

# Practical Runtime Attestation for Tiny IoT Devices

Stefan Hristozov\*, Johann Heyszl\*, Steffen Wagner\* Georg Sigl†

\*Fraunhofer Institute for Applied and Integrated Security AISEC

Email: {stefan.hristozov, johann.heyszl, steffen.wagner}@aisec.fraunhofer.de

†Technical University of Munich

Email: sigl@tum.de

**Abstract**—One of the main challenges in IoT security is to assure the integrity of the firmware running on a constrained low-cost device. A solution to this challenge could be provided by security service called attestation, where the device generates an evidence about its firmware which is attested by a remote verifier. How attestation evidence can be generated at boot time on a tiny microcontroller was investigated in earlier work and also specified by the TCG's DICE specification. It is, however, challenging to generate such attestation evidence during runtime, where the device usually is prone to powerful attacks. Previous contributions have attempted to solve this by using custom hardware extensions of the CPU architecture. We, however, present a method based on DICE to securely generate attestation evidence at runtime using only standard CPU features like MPU, privileged/unprivileged levels of execution and the required by DICE boot ROM and lock mechanism. Precisely, we use the MPU and privilege levels to effectively isolate the attestation firmware and secrets from the remaining application. As a result, our method can immediately be applied to a broad range of popular microcontrollers. We provide a proof of concept implementation for the Cortex-M4-based STM32L476 microcontroller.

## I. INTRODUCTION

A key challenge in IoT security is the vulnerability of constrained microcontrollers against malicious modification of their firmware. This can be a result of (1) reprogramming attacks performed by an adversary with physical access to the device or (2) remote attacks which use vulnerabilities in the software implementation. A popular approach to mitigate such attacks is a security service called *attestation* [5, 14]. Attestation doesn't provide protection against malicious firmware modification but a way to verify that a device runs known firmware, i.e., it is in a trusted state. By definition attestation is a process between two parties: a prover (Prv) and a verifier (Vrf). In the context of this work Prv is the resource constrained IoT device and Vrf is some general more powerful device, e.g., server back-end. From the prover's perspective attestation consists of two steps: (1) generation of an evidence about its trustworthiness and (2) a secure protocol for conveying this evidence to the verifier. In this paper we concentrate on the first part of the attestation process – the generation of the attestation evidence, particularly during runtime.

Previous research has shown that powerful security architectures for low-end microcontrollers may be realized by using on-chip hardware/software features. The most relevant are: SMART [12], TrustLite [17], TyTAN [9], SPM [24] and Sancus [19]. On

the one hand they provide several security services one of which is attestation but on the other they require non-standard on-chip hardware like: custom bus access logic, additional CPU instructions or special execution-aware MPU. It was also shown [21, 13] that more simple features like a secret that can be accessed after reset from a boot ROM and locked by hardware afterwards may be sufficient to generate an attestation evidence at boot time. This is also the approach that Device Identifier Composition Engine (DICE) [4] from the Trusted Computing Group (TCG) follows. An example for such lock mechanism is the STM32 firewall [2]. The firewall can protect memory areas to be read out from malicious software. Once a memory area is locked by the firewall it can only be unlocked by reset.

DICE and all other methods, where the attestation evidence is generated at boot time are vulnerable against an adversary modifying the firmware at runtime after the evidence generation. We approach this problem by proposing a DICE based hardware/software method for fresh evidence generation at runtime, requiring only standard on-chip hardware. Additionally to the required by DICE boot ROM and lock mechanism our method requires only a standard Memory Protection Unit (MPU) and privileged/unprivileged levels of execution and doesn't require e.g. external components like TPM [3], trusted execution environments like TrustZone [1], or any on-chip non-standard hardware features like shown in [12, 17, 9, 24, 19]. Its main idea is to use a small, privileged, write-protected software layer which handles access to secrets at runtime using the MPU and can generate an attestation evidence at request from Vrf. This method is especially targeted at inexpensive off-the-shelf microcontrollers typically running at up to 200 MHz, having up to few hundred KBytes of on-chip RAM and up to 1 MByte of flash. We demonstrate that our approach is feasible and practical for this class of devices by an implementation on the STM32L476 which is an ARM Cortex-M4 microcontroller.

