IoT Boot Integrity Measuring and Reporting

Tom Broström
Cyber Pack Ventures, Inc.
5850 Waterloo Road Suite 140
Columbia, MD 21045
tbrostrom@cyberpackventures.com

John Zhu, Ryan Robucci, and Mohamed Younis
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
1000 Hilltop Circle, Baltimore, MD 21250
zhujohn1, robucci, younis@umbc.edu

ABSTRACT

The current era can be characterized by the massive reliance on computing platforms in almost all domains, such as manufacturing, defense, healthcare, government. However, with the increased productivity, flexibility, and effectiveness that computers provide, comes the vulnerability to cyber-attacks where software, or even firmware, gets subtly modified by a hacker. The integration of a Trusted Platform Module (TPM) opts to tackle this issue by aiding in the detection of unauthorized modifications so that devices get remediation as needed. Nonetheless, the use of a TPM is impractical for resource-constrained devices due to power, space and cost limitations. With the recent proliferation of miniaturized devices along with the push towards the Internet-of Things (IoT) there is a need for a lightweight and practical alternative to the TPM. This paper proposes a cost-effective solution that incorporates modest amounts of integrated roots-of-trust logic and supports attestation of the integrity of the device's boot-up state. Our solution leverages crypto-acceleration modules found on many microprocessor and microcontroller based IoT devices nowadays, and introduces little additional overhead. The basic concepts have been validated through implementation on an SoC with an FPGA and a hard microcontroller. We report the validation results and highlight the involved tradeoffs.

Keywords

Trusted platform, IoT, Integrity, Attestation, Security, Measured boot.

1. INTRODUCTION

The massive integration of computers in all aspects of the human life is to be credited for improving manufacturing, trade, healthcare, travel, entertainment, government services, etc. One of the key advantages of incorporating computers is the flexibility and adaptability provided by the software. Yet, cyber-attacks exploit such flexibility through a variety of means to inject malicious modules, e.g., malware, botnets, etc. Not only the operation of the computers could be altered by a malicious software module but also important information could be leaked. The consequences could be dramatic and constitute a major national security threat. For example, altering the control algorithms for a nuclear reactor could lead to a disaster. Moreover, the effect of information leakage can put individuals and nations at risk. For example, letting classified documents and business bids be in the hand of an adversary would be serious security and economic threats. Allowing access to personal data maintained by government agencies will not only violate the privacy of the citizens, but also make them susceptible to physical crimes and identity theft.

One of the subtle venues for injecting malicious modules is through software updates. Being able to load an unauthorized operating system update is the most serious scenario. In fact, with the increased popularity of flash memory, this may be applicable to firmware. Preventing unauthorized updates would naturally be the intuitive approach; yet it is hard to enforce unless access to a device is restricted. Instead, the technical community has demanded that changes in the software and hardware configuration are to be at least detected [1][2]. The Trusted Computing Group (TCG) recognized this need and has established a standard for measuring and reporting platform integrity. The TPM, developed and ratified by the TCG, constitutes the industry-adopted solution for enterprise PCs, servers and embedded systems. The TPM specifications defines how to measure and attest platform integrity. Overall, hardware implementation of the TPM is preferred; many vendors offer TPM chips to be included in the design and is interfaced as a peripheral to the processor.

With the increased popularity of small computing devices and applications of IoT, the need for platform integrity grows both in scale and scope. Basically, miniaturized devices are becoming pervasive and are being employed in a wide range of applications. Most notable among IoT applications are those involving controlling physical processes, often referred to as cyber-physical systems. The role of IoT devices in this type of application covers sensing, computation and actuation; obviously, such a role is quite critical and ensuring the integrity of the configuration and software on these devices is paramount. However, the constrained design of IoT devices makes the incorporation of a TPM unsuitable. Generally, an IoT device is subject to resource, size, power, and cost constraints; therefore, the standardized TPM based solution would not be viable both economically and contextually. Hence, a lightweight approach is needed for IoT devices. The desired approach fundamentally must cope with deign constraints by trading off some of the trust management functionally [3].

