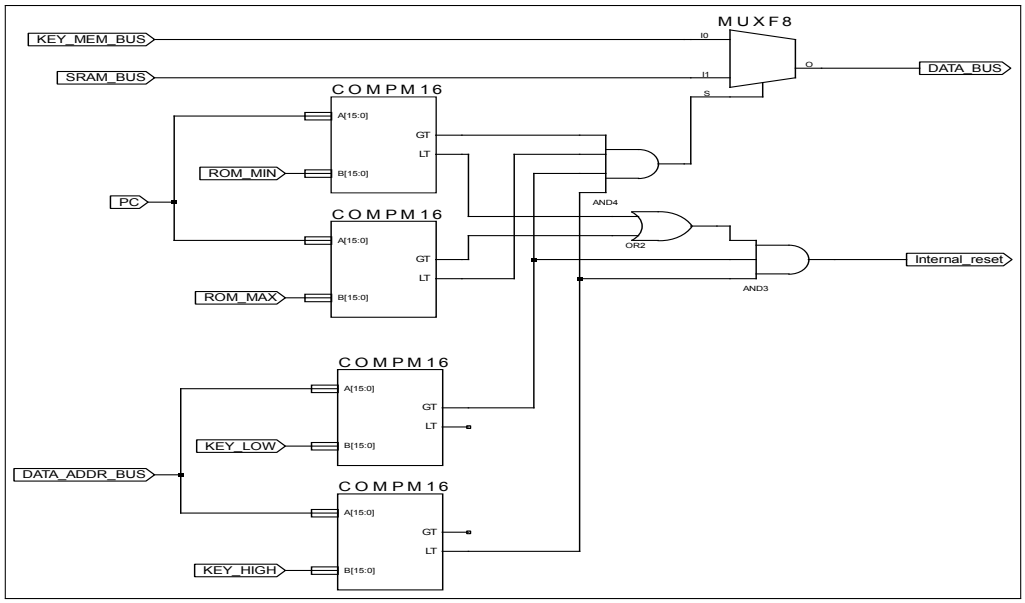


Figure 3: Schematic view of access control for attestation key.

A1 Cryptographic checksum C computed by $\mathcal{PRV}$ cannot be forged. Since C is a result of secure HMAC function (e.g., HMAC-SHA) we assume that, for any $\mathcal{ADV}$ – external to $\mathcal{PRV}$ – that observes a polynomial number of such checksums, finding HMAC collisions and/or learning bits of the attestation key is infeasible.

A2 Physical and hardware-based attacks on $\mathcal{PRV}$ are beyond $\mathcal{ADV}$'s capabilities.

A3 Attestation key $\mathcal{K}$ can be accessed only from within ROM-resident SMART code. This is guaranteed by MCU-based access controls.

A4 SMART code cannot be modified since it resides in ROM.

A5 SMART code can be only invoked at its beginning. The hardware checks that, except for the very *first* instruction in $\mathcal{RC}$, if the program counter is in $\mathcal{RC}$ range, then the previous executed instruction must also be in ROM.

A6 $\mathcal{RC}$ execution can only terminate at the very last instruction address in $\mathcal{RC}$. The hardware checks that, except for the very *last* instruction in $\mathcal{RC}$, if the program counter is not in $\mathcal{RC}$ range, then the previous instructions must also be outside $\mathcal{RC}$ range.

A7 Upon each invocation of SMART, all interrupts are disabled [8] and remain so if, upon completion of SMART, control is passed to $\mathcal{HC}$.

A8 $\mathcal{K}$ cannot be extracted by any software-based $\mathcal{ADV}$ internal to $\mathcal{PRV}$. Upon completion of SMART execution, $\mathcal{K}$ is no longer accessible. Also, all memory used by SMART code is securely erased. The only value based (statistically dependent) on $\mathcal{K}$ is the output C.

A9 For each invocation, SMART computes C based on the contents of the requested memory segment $[a, b]$. Although C is guaranteed to be computed correctly, it may or may not result in $\mathcal{PRV}$ passing attestations, since $[a, b]$ might be previously corrupted by $\mathcal{ADV}$.

A10 Any erroneous state (e.g., violation of assertions A3, A5, A6) leads to a hardware reset. Upon reset, all data memory and registers are erased, which prevents $\mathcal{K}$ leakage. This boot-time memory erasure also guarantees that, if power loss occurs during SMART execution, no information about $\mathcal{K}$ is retained in memory.