## II. RELATED WORK

In this section we give an overview over the most common techniques for attestation evidence generation and the TCG's DICE.

### A. Generating an Attestation Evidence

The attestation evidence is a block of data which provides sufficient information to Vrf in order to decide whether Prv runs known and trusted software, i.e., Prv is in a trusted state. In the most general case the attestation evidence is a measurement generated by a cryptographic hash function over the firmware of the device, or parts of it. In the following, we give an overview over the most common methods of generating attestation evidence, see also [5, 10]:

1) *Standard-Hardware-Based*: One of the most popular attestation methods uses a discrete co-processor called Trusted Platform Module (TPM) [3]. TPMs are equipped with special purpose registers called Platform Configuration Registers (PCRs). PCRs cannot be overwritten but only extended by hashing of software measurements together with the PCRs previous values. The TPM can sign the PCRs with an attestation private key in order to generate an attestation evidence.

Other standard-hardware-based methods use tightly integrated trusted execution environments within the main application processor. ARM TrustZone [1] is an example for such execution environment. TrustZone provides two virtual processors known as secure world and normal world. The isolation between them is enforced by hardware. An example how TrustZone may be used for attestation is given by Microsoft's fTPM [20] where TrustZone is used for a firmware TPM emulation.

Another trusted execution environment is Intel SGX [18]. SGX provides instruction and memory access features which can be used to instantiate protected containers referred to as enclaves. SGX has a built-in mechanism which uses special instructions and processor extensions for attestation.

In this paper we provide an attestation method for low-cost IoT devices, where a discrete TPM typically cannot be used because of economic and power consumption reasons. Even though ARM recently announced the ARMv8-M low-end microcontroller architecture with support for TrustZone we are not considering using such trusted execution environments because the majority of microcontrollers available now and in the future will still lack such hardware features.

2) *Software-Based*: Software-based methods use functions with side channel information such as timing required for certain operations. Any emulation of this functions leads to significant overhead, which may be used to detect fraud, see [23, 22]. Such methods are unsuitable for IoT applications because they require a communication environment with precise time behavior.

3) *Hardware/Software-Based*: As shown above, the standard-hardware-based and the software-based techniques are not suitable for low-end IoT applications. In order to overcome their limitations several hardware/software techniques were developed; the most relevant of which are:

**SMART**: SMART [12] is a dynamic root of trust architecture for low-end devices, which is capable to generate an attestation evidence at runtime. SMART uses custom hardware bus access logic and hence, is not applicable for the currently available of-the-shelf microcontrollers.

**SPM/Sancus**: SPM [24] and Sancus [19] present a security architecture which provides isolation of software modules using additional CPU instructions. This architecture allows very fine grained task isolation, but has the obvious disadvantage, that it requires modification of the processor's instruction set in hardware.

**TrustLite/TyTAN**: TrustLite [17] and its successor TyTAN [9] provide flexible, hardware-enforced isolation of software modules. This is achieved by using a dedicated Execution-Aware Memory Protection Unit (EA-MPU) in hardware.

**RIoT/DICE**: The described hardware/software techniques are powerful architectures. Compared to them, RIoT [13] and DICE [4] are simpler. They propose a method for generating attestation evidence at boot time by requiring a secret value which is accessed from a boot ROM and then locked. The feasibility of DICE for off-the-shelf devices is already demonstrated in [16]. Another similar boot time attestation technique is presented by Schulz et al. in [21]. All boot time methods are vulnerable against adversaries, which can modify at run time the device's firmware after the evidence generation.