The abovementioned issues have motivated the technical community to develop suitable schemes. The introduction of DICE is among the most notable efforts in that regard [5]. DICE, which stands for Device Identifier Composition Engine opts to enable attestation without requiring a TPM in order to limit the required resources. To do so, DICE uses a secrete device identifier to measure the integrity of software modules sequentially during device boot-up; such a process yields a sequence of secrets. The last secret can be checked by the attester against a known value to confirm the device integrity. Unlike the TPM, DICE does not provision for storing integrity measurements and does not support secure attestation. Thus, DICE may expose the IoT device to a replay and impersonation attacks. Basically, if an intermediate

AIoTAS, June 2018, Los Angeles, CA, United States

secret in the sequence is leaked, it becomes possible for malware to impersonate legitimate code by replaying the leaked value.

This paper strives to overcome the shortcoming of DICE and to provide an effective, yet lightweight, solution for ensuring platform integrity of IoT devices. The proposed solution leverages crypto-acceleration modules found on many microprocessor and microcontroller based IoT devices nowadays in order to provide secure attestation services. Fundamentally we provide a wrapper that provisions for secure storage and reporting of platform integrity measurements. Our proposed approach has been validated through implementation on an SoC with an ARM-based hard processor system (HPS) and an Intel/Altera FPGA. The validation results confirm the effectiveness of our approach in terms of the supported functionality and low overhead. We also highlight the involved tradeoffs and provide some design guidelines. To the best of our knowledge, our approach is the first to sustain the key TPM functionality in hardware while addressing the design constraints of IoT devices.

The paper is organized as follows. The next section provides an overview of the TPM and DICE, analyzes the shortcoming of DICE, and outlines the desired features for our approach. Section 3 covers related work in the literature. Our methodology is presented in Section 4. Section 5 discusses the prototype-based validation and reports the testing results. The paper is concluded in Section 6 with a summary of the contribution and future work.

2. Design Goals

This section mainly summarizes the functionality of a TPM and explains how DICE provides platform integrity. An analysis is then provided to highlight the shortcomings of the DICE based methodology and to outline the desirable features for measuring and attesting the platform integrity of an IoT device.

2.1 TPM Overview

While the TPM has grown into a sophisticated and complex hardware security module, as evidenced by its 1,479 page, four-volume specification, it retains a simple objective at its core. Basically, the TPM opts to provide methods for collecting and reporting the identities of hardware and software components that comprise a platform [4]. A software identity is represented by a digest produced from a cryptographic hash of that component. The first digest of mutable code on a platform is produced by the platform's Root of Trust for Measurement (RTM), e.g. in boot ROM. The RTM delivers the digest to the TPM's collector, known as the Root of Trust for Storage (RTS). The RTS extends this and subsequent digests as a cumulative digest of digests (referred to as DoDs in this paper). The RTS stores the DoDs into protected regions known as Platform Configuration Registers (PCRs). Extending a digest in a PCR is done as follow:

$$PCR_{new} = \mathbf{H}\,(PCR_{old} \,||\, digest),$$

where **H** is a secure hash function, '||' is the concatenation operator, and *digest* is the hash of the next software module to be executed in the boot sequence.

Attestation refers to generating a proof to confirm the platform integrity. To build an attestation, the TPM's reporting agent, the Root of Trust for Reporting (RTR), quotes selected PCRs by digitally signing their DoDs. The RTR signs with its Attestation Identity Key (AIK). The AIK is generated by the TPM and signed, in the form of a certificate, by its Endorsement Key (EK). EK may be provisioned by the TPM manufacturer so that assessors who trust the manufacturer can trust the AIK used in attestation. Also, the EK public key (EK_{pub}) could be used as the hardware identity.