A11 Observing normal execution of SMART should leak no information about $\mathcal{K}$. Therefore, SMART execution time and amount of memory used must not be key-dependent.

8. From the security perspective, executing SMART with interrupts disabled is redundant with respect to assertions A5 and A6. However, this (assertion A7) prevents a reset if an interrupt occurs during SMART execution, thus improving reliability.

Key Protection Guarantee. Assertion A3 implies that $\mathcal{K}$ is not directly available to untrusted software. Assertions A5 and A6 guarantee that code reuse attacks to recover $\mathcal{K}$ are impossible. A10 implies that, when error condition occurs, execution is stopped and no information about $\mathcal{K}$ is leaked. A11 guaranties that side-channels cannot be used to gather information about $\mathcal{K}$ by untrusted software executing on the MCU. Other side-channels commonly used in key recovery attacks rely on power consumption analysis and electromagnetic emanations [34]. However, these are hardware/physical attacks [9].

Given the above assertions and the key protection guarantee and assuming that $\mathcal{VRF}$ receives and successfully verifies C, we argue that postulated security objectives are satisfied:

Prover Authentication. If C is correctly computed and n is a random nonce of sufficient bit-length, $\mathcal{VRF}$ concludes that C was computed by $\mathcal{PRV}$ within the interval of time between the initial request message and the receipt of C. This yields fresh authentication of $\mathcal{PRV}$.

External Verification. Assertions A1-A8 imply that C was computed by SMART code on $\mathcal{PRV}$. Therefore, memory region $[a, b]$ on $\mathcal{PRV}$ contained code or data expected by $\mathcal{VRF}$.

Guaranteed Execution. Assertion A6 implies that, immediately after computing C, $\mathcal{PRV}$ executes code at x, if x_{flag} is set. If C is deemed correct by $\mathcal{VRF}$ and $x = a$, $\mathcal{VRF}$ is assured that the expected code at location a was executed.

5. Other Uses of SMART

In this section we describe several techniques that can be implemented using SMART as a building block.

5.1. Remote Attestation of Parts of Memory

The most natural usage of SMART is to attest a memory segment and verify that it contains data (or code) that it is expected to contain. This can be achieved by invoking SMART with the start and end addresses of the memory range to be attested, as shown in Algorithm 2.

9. We note that these side-channels might be exploitable in very specific cases by a local attacker, e.g., if hardware to perform such measurements is available as a peripheral of the device, e.g., a coulomb counter that measures remaining battery power. This could, in theory, provide information on power consumption of SMART code. We assume that such features are not available on the device.

Algorithm 2: SMART usage to attest a memory range.

input : n nonce sent by $\mathcal{VRF}$
a start address to attest
b end address to attest
H HMAC result (global variable)
output: HMAC output
begin
SMART (a, b, $\emptyset$, $False$,n,&H,$\emptyset$) ;
Send(H);
end

5.2. Remote Proof of Reset

Some applications need to ensure that a device has been reset successfully. This can be easily done with SMART, as shown in Algorithm 3. HMAC guarantees that the reset function (R) has been verified and executed. Here, we assume that output of HMAC is not erased during reset, e.g., stored in Flash or EEPROM.

5.3. Attested Reading of Measurements

Some applications need to make sure that values read from a peripheral device cannot be forged by malware possibly present on that device. For example, large-scale incorrect reports of current electricity consumption by smart meters might lead to power outages. Or, an IMD that returns incorrect values when queried by a physician might result in an incorrect prescription issued to a patient, with potentially catastrophic consequences. Predictably, attestation of measurements should provide: (1) freshness of the values read, (2) proof of reading the values from the peripheral and (3) integrity of the values.

Freshness is provided via a nonce, present by default in SMART invocation. Proof of reading the value is provided by calling SMART to attest and run $\mathcal{HC}$, that reads the values. Finally, $\mathcal{HC}$ calls SMART a second time, as a normal HMAC function, to protect integrity of the read values. Algorithm 4 presents this primitive.

Although this approach, using hash chains, bears some resemblance to the extend operation of a TPM, there are some important differences: HMAC attests each output of SMART with the secret key of the device. This allows for a simpler design. Besides integrity, HMAC correctness confirms that it was produced by SMART. This is fundamentally different from the extend operation preformed by a TPM, since integrity of the PCR is enforced by hardware.