### B. Boot Time Attestation with DICE

A device that implements the DICE [4] specification may consist of  $n$  software layers  $\mathcal{L} = \{L_0, L_1, \dots, L_{n-1}\}$  and an Unique per Device Secret ( $UDS$ ) which doesn't change during the lifetime of the device. The lowest layer  $L_0$  is placed in ROM. After reset  $L_0$  reads the  $UDS$  and locks it using a hardware feature e.g. a firewall, therefore  $UDS$  cannot be accessed from firmware in the other layers. The layers are executed sequentially starting from  $L_0$ . Each layer  $L_m$  measures the software of the next layer  $L_{m+1}$  using a cryptographic hash function  $h()$  in order to generate a digest  $D_{m+1}$  of its binary.

$$D_{m+1} = h(L_{m+1}) \quad (1)$$

A deterministic one-way key derivation function  $KDF()$  is used in order to generate a derived secret  $K_m$ . The derived secret  $K_m$  is based on a measurement of the next layer  $D_{m+1}$  and  $UDS$  or  $K_{m-1}$ , see Eq. 2 and Fig. 1.

$$K_m = \begin{cases} KDF(UDS, D_1) & \text{for } K_m = K_0 \\ KDF(K_{m-1}, D_{m+1}) & \text{for } K_m > K_0 \end{cases} \quad (2)$$

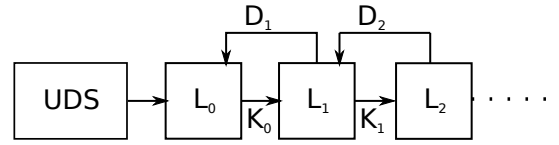


Figure 1: Basic DICE structure

After all operations within a given layer  $L_m$  are executed, the derived secret of the previous layer  $K_{m-1}$  and all its artifacts in registers and RAM are deleted. Thus, it is obvious that any modification of the firmware on any layer  $L_m$  except  $L_0$ , which is placed in ROM will lead to different measurement  $D_m$  and different derived secret  $K_{m-1}$ . This means that a derived secret is suitable to be used as an attestation evidence  $E$ .

### III. PROBLEM STATEMENT

The firmware of a device that implements DICE or any similar boot time attestation technique may be compromised at runtime after the evidence  $E$  is generated. In this case the adversary may access  $E$  and use it to attest the device. Moreover, the adversary may save  $E$  and a malicious firmware in the non-volatile memory, which will cause a different evidence  $E'$  to be calculated at the next reboot. However, this would be irrelevant because the adversary will use the stored  $E$  for any further attestations. DICE provides no mechanism to detect

such a malicious modification of the device firmware, i.e., the attestation evidence cannot be trusted.

A possible approach to handle such situations is to use  $E$  once and then delete it. This approach requires the strong assumption that the attestation happens just after power up, thus the exposure of  $E$  is constrained. A bigger disadvantage is that for a second attestation, a reboot is required and this is unacceptable for the majority of real world applications.

Another approach which doesn't require a reboot and is currently state of the art is to update the firmware of all devices with a new patched version when a vulnerability is discovered. Then all devices must attest with a new  $E^{new}$  which corresponds to the patched version. This approach has two disadvantages: (1) a successful attack on an embedded device does not necessarily manifest itself in a way that can be recognized, and, (2) sufficient support for security patches may be difficult because IoT devices are expected to be very long-lived and inexpensive.

#### IV. ADVERSARIAL MODEL

Our proposed attestation method provides a mechanism to detect malicious firmware modification caused by one of the following adversary types:

1) *Physical Adversary*: A physical adversary has physical access to the device and can perform simple reprogramming attacks overwriting firmware layers excluding the boot layer  $L_0$  and  $UDS$ , which are placed in ROM. Our method provides no protection against a physical adversary who can debug the device, readout its firmware or perform more powerful and complex physical attacks.

2) *Remote Adversary*: A remote adversary can remotely modify the firmware of Prv residing in flash memory. For this purpose, the remote adversary may use a memory corruption vulnerability of the Prv at runtime to gain access over the program execution and trigger permanent modification of the flash memory.

#### V. RUNTIME ATTESTATION

In this section we describe our runtime attestation method.

##### A. Overview

The overview of our method is structured as following: first the main components that we use are described, then the concept of privileged levels is introduced. We conclude the overview with a brief description of our runtime attestation method.