During the measured boot process, executable modules, beginning with the RTM, perform the following steps on subsequent modules: load, measure (create a digest by hashing), log, extend, and execute (transfer control). The log of digests built is commonly known as the event log. An assessor can validate the event log by reiterating the *Extend* operations in an attempt to reproduce the DoDs reported in the PCRs. Trust is said to transit from the RTM to subsequent modules in that a trail of evidence in the PCRs and boot log can be used to assess trustworthiness as a whole. Therefore, the sequence of recorded events is described as building a transitive trust chain, or chain of trust. In order to describe characteristics that affect a platform's trustworthiness, the TCG requires three Roots of Trust, namely RTM, RTS and RTR, as described above. The device should respond to attestation confirmation inquiry where a nonce is provided and quote is requested in return. The RTR hashes the concatenation of an assessor's nonce and the DoDs in the PCRs and then encrypts (digitally signs) using AIK_{priv}. The signed attestation and log constitute the device's response to quote request. Figure 1 illustrates the attestation process for a TPM.

A key feature of this architecture is that if a boot module is changed, it will be reflected in the DoDs. If that module were malicious, to allude detection, it would need to modify the DoDs so that it appears benign. Since the only modification permitted is *Extend*, the module would need to calculate a $data_{new}$ value to $H(DoDs_{bad} \,||\, data_{new})$ in order to produce $DoDs_{good}$. This is mathematically infeasible.

2.2 DICE Methodology

DICE is intended to serve a similar role as that of a TPM that is to attest to the trustworthiness of an embedded device [5]. Generally, a TPM could constitute an unwarranted overhead for resource and cost constrained smart sensing and actuation devices that would serve within an IoT. DICE is designed by the TCG for these devices that do not have a TPM. While DICE is fundamentally similar to an RTM, it does not provide RTS and RTR functionality and only supports RTM-like functionality.

While the TPM has its own secrets (e.g., AIK_{priv}), DICE specifies its secret, accessible only to DICE, as a Unique Device Secret

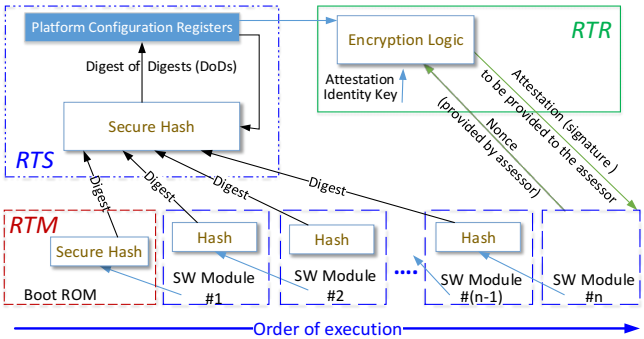

Figure 1: Illustraing how the TPM concept can be applied to boot measurment and attestation.

(UDS, or $Secret_0$). DICE extends the UDS with the digest of the

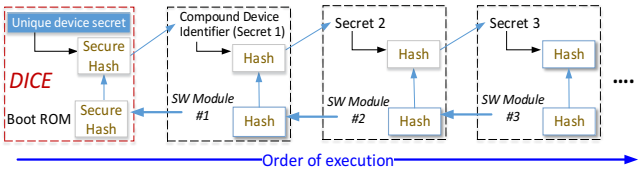

Figure 2: Illustration of DICE in operation. Note the absence of RTS and RTR.

first mutable code. The result, and output of DICE is the Compound Device Identifier (CDI, or $Secret_1$). When $Module_i$ prepares to execute $Module_{i+1}$, it computes $Secret_{i+1}$, destroys $Secret_i$, and passes $Secret_{i+1}$ as summarized by the following equations and illustrated in Figure 2:

$$Secret_0 = UDS,\ Module_0 = DICE$$

$$Secret_{i+1} = \mathbf{H}(Secret_i \,||\, \mathbf{H}(Module_{i+1}))$$

2.3 DICE Analysis

DICE's secrets are akin to the TPM's DoDs as they are produced conceptually in a chain from a trusted root. However, DICE does not provide any protection for its secrets as the TPM's RTS protects DoDs with PCRs. Consequently, the secrets are loosely protected by untrusted (i.e., mutable, non-root-of-trust) code.