Algorithm 3: SMART usage to securely reset a device.

input : n nonce sent by $\mathcal{VRF}$
R reset function address
$|R|$ the reset function size
H HMAC result (global variable)
output: HMAC output
begin
SMART (R, $R+|R|$, R, $True$,n,&H,$\emptyset$) ;
end
// ResetFunction: R()
begin
ShutdownDevices();
EraseAllMemoryButH();
PC = 0 ;
end
// The value H will be returned to $\mathcal{VRF}$
// after boot is completed.

Algorithm 4: SMART usage to attest a measurement, e.g., a reading from a peripheral accessed from memory mapped I/O.

input : n nonce sent by $\mathcal{VRF}$
in address to read from device
R reading function address
$|R|$ the reading function size
$H1$ first HMAC (global variable)
$H2$ second HMAC (global variable)
output: HMAC output
begin
SMART (R, $R+|R|$, R, $True$,n,&$H1$,in) ;
/* Function R will be called by SMART code */
Send(V,H2);
end
// ReadingFunction: R(in)
begin
$V \longleftarrow$ ReadValueFromHW(in) ;
tmp=$V||H1$;
SMART (&tmp,&$tmp+sizeof(tmp)$,0,$False$,n,&$H2$,$\emptyset$) ;
RestoreIRQ();
end

We note that the $Send$ function, that sends the HMAC to $\mathcal{VRF}$, is not guaranteed to be executed since it is not verified by SMART. However, this does not impact validity of the HMAC or the obtained measurements.

5.4. Further Uses and Extensions

A primitive providing a dynamic root of trust, such as SMART, can be used many other purposes. For example, if certain known malware propagates over a network of embedded devices, $\mathcal{VRF}$ can introduce detection or disinfection code. This code could be launched by SMART to perform remote search for known malicious patterns in code or data. Using SMART, validity of returned HMAC would guarantee that detection code was executed uninterrupted and that the detection result is genuine.

SMART can also facilitate mutual authentication and shared key generation between two (or more) previously paired devices. In this case, each device acts as both a $\mathcal{PRV}$ and $\mathcal{VRF}$. SMART guarantees that, even in the event of full software compromise of either device, a device's long-term attestation key cannot be modified or disclosed. Consequently, the adversary cannot clone a genuine device or eavesdrop on communication between two devices. One possible application example is in car key fobs. Such a fob, paired with the car's on-board Embedded Compute Unit (ECU) could share a key protected by SMART.

Fine-grained access control to sensitive peripherals can be limited to $\mathcal{HC}$ only with simple hardware extensions to SMART. For example, $\mathcal{HC}$ code can be provided in a bundle with its own HMAC and a bit field that describes authorization to access specific memory regions corresponding to memory mapped peripherals. Access to these memory regions would, in turn, be authorized only if HMAC is validated. This is useful in many applications, e.g., pacemakers where it could control delivery of pacing impulses.

6. Implementation

To assess feasibility, practicality and impact of SMART we implemented it on two low-end commodity MCU platforms. We believe that this is the best way to understand its benefits and limitations as well as to evaluate the impact of required MCU modifications. We chose to base our implementation on two fully open-source clones of widely used off-the-shelf MCU-s: Atmel AVR and Texas Instruments MSP430. These processors share many features. They both have a limited memory address space with 16-bit addresses. Common memory sizes in both devices are between $2-16$ KBytes of SRAM used as data memory and between $16-64$ KBytes of flash memory used for program storage. Both are designed for low-power as well as low-cost and are widely adopted in many

application areas, e.g., in the automotive industry, utility meters, consumer devices and peripherals.

AVR and MSP430 also have some major architectural differences. Notably, MSP430 is a 16-bit Von Neumann architecture processor with common data and code address spaces. Whereas, AVR is an 8-bit Harvard architecture processor that has separate address spaces for data and program memory. Another prominent difference is in the instruction set: AVR is a RISC architecture with most instructions requiring a single 16-bit word and executing in one clock cycle. In contrast, MSP430 can perform multiple memory accesses within a single instruction. Its instruction execution time can range from 1 to 6 clock cycles, and instruction length can vary from 16 to 48 bits.

The differences between AVR and MSP430 makes them good representatives of architectures commonly used in many modern embedded systems.

6.1. Implementation Details

SMART implementation consists of three main components:

- Processor modifications to add ROM code, key storage and memory access controls.
- Largely architecture-independent SMART routine stored in ROM that implements Algorithm 1. This C code has a small number of architecture-dependent lines.
- One or more software protocol implementations that utilize the SMART primitive.