1) *Main Components*: Our runtime attestation method is based on software architecture consisting of three software layers: boot layer  $L_0$ , core layer  $L_1$  and OS/App layer  $L_2$ . The static software components (code and constants) of  $L_0$  are placed in ROM. The static software components of  $L_1$  and  $L_2$  are placed in flash. The layers share the same RAM area for global variables, stack and heap. As by DICE we use an Unique Device Secret  $UDS$  which has to be immutable, thus it is placed in ROM.

The boot layer is responsible for calculating the first derived secret by measuring  $L_1$  and calculating  $K_0 =$

$KDF(UDS, D_1)$ . The core layer is responsible for the generation of the attestation evidence. The OS/App layer contains the application on top of optional real time OS.

2) *Privilege Levels*: Our attestation method uses privileged and unprivileged software execution. An executed as privileged software can change the configuration of certain resources, but the same software cannot change their configuration when executed as unprivileged. Relevant privileged resources in this paper are: the MPU, the interrupt controller and the vector table relocation register.


The switch from unprivileged to privileged happens by means of an interrupt, i.e., when an interrupt occurs the corresponding Interrupt Service Routine (ISR) is executed as privileged software. A regular function call to an ISR doesn't change the privilege level of execution, i.e., when the call comes from unprivileged software an ISR is executed as unprivileged, when it comes from privileged software the ISR is executed as privileged. The switch from privileged to unprivileged happens when execution returns from an ISR or by writing to a special hardware register, which is only writable from privileged software. We refer to this register as `control` register. We also assume that after power up or reset, code is executed as privileged.

3) *Main Concept of Runtime Attestation*: Our method is based on the idea that secrets and privileged software are protected by using either ROM, MPU or hardware memory lock (firewall). Such privileged software is the software in  $L_0$ ,  $L_1$  and the ISRs. The software of  $L_0$  is placed in ROM, thus it is write-protected by default. The software of  $L_1$  and the ISRs is write-protected with the MPU. The layer  $L_2$  is not write-protected, thus it can be overwritten by software update.

At runtime the application code in  $L_2$  is executed as unprivileged, i.e., it cannot modify the MPU configuration. At some point in time  $L_2$  gets an attestation request from Vrf which contains a nonce  $N$ . The application code generates a software interrupt. The corresponding ISR contains the core layer  $L_1$  which is executed as privileged and can modify the MPU in order to unlock a secret previously generated and held in RAM. The core layer then measures the OS/App layer  $L_2$  and binds cryptographically the measurement, the secret and the nonce in order to generate an attestation evidence. After this, the secret is locked with the MPU and the execution returns into  $L_2$ .

##### B. Runtime Attestation Process

In the following we describe our runtime attestation process step-by-step and refer to Fig. 2:

1) *Step 1*: After power-up or reset the program execution starts in the boot layer  $L_0$  as privileged, see  symbol in Fig. 2.  $L_0$  measures the core layer  $L_1$  and calculates  $K_0 = KDF(UDS, D_1)$ . Then  $K_0$  is written in dedicated area in RAM. Then the MPU is configured to restrict read and write access this area, and write access to  $L_1$ , ISRs and the vector table. The  $UDS$  is locked with a hardware feature e.g. a firewall and it can only be unlocked by reset. Then the execution is passed to the ISR containing core layer by a function call.

2) *Step 2*: In this step, the core layer may enforce a Writable XOR Executable ( $W\oplus X$ ) policy [25], which prevents a remote adversary to load and execute code from RAM. For this purpose, the MPU is used, whose configuration is modifiable since the core layer is executed as privileged. The  $W\oplus X$  policy states that a memory area can either be writable or executable, but not both. The core layer may load code from flash into a certain RAM area and make this area non-writable but executable using the MPU. The rest of the RAM excluding the area which contains  $K_0$  should be made writable but non-executable, in order to hold stack, heap and global variables. Alternatively, if it is not desired to execute code from RAM, the complete RAM may be made non-executable. Then the privilege level is switched to unprivileged by writing to the control register and execution is passed to the OS/App layer by a function call.

3) *Step 3*: In this step code from the application layer  $L_2$  is executed. The application builds communication channel to Vrf. Then at some point in time Vrf sends an attestation request which contains a nonce  $N$ . The nonce is made available for core layer  $L_1$  by using a global variable. Then the application generates a software interrupt which passes the execution to the ISR containing  $L_1$ .