Under the TPM scheme, it is generally expected that event logs and DoDs can be freely shared as they are not cryptographic secrets. Under DICE, there is no event log per se as there are no trusted mechanisms to protect and report its integrity. This results in a reduced ability to attest reliably to the boot state of the platform.

Obviously, secrets must not be accessible to unauthorized entities. The loss (i.e. leakage) of a secret makes a DICE-enabled device vulnerable to replay and impersonation attacks [5]. Only the engine and the (initial) UDS secret are protected. Derived secrets are left unprotected. Protection of measurement and attestation assets is vital [4][20][26].

The type of attestation being developed by the DICE working group is Implicit Identity Based Device Attestation [19]. Instead of passing $Secret_2$ to $Module_2$, it generates and passes an asymmetric key pair, namely the Alias Key, derived from the CDI, and the hash of $Module_2$. The result is the same – unprotected secrets passed between modules make the process vulnerable to attack.

2.4 Design Objectives

It is reasonable to expect that certain applications must process secrets and that it is desirable to minimize the attack surface wherever possible. In the case of how to perform a measured boot with attestation, the TCG determined that a specific minimum set of discrete hardware, including the RTS and RTR, were essential [4]. DICE falls short of these requirements and essentially sacrifices robustness to minimize resources. The objective of this paper is to satisfy both the original need for the RTS and RTR and the need to function on a device without a TPM.

3. Related Work

Prior work can be categorized based on how to support attestation into hardware and software schemes. The most notable hardware solutions are TPM [4] and TrustZone [6][7]. However, these solutions are geared for enterprise computing, e.g., servers and desktop systems, are not fit for IoT devices due to cost, size and power constraints. To support portable devices like smart phones, some work has focused on reducing the hardware complexity, or pursued a software-based or a hybrid-software and hardware solutions. For example, M. Kim, et al. [8] have focused on optimizing the hash function implementation within a TPM to suit mobile devices. Meanwhile, N Aaraj et al. [9] promote the use for a software TPM to overcome the cost constraints for embedded devices. The software is to run in a protected execution mode. Kursawe and Schellekens [10] opt to simplify the hardware implementation of a TPM by moving some of the functionality to software. Yet the key complexity in terms of resources and asymmetric cryptography support are not tackled and the solution does not suit IoT devices.

The TPM concept is further extended to support dynamic system status [11]-[13]. In [11], the idea is to enable attesting to the correctness of the control flow within a program in order to detect runtime attacks, e.g. stack overflow. The solution, which is hardware based, is geared for cyber-physical systems where the integrity of the control software should be monitored both at bootstrapping and at runtime. LeMay and Gunter [12] tackle the same issue through a software-based approach through the inclusion of a micro-kernel that is not remotely upgradeable. A similar objective is targeted in [13] to factor in the data generated by the program rather than the flow of execution. Again these solutions are not geared for resource-constrained IoT devices. In addition, continual attestation is unwarranted for IoT devices unless they perform actuation within mission critical applications. We argue that our methodology makes hardware-based attestation both effective and efficient.

Supporting attestation for resource-constrained devices has also received attention from the research community. R. N. Akram, et al. [14] have studied attestation for smart cards, which are becoming very popular in mass transient, university campus, etc. DICE, as discussed earlier, is endorsed by the TCG. Variants to DICE are also being pursued, e.g. Microsoft RIOT [15]. Again, all these solutions are software based and trade off resources for robustness by leaving out the RTS and RTS functions. On the other hand, some published work opts to mitigate such shortcoming by using heterogeneous setups where attestation of IoT devices involves unconstrained devices as well. For example, MTRA is a Multiple-Tier Remote Attestation protocol that supports IoT setups which involve heterogeneous devices in terms of the computational resources [16]. Some of the devices are assumed to be equipped with TPM hardware. Meanwhile, the attestation of resource-constrained devices is handled through software. Basically, a TPM-equipped device will act as an assessor for each of the resource-constrained devices that are within its communication range.