Implementation on AVR and MSP430 Cores. We first implemented the hardware part of SMART on the AVR processor, an Atmega103 [5] clone from the OpenCores Project [31]. Figure 4a illustrates the execution core and its memory. Parts that had to be modified or added are shaded. They mainly correspond to memory and memory access controls on memory buses.

Next, we implemented SMART on MSP430. We used the open-source OpenMSP430 core from the OpenCores Project [31] and ported SMART to it. The port consists of processor modifications, adaptation of ROM code to MSP430 architecture as well as testing and synthesizing the resulting core. These tasks were performed in one week by one developer with moderate Verilog knowledge and no previous experience with the OpenMSP430 core. Processor modifications were limited to implementing and adding modules for ROM code and key memory. In addition, minor modifications and address checks were required in the memory backbone module of the OpenMSP430 core. The memory backbone module performs arbitration of memory accesses. Figure 4b presents required modifications (shaded) for MSP430.

Component	Original	Changed	
	Lines	Lines	Ratio
AVR, core (VHDL)	3932	151	3.84%
AVR, tests	2244	760	
MSP430, core (Verilog)	4593	182	3.96%
MSP430, tests	17665	1122	

Table 1: Changes made (in # of HDL lines of code) in AVR and MSP430 processors, respectively, excluding comments and blank lines.

Data Size	Cycles	Time at 8MHz
1 KByte	2302281	287 ms
512 Bytes	1281049	160 ms
32 Bytes	387471	48 ms

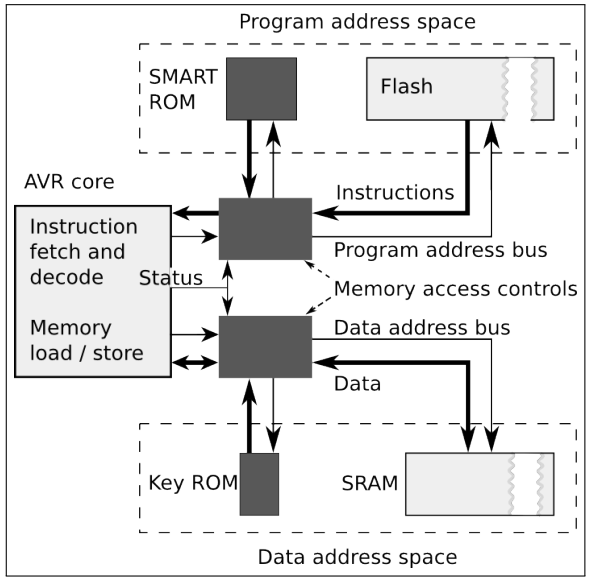
Table 2: HMAC execution timing.

In both processors, less than 200 lines of code (Table 1) were changed to implement these modifications. In addition to processor modifications, we extended existing regression tests (or test benches) to verify correct implementation of each of assertion from Section 4 that is relevant here: A3, A5, A6, and A10.

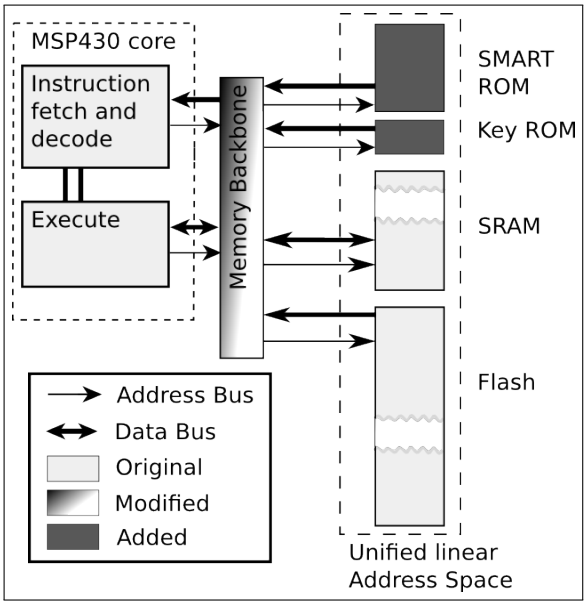
ROM-Resident Code. This code corresponds to 487 lines of portable C and uses a standard SHA-1 implementation [13]. It requires 4KBytes of ROM for the AVR and 6KBytes for MSP430. It executes in 10-s to 100-s of milliseconds (see Table 2), depending on the size of $\mathcal{HC}$ to attest.