4) *Step 4*: This step is divided in two sub steps for clarity: First, the core layer  $L_1$  measures the OS/App layer  $L_2$  and ISRs excluding the core layer itself. For the measurement, the function  $h(\cdot)$ , see Eq. 1 is used. Then  $K_0$  is unlocked and read out. An asymmetric key pair  $\{K^{pub}, K^{priv}\}$  is deterministically generated based on  $K_0$ :

$$\{K^{pub}, K^{priv}\} = KDF_a(K_0). \quad (3)$$

The private key  $K^{priv}$  is used to sign the nonce  $N$  and the measurements  $D_{isr}$ ,  $D_2$ , in order to calculate the attestation evidence  $E$ :

$$E = \{N, D_{isr}, D_2\}_{K^{priv}}. \quad (4)$$

Additionally, if application code is contained in RAM, it has to be measured and its measurement has to be included in the evidence  $E$ :

$$E = \{N, D_{isr}, D_2, D_{ram\_code}\}_{K^{priv}}. \quad (5)$$

In the second sub step  $K^{priv}$  is saved and access to it is restricted using the MPU.  $E$  is made available to  $L_2$  by using a global variable. Key artifacts are erased from stack and registers. Then the execution returns from the ISR to the Application in  $L_2$ , which is executed as unprivileged.

5) *Step 5*: In this step the application in  $L_2$  sends  $E$  to Vrf.

6) *Subsequent attestations*: In case that Vrf sends another request, the steps are executed again starting from step 3 but instead of generating  $\{K^{pub}, K^{priv}\}$  again,  $K^{priv}$  is used.

### C. Asymmetric cryptography

We use asymmetric cryptography for the evidence generation. An alternative might be to use an HMAC. The advantage of asymmetric cryptography for our use case is that Vrf and Prv doesn't need to have a shared secret. However,  $K^{pub}$  must be extracted from Prv in a trusted environment, e.g., during production and provided to Vrf through separate trusted channel. Alternatively,  $K^{pub}$  can be generated by the production server. The verifier Vrf must also know the measurements  $D_{isr}$ ,  $D_2$  and  $D_{ram\_code}$  additionally to  $K^{pub}$ .