Cooperative attestation is another methodology for overcoming the resource constraints of IoT devices. For example, in [17], devices are grouped into clusters and each cluster is considered as an entity whose trust is to be attested. Devices within a cluster run distributed consensus algorithms to attest to their individual trust. The attestation here is based on a simple operation such as reporting a sensor value. Since the devices are homogenous, the distributed consensus algorithm, e.g., majority voting, could be used to determine outliers and deem them as untrusted. A full TPM-based attestation is then performed at the inter-cluster level, where the individual operations are divided among the cluster members. Instead of asymmetric cryptograph, a residue number

system (RNS) based homomorphic share scheme is used to enable the generation of a combined cluster-based attestation. We argue that cooperative attestation could be impractical and would make security hard to ensure. Our approach opts to enable hardware support while coping with the resource limitation of the devices.

4. Proposed Architecture

The cryptographic primitives found in the TPM to support the RTS's *Extend* and the RTR's *Quote* operations are hash (e.g., SHA) and public key (asymmetric) cipher to produce a digital signature (e.g., RSA). DICE, on the other hand, omits the RTS and RTR. Our objective in this paper is to develop a solution that provisions the RTS and RTR support, while staying conscious of resource concerns for IoT devices. Our methodology is to pursue symmetric rather than asymmetric cryptography and repurpose on-chip crypto accelerators to also function as RTS and RTR. Since the bulk of encryption needs are met with faster symmetric ciphers rather than slower asymmetric alternatives, it is becoming increasingly common to find symmetric algorithms integrated into microcontroller hardware. For example, variations on the ST STM32L081xx, TI MSP430 and the Atmel XMEGA microcontrollers have embedded AES acceleration. We argue that our methodology strikes a balance between overhead and robustness and would suit resource-constrained IoT devices.

In the balance of this section, we show how integrated hash and symmetric cipher accelerators can be augmented to provide the needed roots of trust in hardware. This augmentation can be thought of as a "wrapper" with virtually no disruption to the core crypto accelerator logic, or "engine". The wrapper would function as a trusted, special-purpose interface to the general-purpose accelerator. Thus, as manufacturers dedicate silicon to these accelerators, they could find that a fractional increase could provide an important layer of trust attestation for very little cost. We confirm that in Section 4 when discussing our prototype-based validation of our methodology.

4.1 Supporting Extend Operation

While the TPM supports multiple PCRs and multiple hashing algorithms, our solution will be stripped down to a single PCR and a single hashing algorithm. Note that the PCR is to be protected from arbitrary writes, and may only be updated with Extend. The initial value of PCR is 0, and may only be reset with a full platform reset.

Figure 3 illustrates a notional implementation of *Extend* using a generic hash engine. The blue boxes represent the unmodified hash core and the digest register. The orange box and arrows represent the additional logic (i.e., wrapper) added to perform an *Extend*. The wrapper includes a PCR and a small state machine

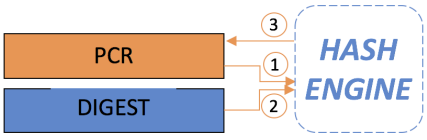

Figure 3: Hash engine with Extend wrapper

that control the operation of the hash engine. It is assumed that the digest to be extended has already been produced. The operation proceeds as follows: (1) the PCR and (2) DIGEST are effectively concatenated by feeding them sequentially to the hash engine where **H**(PCR||DIGEST) is generated, and (3) the result is written back to the PCR.

4.2 Supporting Quote Operation

Instead of quoting multiple PCRs with one of the TPM-supported asymmetric ciphers, our approach will quote one PCR using a symmetric cipher. This will require the device and the assessor of the attestation to possess, or otherwise indirectly use, a shared secret. The attestation is built by signing (or encrypting) the hash of the concatenation of the PCR and an assessor's challenge nonce, i.e., E(H(PCR||NONCE)).

Figure 4 illustrates the operation of QUOTE, which proceeds as follows: (1) the PCR and (2) the challenge nonce are fed to the hash engine sequentially, and (3) hashed to produce a digest as H(PCR||NONCE) that is (4) sent to the cipher's input block, the (5) device secret is place in the cipher's key to (6) encrypt the digest, (7) the key is zeroed to prevent unauthorized use, and (8) the resulting signature, E(H(PCR||NONCE) is delivered to the host.