Memory usage in SMART has to be carefully managed. SMART code cannot reserve memory for its own usage. Memory should only be allocated on the stack (i.e. local functions variables). It should not attempt to use global variables or heap allocated memory. Doing so allows us to avoid relying on untrusted data. Finally, the code is compiled and linker scripts are used to generate the ROM image suitable to the modified processor.

Hardware Footprint. Simulating the design demonstrates its functional status. Whereas, comparing the number of lines of code of its implementation provides insights into the amount of effort required to implement SMART on a given MCU. However, this is insufficient to assess real impact of SMART in terms of hardware overhead, i.e., surface increase due to its presence on an actual manufactured device. A single line of HDL can add a simple wire, a register or an entire memory block; each of these would be counted



(a) AVR: Dark gray boxes represent logic added to the processor. Core control signals provide information about internal processor status to memory bus controls.



(b) MSP430: Memory backbone was modified to control access to ROM and $\mathcal{K}$. Since MSP430 is based on Von Neumann architecture, concurrent access can occur to different memory parts (e.g., instruction fetch and read data). In that case, memory backbone arbitrates bus access and temporarily saves/restores data.

Figure 4: Modifications to AVR and MSP430.

as one line of code, although they have very different impact on synthesized hardware. We synthesized the original and SMART-ified designs for both AVR and MSP430. This provides an initial estimate of the impact of SMART on the final devices. Synthesizing is the act of transforming (or compiling) the design from a high-level description language (Verilog or VHDL) into a set of wires and elementary gates that serve as building blocks of an Application-Specific Integrated Circuit (ASIC).

Synthesis needs to be performed for a specific target hardware. We used the library from UMC 180nm process [18] and Synopsys Design Compiler [44]. For better performance, RAM and ROM memories were generated with a specific tool [17], [16]. Flash memory numbers were gleaned from publicly available information [10]. Results can vary substantially depending on many parameters, such as: required maximum frequency, latency, placement and routing and availability of better memory IP. However, our current measurements (in Table 3) show that the impact of SMART on surface area is minimal. Adding SMART to both AVR and MSP430 caused only a 10% increase in their respective surface areas. As mentioned before, most of that added area is due to the ROM housing SMART code. Modifications to the core required only 1K and 0.7K gate equivalents in AVR and MSP430, respectively. This could probably be reduced as we did not perform optimizations.

Component		Size in kGE		
		Orig.	with SMART	Ratio
AVR MCU		103	113	10%
Core		11.3	11.6	2.6%
SRAM	4 kB	26,6	26.6	0%
Flash	32 kB	65	65	0%
ROM	6 kB	-	10.3	-
MSP430 MCU		128	141	10%
Core		7.6	8.3	9.2%
SRAM	10 kB	55.4	55.4	0%
Flash	32 kB	65	65	0%
ROM	4 kB	-	12.7	-

Table 3: Comparison of chip surface used by each component of the original MCU to its modified version. kGE stands for thousands of Gate Equivalents (GEs). One GE is proportional to the surface of the chip and computed form the module surface divided by the surface of a NAND2 gate, $9,37 * 10^{-6} mm^2$ with this library.

6.2. Lessons Learned From Experiments

The first observation from our experiments is that implementing SMART is not a complex task and porting it to a different architecture is even easier. Second, additional footprint of our implementation is minimal. One change that impacted chip surface area the most is the additional ROM storing SMART code.

Another important result is that, in both cases, we did not have to change the processor core itself. Instead, we only had to modify the memory access controller.[10] Therefore, SMART might be also well-suited to settings where the processor core is available only as a "black box" and provides enough information about accessed memory on its external interface, e.g., low-end ARM cores.

One limitation is that we rely on "reasonably" fast HMAC computation, which might make SMART too slow for some applications. This is a consequence of the conscious trade-off made when we chose to limit the amount of hardware changes in the processor. Depending on the application, it may be possible to use a hardware-based SHA-1 implementation (e.g., [30]), which would significantly improve performance without requiring major processor modifications.