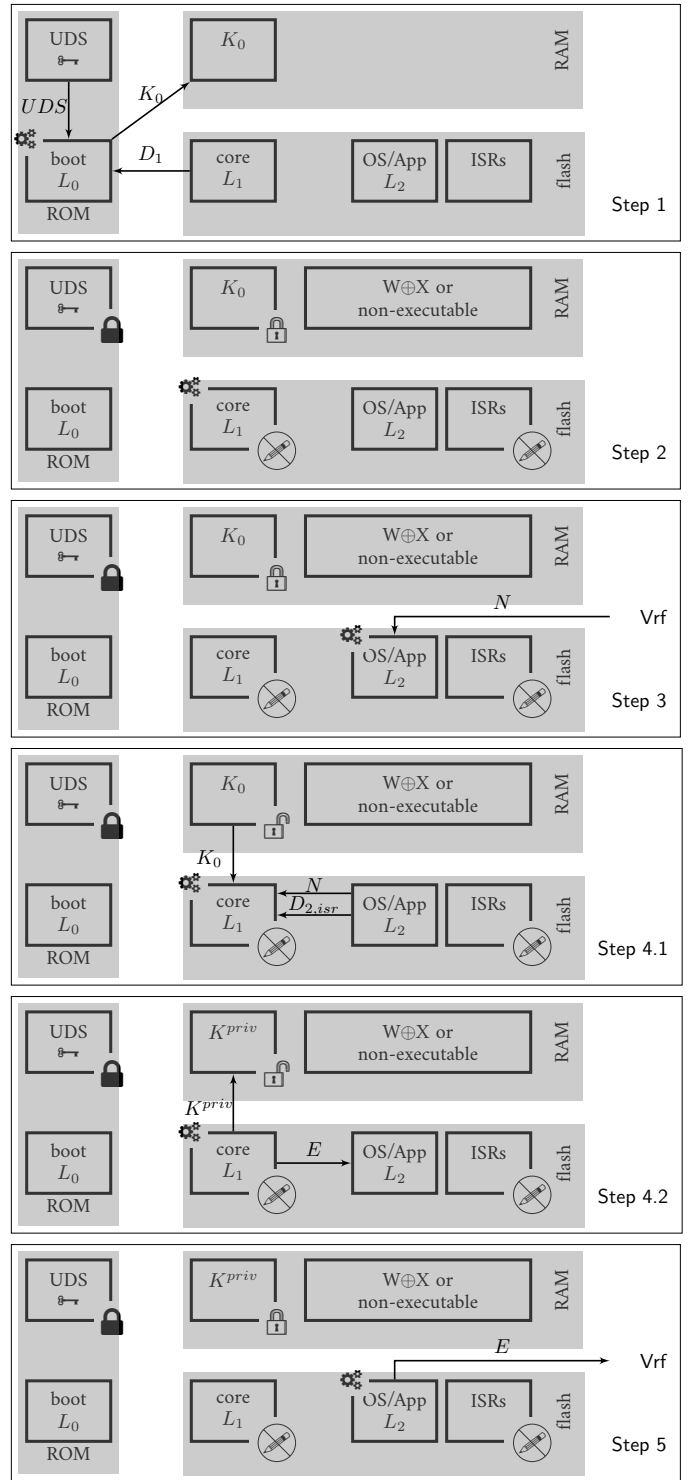


Figure 2: Runtime Attestation Process

Legend: code from this code area is executed, UDS locked with memory lock (firewall), secret in RAM locked with MPU, secret in RAM unlocked, wire-protected privileged software

#### D. Interruptibility

The evidence generation is a time consuming task. Therefore, it is realistic to assume that a real world application will require to interrupt  $L_1$  in order to execute some hard real-time operations. We approach this problem by configuring the ISR, which contains  $L_1$ , to have the lowest interrupt priority and all other interrupts to have higher priorities. In this way more urgent interrupts can be handled. It is secure to do so, because all ISRs are write-protected and their behavior is defined, thus they will not, e.g., read out secrets during the execution of  $L_1$ . Also if a remote adversary is presented in OS/App layer, it will not be possible for the remote adversary to program new ISRs because the interrupt controller can only be modified from software executed as privileged. A remote adversary is also not capable to re-target interrupts to malicious code, since the vector table is non-modifiable and it cannot be relocated from unprivileged software.

#### E. Use Of Privileged Resources

An OS or application in  $L_2$  may require to configure other privileged resources which are not related to our attestation method. For this purpose, the same concept of software generated interrupt may be used. The ISR corresponding to the software interrupt, may contain a switch-case statement with predefined configurations of privileged resources. Which branch will be executed might be controlled by passing a value by a global variable.

#### F. Loading Software Components in RAM at Runtime

For some applications may be required to load different software components at runtime in RAM and execute them. This task may be realized by the core layer. Then, at an attestation request, additionally to the attestation evidence  $E = \{N, D_{isr}, D_2, D_{ram\_code}\}_{K^{priv}}$  also an identifier of the currently loaded component has to be sent to Vrf. In this way Vrf can determine which software component is currently contained in RAM and verify the evidence.

### VI. REQUIREMENTS AND LIMITATIONS

#### A. Requirements

1) *MPU and Privileged/Unprivileged Execution:* The proposed method requires an MPU and privileged/unprivileged software execution. The switch between privileged and unprivileged execution has to happen by means of an interrupt or a write to a register as described in Section V-A2.

2) *Boot ROM:* Our attestation method also requires a ROM memory for holding the boot layer  $L_0$  and  $UDS$ . Alternatively, often the commercially available chips can provide a mechanism to write-protect flash areas using option bytes, which can be used instead of boot ROM.

3) *UDS and UDS Lock Mechanism:* Our attestation method also requires Unique per Device Secret ( $UDS$ ) placed in ROM and  $UDS$  lock mechanism as required by the DICE specification.