5. Prototype

To demonstrate the viability of our lightweight attestation method, a prototype was implemented to gain an understanding of the resource consumption necessary for each part of the method. The prototype was built using the DE1-SoC Development Kit, which is hardware design platform built around the Altera System-on-Chip (SoC) FPGA and combines the ARM dual-core Cortex-A9 hardcore processor. The ARM processor is responsible for invoking *Extend* and *Quote* during and after measured boot, initially from ROM. It also represents the less secure portion of an embedded device, as it is vulnerable to changes in software. The FPGA will be used to implement the secure hardware portion of the methods that perform *Extend* and *Quote*, which would otherwise be constructed in an ASIC.

5.1 Crypto Accelerators

Ideally, the implementation of the wrappers should depend very little on the choice of hash and cipher algorithms, as the focus is on producing lightweight wrappers. In our initial *conventional* prototype, the SHA256 hash [27] and AES cipher algorithms were selected. Like the *optimized* prototype presented in section 6, this prototype produces a 128-bit signature, but from a truncated 256-bit digest as depicted in Figure 4, step 4. Truncation of digests is addressed in Section 7 of [27].

5.2 Hash Wrapper

The hash wrapper is used to execute *extend* operations by

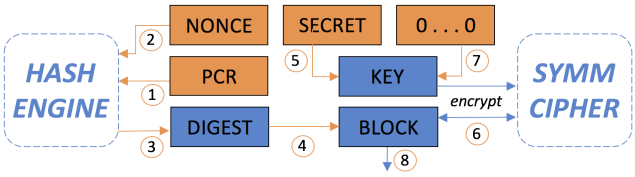

Figure 4: Hash and cipher engines with *Quote* wrapper

controlling the SHA256 engine to hash host modules while preventing important information, such as the PCR, from being maliciously altered by the software. To accomplish this task, the

wrapper is implemented as a state machine which receives signals from the ARM processor and controls the hardware execution.

Figure 5 illustrates the hardware setup of the state machine and how it interacts with the multiplexer, and the SHA256 engine. The state machine acts as the arbiter between the software and hardware; it receives the *hash*, *extend*, *overwrite*, and *end* signals from the ARM processor and then controls the hardware components to produce the output necessary for the request. This layout prevents the software from having direct control of the SHA engine. To control the input of the SHA engine, two 32-bit input multiplexers are used. In this way, the SHA engine can be fed either a message input from the ARM processor for normal operation or be fed register values directly for *Extend* or *Quote* operations. The counter is used to control 32-bit data slicing of the 256 bit inputs. For example, when performing an *Extend* operation, the state machine controls the multiplexer and counter to first feed the 256-bits PCR, then 256-bits Digest to the SHA engine in 32 bit blocks in the correct order. It is important to note that the *Quote* signal is present in the diagram, although *Quote* is technically part of RTR. The need for this signal will be explained in the next section.

Because the goal of the prototype is to minimize the resource consumption, the data width of the *HDAT_input* is set to 32 bits at a cost of performance speed. For example, in a previous implementation, the data width was set to 512 bits because SHA256 processes 512 bits at a time. This allowed the engine to load the message in one clock cycle. However, this incurred large costs since the multiplexer had to have 512 bits inputs, which greatly increases the combinational logic. In addition, it required a 512-bit register within the SHA engine to prevent race conditions, which grows the number of needed registers. Thus, the *HDAT_input* was decreased to 32 bits. This number was selected because SHA256 needs to process a minimum of 32 bits every clock cycle. While such data slicing increases the number of clock cycles needed to load the message from one to sixteen, it greatly reduces the size of the multiplexer and register.