7. Related Work

Related work falls into several categories:

Hardware Attestation. Secure boot [2] checks the integrity of a system at power-on. The *root of trust*, usually a small bootloader, computes a hash of loaded memory, and compares it to a signed value, a device is allowed to boot-up only if all checks are passed. Trusted Platform Modules (TPM) [45] are secure co-processors that are nowadays present in most commodity systems. TPMs compute integrity checksums of loaded memory at boot time and send them to be verified by a *remote* verifier. TPMs also protect data against compromised operating system, i.e. make an encryption key available only when Platform Configuration Registers (PCRs) are in a given state. The integrity measurements are stored in PCRs inside the TPM. Security is based on the following facts: (1) PCRs are only accessible through the TPM and (2) measurements stored in PCRs can only be *extended* by including the previous values in the computation. Each extension is computed using a cryptographic hash of the current measurement and the previous PCR value. Trust is established because the very first extension is performed by BIOS upon boot. Several approaches have been proposed that rely on the TPM as a common foundation [33], [14], [25].

Software Attestation. Pioneer [37] provides device attestation without relying on any specialized hardware or secure co-processor. It computes a checksum of memory using a function that relies on "enough" side-effects of computation (status registers, etc.) such that malicious emulation of this function incurs a temporal overhead that is sufficient to detect cheating. Attestation that relies on timed software checksums has been also adapted to embedded devices in [36], [38], [40]. However, security of such solutions has been challenged by [41] and several attacks on such schemes have been proposed [8]. Other hybrid solutions (e.g., [32]) rely on ROM and fill prover's entire memory to ensure absence of malicious code and then restore a device to a known secure state. All software solutions rely on strong assumptions on adversarial capabilities and do not consider that colluding devices can actively participate in the protocol to defeat attestation. This, combined with the high overhead of software solutions, makes the application of software attestation for time critical devices questionable.

Dynamic Root of Trust. Recently a dynamic root of trust mechanism has been added to the TPM specifications [46] and has been implemented as AMD SVM [1] and Intel TXT [24]. This provides a way to perform attestation *dynamically* after boot. This is accomplished by allowing a specific CPU instruction to atomically reset the state of some PCRs, isolate a region of memory, hash the contents of that memory and execute it. Several hardware protections measures, such as disabling DMA, debugging and resetting the TPM PCRs, are included to prevent fraudulent attestation. The Flicker system architecture [29] establishes a dynamic root of trust on commodity computers, leveraging AMD and Intel advances, by running a Piece of Application Logic (PAL) on the prover. The execution of PAL is guaranteed even if BIOS, OS and DMA of the system are all compromised. This was further extended into TrustVisor [28] which provides a dynamic root of trust for PALs directly from a minimal hypervisor. This significantly improves the performance of the Dynamic Root of Trust mechanism. Flicker and Trustvisor are the closest to the approach considered in this paper. However, their complexity and reliance on a TPM and Intel or AMD architectures inhibits their use in low-cost commodity embedded

10. The only exception is that, in some cases, we needed information about the execution engine state (e.g., detection of wait states).

devices.

Other Hardware-Based Techniques. SPM [43] is a hardware-based mechanism for process isolation. It relies on a special *vault* module that must be bootstrapped with a static root of trust. This vault bootstraps SPM protected programs that gain exclusive control over the protection of their own memory pages. SPM and SMART share some key features, such as the use of program counter to restrict access to secret storage, and code entry point enforcement. However, unlike SMART, SPM does not provides a dynamic root of trust. It also involves a larger TCB and is generally oriented towards higher-end embedded systems with an MMU or an MPU. Furthermore, SPM requires new custom instructions to be added to the core. Finally, its feasibility (i.e., effort needed to implement on a real hardware platform) and *footprint* remain unclear.

8. Conclusions

This paper is motivated by lack of currently feasible techniques for providing dynamic root of trust on remote embedded devices. We proposed SMART a very simple, lightweight and low-cost architecture that nonetheless offers concrete security guarantees in the presence of any kind of non-physical attacks. Future work will consist in formally verifying the ROM-resident code in order to obtain a strong security proof for the entire architecture; this is likely to be a challenging task. More experiments using current MCU implementations need to be performed to better assess the overhead. We also plan to implement and evaluate SMART on several other common MCU platforms and among a larger project we plan to produce a few test ASIC samples of microcontrollers with SMART.

9. Acknowledgements

We thank Frank K. Gürkaynak for his help with ASIC design as well as Travis Goodspeed, Kasper Rasumussen, Srdjan Ĉapkun and NDSS'12 anonymous reviewers for their insightful comments that helped us improve this paper.

Daniele Perito was supported in part by the European Commission within the STREP WSAN4CIP project. Aurélien Francillon was supported by the European Commission within the STREP TAMPRES project. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsement of the WSAN4CIP or TAMPRES projects or the European Commission.