#### B. Limitations

1) *Strong Physical Adversary:* The proposed attestation method provides no protection against a strong physical adversary, which can overcome the device's debug protection or perform more powerful physical attacks.

2) *ROP Vulnerability:* The mentioned in Section II methods, as well as our method, do not provide detection of Return-Oriented Programming (ROP) attacks. Related work in this area is DynIMA [11], which combines boot time TPM-based binary measurement and code instrumentation in order to attest runtime program flow. DynIMA doesn't provide any detection of the binary compromise after measurement.

3) *DOS Vulnerability:* The proposed attestation method may be vulnerable against DOS attacks started by a malicious verifier. This is because measurement and signing are slow and energy consuming operations. An additional mechanism is required to ensure that Vrf is not malicious. We note that this is a separate branch of research and refer to [10, 15] for more information.

### VII. IMPLEMENTATION AND EVALUATION

The proposed attestation method can be implemented on devices which meet the requirements from Section VI-A. One of this requirements is MPU and privileged/unprivileged execution, which is met by, e.g., microcontrollers based on ARM Cortex-M0+/M3/M4/M7. Switch between privileged and unprivileged execution is achieved by means of an interrupt and a register as described in Section V-A2. A software interrupt can be generated by executing the `svc` instruction.

Another requirement is ROM memory. Many of the commercially available microcontrollers may provide ROM memory. On chips which don't provide such memory, flash memory protection by means of option bytes is often available.

DICE as well as our method require also a dedicated mechanism to make  $UDS$  not accessible after it is read out by the boot ROM.  $UDS$  must become accessible again only after reset. This requirement is satisfied by the STM firewall [2]. A list of similar hardware features from other vendors is given in [21].

#### A. Cryptographic Operations

The proposed attestation framework requires the use of several cryptographic operations which can be divided in two groups: (1) hash-based cryptography for the measurement function, see Eq. 1, key derivation function, see Eq. 2, and (2) asymmetric cryptography for the derivation of the asymmetric keys, see Eq. 3 and signing for the evidence generation, see Eq. 4. As measurement function we used SHA256. As key derivation function we used SHA-256-HMAC. For the asymmetric operations we chose the Ed25519 signature algorithm proposed by Bernstein et al. [7]. Ed25519 has two advantages compared to other asymmetric signature algorithms – (1) it is significantly faster compared to RSA or ECDSA and (2) the private key can be any 32 Byte long number. The second allows us to omit the need of dedicated asymmetric key derivation function (Eq. 3) and to use  $K_0$  as a private key in Eq 4. For the crypto operations we used the library cifra [8] and Ed25519 implementation by Beer et al. [6].

## B. Evaluation

For the evaluation we used an STM32L476, which is based on Cortex-M4, has 1 MByte flash, 128 KByte SRAM and runs at 80 MHz. This chip provides the possibility to use option bytes in order to write-protect parts of the flash memory which is required for layer  $L_0$  and  $UDS$ . For the  $UDS$  lock we use the firewall [2] provided by this chip. For all presented results the highest compiler optimization  $-O3$  was used. Table I presents the static memory requirements for code and constants of  $L_0$  and  $L_1$  as well as secrets in ROM and RAM. Table II shows

Table I: Static memory requirements

Memory area	Memory type	Size
$L_0$ boot	ROM	3768 Byte
$L_1$ core	flash	7672 Byte
Secrets in ROM	ROM	32 Byte
Secrets in RAM	RAM	32 Byte

the dynamic memory requirements in terms of peek stack usage and required time for the operations. As shown in Table I, the

Table II: Peek stack usage and speed

$L_x$	Operation on bytes	Stack peek	Cycles	Seconds
$L_0$	HMAC / 8 K	1392 Byte	647.42e3	8.1 ms
$L_1$	sha256 / 1011 K	896 Byte	117.38e6	1.46 s
$L_1$	ed25519 sign	1280 Byte	51.73e6	646.62 ms
$L_1$	total	1280 Byte	169.11e6	2.12 s

static memory requirements are less than 12 KByte, which is acceptable for a broad range of low-end microcontrollers. The dynamic RAM requirements for the stack are also acceptable. As shown in Table II, the most time-consuming operations are the SHA256 measurement of the memory area available for the application code, which for this micro-controller is almost 1 MByte (1011 KByte) and Ed25519 signing.