References

[1] ADVANCED MICRO DEVICES. AMD, Secure Virtual Machine Architecture Reference Manual. Publication No. 33047, Revision 3.01, May 2005.

[2] ARBAUGH, W. A., FARBER, D. J., AND SMITH, J. M. A secure and reliable bootstrap architecture. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1997), IEEE Computer Society, p. 65.

[3] ARM CORPORATION. Building a secure system using TrustZone technology. Publication number: PRD29-GENC-009492C.

[4] ARM CORPORATION. Securcore processors, 2011. http://www.arm.com/products/processors/securcore/index.php.

[5] ATMEL CORPORATION. 8-bit microcontroller with 128k bytes in-system programmable flash.

[6] ATMEL CORPORATION. *ATMEL Trusted Platform Module AT97SC3201*, June 2005. http://www.atmel.com/atmel/acrobat/doc5010.pdf.

[7] BUCHANAN, E., ROEMER, R., SHACHAM, H., AND SAVAGE, S. When good instructions go bad: generalizing return-oriented programming to RISC. In *CCS '08: Proceedings of the 15th ACM conference on Computer and Communications Security* (2008), ACM.

[8] CASTELLUCCIA, C., FRANCILLON, A., PERITO, D., AND SORIENTE, C. On the difficulty of software-based attestation of embedded devices. In *CCS 09: Proceedings of 16th ACM Conference on Computer and Communications Security* (November 2009).

[9] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A.-R., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *Proceedings of CCS 2010* (Oct. 2010), A. Keromytis and V. Shmatikov, Eds., ACM Press, pp. 559–72.

[10] CHINGIS TECHNOLOGY CORPORATION. Embedded e^2 Flash IP, PF32K17E, 32Kbyte (16K x16) Embedded Flash Macro (EFM), 2011. Details available online at: http://www.chingistek.com/pfusion_03.asp?seq=9.

[11] CONDIT, J., HARREN, M., ANDERSON, Z., GAY, D., AND NECULA, G. Dependent types for low-level programming. In *Programming Languages and Systems*, R. De Nicola, Ed., vol. 4421 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007, pp. 520–535.

[12] COSTAN, V., SARMENTA, L. F., DIJK, M., AND DEVADAS, S. The trusted execution module: Commodity general-purpose trusted computing. In *Proceedings of the 8th IFIP WG 8.8/11.2 international conference on Smart Card Research and Advanced Applications* (Berlin, Heidelberg, 2008), CARDIS '08, Springer-Verlag, pp. 133–148.

[13] EASTLAKE, D., AND JONES, P. RFC 3174 - US Secure Hash Algorithm 1 (SHA1). IETF RFC, September 2001.

[14] ENGLAND, P., LAMPSON, B., MANFERDELLI, J., PEINADO, M., AND WILLMAN, B. A trusted open platform. *IEEE Computer 36*, 7 (2003).

[15] FALLIERE, N., MURCHU, L. O., AND CHIEN, E. W32.stuxnet dossier, version 1.4. Symantec Security Response, February 2011.

[16] FARADAY TECHNOLOGY CORPORATION. 0.18μm synchronous VIA1 programmable ROM compiler, FSA0A_C_SP, 2004. Details available online at: http://www.faraday-tech.com.

[17] FARADAY TECHNOLOGY CORPORATION. 0.18um synchronous high speed single port memory compiler fsa0a_c_su, 2004. Details available online at: http://www.faraday-tech.com.

[18] FARADAY TECHNOLOGY CORPORATION. Faraday FSA0A C 0.18 μm ASIC standard cell library, 2004. Details available online at: http://www.faraday-tech.com.

[19] FOSTER, J. S., TERAUCHI, T., AND AIKEN, A. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (New York, NY, USA, 2002), PLDI '02, ACM, pp. 1–12.

[20] FRANCILLON, A., AND CASTELLUCCIA, C. Code injection attacks on Harvard-architecture devices. In *CCS 08: Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), P. Ning, P. F. Syverson, and S. Jha, Eds., ACM.

[21] GOODSPEED, T. Extracting keys from second generation zigbee chips. Black Hat USA, July 2009.

[22] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: Cold boot attacks on encryption keys. In *USENIX Security Symposium* (2008), USENIX Association, pp. 45–60.

[23] HU, W., TAN, H., CORKE, P., SHIH, W. C., AND JHA, S. Toward trusted wireless sensor networks. *ACM Trans. Sen. Netw. 7* (August 2010), 5:1–5:25.

[24] INTEL CORPORATION. *Intel Trusted Execution Technology (Intel TXT) – Software Development Guide*, December 2009. Document Number: 315168-006.

[25] KIL, C., SEZER, E. C., AZAB, A. M., NING, P., AND ZHANG, X. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *DSN 09: Proceedings of the 39th IEEE/IFIP Conference on Dependable Systems and Networks* (June 2009).

[26] KOSTIAINEN, K., DMITRIENKO, A., EKBERG, J.-E., SADEGHI, A.-R., AND ASOKAN, N. Key attestation from trusted execution environments. In *Proceedings of the 3rd international conference on Trust and trustworthy computing* (Berlin, Heidelberg, 2010), TRUST'10, Springer-Verlag, pp. 30–46.

[27] KRAHMER, S. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. Tech. rep., suse, September 2005. available at http://www.suse.de/ krahmer/no-nx.pdf.

[28] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V. D., AND PERRIG, A. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland 2010)* (May 2010).

[29] MCCUNE, J. M., PARNO, B. J., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: an execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 315–328.

[30] O'NEILL (MCLOONE), M. Low-Cost SHA-1 Hash Function Architecture for RFID Tags. In *Workshop on RFID Security – RFIDSec'08* (Budapest, Hungary, July 2008).

[31] The Opencores Project. http://opencores.org/.

[32] PERITO, D., AND TSUDIK, G. Secure code update for embedded devices via proofs of secure erasure. In *Proceedings of the 15th European conference on Research in computer security* (Berlin, Heidelberg, 2010), ESORICS'10, Springer-Verlag, pp. 643–662.

[33] SAILER, R., ZHANG, X., JAEGER, T., AND VAN DOORN, L. Design and implementation of a tcg-based integrity measurement architecture. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium* (Berkeley, CA, USA, 2004), USENIX Association, pp. 16–16.

[34] SCHRAMM, K., LEMKE, K., AND PAAR, C. Embedded cryptography: Side channel attacks. In *Embedded Security in Cars*, K. Lemke, C. Paar, and M. Wolf, Eds. Springer Berlin Heidelberg, 2006, pp. 187–206.

[35] SESHADRI, A., LUK, M., AND PERRIG, A. SAKE: Software attestation for key establishment in sensor networks. In *DCOSS '08: Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems* (2008).

[36] SESHADRI, A., LUK, M., PERRIG, A., VAN DOORN, L., AND KHOSLA, P. SCUBA: Secure code update by attestation in sensor networks. In *WiSe '06: Proceedings of the 5th ACM workshop on Wireless security* (2006), ACM.

[37] SESHADRI, A., LUK, M., SHI, E., PERRIG, A., VAN DOORN, L., AND KHOSLA, P. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (2005), ACM.

[38] SESHADRI, A., PERRIG, A., VAN DOORN, L., AND KHOSLA, P. K. SWATT: SoftWare-based ATTestation for embedded devices. In *IEEE Symposium on Security and Privacy* (2004), IEEE Computer Society.

[39] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and Communications Security* (2007), ACM.

[40] SHANECK, M., MAHADEVAN, K., KHER, V., AND KIM, Y. Remote software-based attestation for wireless sensors. In *ESAS* (2005).

[41] SHANKAR, U., CHEW, M., AND TYGAR, J. D. Side effects are not sufficient to authenticate software. In *Proceedings of the 13th USENIX Security Symposium* (August 2004).

[42] SOLAR DESIGNER. *return-to-libc attack*. Bugtraq mailing list, August 1997.

[43] STRACKX, R., PIESSENS, F., AND PRENEEL, B. Efficient isolation of trusted subsystems in embedded systems. In *Proceedings of ICST Conference on Security and Privacy in Communication Networks (SecureComm'10)* (2010).

[44] SYNOPSYS, INC. Design compiler 2010, 2010. http://www.synopsys.com/home.aspx.

[45] TRUSTED COMPUTING GROUP. TCPA main specification, version 1.1b.

[46] TRUSTED COMPUTING GROUP. TPM main specification level 2 version 1.2.

[47] YANG, Y., WANG, X., ZHU, S., AND CAO, G. Distributed software-based attestation for node compromise detection in sensor networks. In *SRDS* (2007), IEEE Computer Society.