## VIII. CONCLUSION

In this paper we presented an attestation method for low-end IoT devices. Our method provides detection of runtime compromise of the device firmware. This is a feature that other attestation techniques, where the attestation evidence is generated at boot time do not provide. Moreover, we achieve this by using only standard on-chip hardware. More precisely, we use MPU and privilege levels to effectively isolate the attestation firmware and secrets from the remaining application. As a result, our method can immediately be applied to a broad range of popular microcontrollers, which we demonstrated with an implementation on the Cortex-M4-based STM32L476. We conclude our paper by presenting an evaluation of the implementation in terms of memory and computation speed.

## ACKNOWLEDGEMENTS

The work presented in this contribution was supported by the German Federal Ministry of Education and Research in the project secUnity through grant number 16KIS0394.

## REFERENCES

- [1] "Building a secure system using trustzone technology." [Online]. Available: <http://www.arm.com>
- [2] "Stm32l4 firewall." [Online]. Available: [http://www.st.com/content/ccc/resource/STM32L4\\_Security\\_Firewall.pdf](http://www.st.com/content/ccc/resource/STM32L4_Security_Firewall.pdf)
- [3] "Tpm main specification level 2 version 1.2, revision 116," March 2011. [Online]. Available: <https://trustedcomputinggroup.org/>
- [4] "Trusted platform architecture hardware requirements for a device identifier composition engine," December 2016. [Online]. Available: <https://trustedcomputinggroup.org/>
- [5] T. Abera *et al.*, "Things, trouble, trust: On building trust in iot systems," in *2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*.
- [6] D. Beer, "Curve25519 and Ed25519 for low-memory systems," <http://www.dlbeer.co.nz/oss/c25519.html>, 2017, [Online; accessed on 30.10.2017].
- [7] D. J. Bernstein *et al.*, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, 2012.
- [8] J. Birr-Pixton, "Cifra," <https://github.com/ctz/cifra>, 2017, [Online; accessed on 30.10.2017].
- [9] F. Brasser *et al.*, "Tytan: Tiny trust anchor for tiny devices," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference*.
- [10] —, "Remote attestation for low-end embedded devices: The prover's perspective," in *2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*.
- [11] L. Davi *et al.*, "Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks." 2009.
- [12] K. Eldefrawy *et al.*, "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust," in *NDSS 2012*.
- [13] P. England *et al.*, "Riot - a foundation for trust in the internet of things," Tech. Rep., April 2016.
- [14] A. Francillon *et al.*, "A minimalist approach to remote attestation," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14.
- [15] A. Ibrahim *et al.*, "Seed: Secure non-interactive attestation for embedded devices," in *10th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2017)*.
- [16] L. Jäger *et al.*, "Rolling dice: Lightweight remote attestation for cots iot hardware," in *Proceedings of the 12th International Conference on Availability, Reliability and Security*, 2017.
- [17] P. Koeberl *et al.*, "Trustlite: A security architecture for tiny embedded devices," ser. EuroSys '14.
- [18] F. McKeen *et al.*, "Innovative instructions and software model for isolated execution," 2013.
- [19] J. Noorman *et al.*, "Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base," in *USENIX Security 13*.
- [20] H. Raj *et al.*, "ftpm: A firmware-based tpm 2.0 implementation," Tech. Rep., 2015.
- [21] S. Schulz *et al.*, "Boot attestation: Secure remote reporting with off-the-shelf iot sensors," Cryptology ePrint Archive, Report 2017/577, 2017.
- [22] A. Seshadri *et al.*, "Swatt: software-based attestation for embedded devices," in *IEEE Symposium on Security and Privacy*.
- [23] —, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," *SIGOPS*.
- [24] R. Strackx *et al.*, *Efficient Isolation of Trusted Subsystems in Embedded Systems*.
- [25] L. Szekeres *et al.*, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*.