The PCR is protected from malicious software attacks through two methods. First, the PCR only reads the output from the SHA engine. It does not have any other input. Second, the *enable* signal of the register is controlled by the state machine. With these two methods, an attacker, i.e., though malicious software, cannot overwrite the PCR with their own value. An attacker also cannot

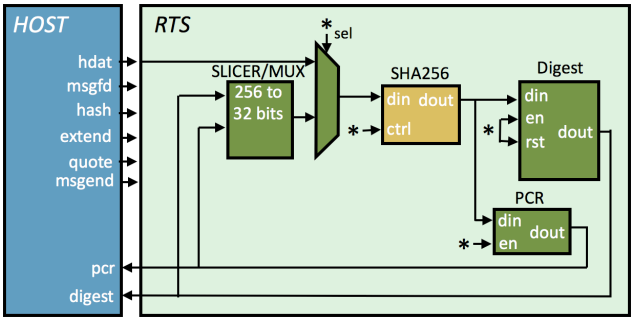

Figure 5: Data Path of the Hash Wrapper

control the input of the PCR register, nor is able to control the *enable* signal to allow overwriting the PCR.

5.3 Cipher Wrapper

The cipher wrapper is used to execute *Quote* operations using a lightweight symmetric-key algorithm, which is AES in this prototype. The goal is to sign the PCR securely. Doing so requires protection of the secret key, the PCR contents, and the signing mechanism. Such protection is accomplished by using hardware to sign the PCR with the symmetric key instead of software.

Figure 6 illustrates the data path of the cipher wrapper and its interface with the ARM processor. The main components are the state machine, counter (not shown), data slicers, and multiplexer, and the AES engine [32]. The state machine is used to arbitrate between the ARM processor and the FPGA to prevent direct control of the AES engine during sensitive operations. It receives signals *encrypt, decrypt* and *quote* from the ARM processor, and in turn controls the other components. The multiplexer is used to control whether the plaintext input of AES is fed messages from ARM processor or the H(PCR || NONCE). This allows for general purpose use of the AES engine or for *Quote* operations. The counter and the data slicers are slightly different from their counterparts in the hash wrapper. For example, the cipher wrapper's counter only needs a data width of three instead of four because the engine deals with a maximum data width of 256, or eight 32-bit blocks.

When performing *Quote* operations, the state machine requests the digest of PCR||NONCE from the hash wrapper through the *hash_sig* signal. If the RTS wrapper is idle, it will hash the PCR||NONCE by controlling its multiplexer to feed in the PCR||NONCE. Once a valid digest is calculated, the wrapper will read in the leftmost 128 bits and encrypt it with the secret key. To prevent race conditions, *hash_sig* signal is kept high to prevent any other hash or *Extend* operations from being executed until the *Quote* operation is complete. This prevents the PCR||NONCE digest from accidentally being rewritten with a different value.

To interface with the processor, the 128-bit plaintext input are fed to the AES engine in multiple 32-bit blocks with the use of the data slicer and counter. The data slicer extracts 32 bits from the input, while the counter controls which 32 bits are extracted. While this increases the clock cycles required to load the input message, it reduces the multiplexer from having 128 bit data width to only 32 data width.

5.4 Software

Preferably, the measurement chain should be started by a Root of Trust for Measurement. Since the objective is only to restore the

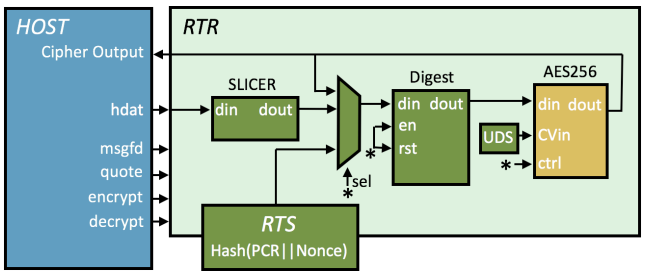

Figure 6: Data Path of the Cipher Wrapper

RTS and RTR that would have otherwise been provided by a TPM, the prototyping does not focus on providing an RTM. On the selected hardware, the Boot ROM could have been written to

function as an RTM. Since the Boot ROM is literally stored in read-only memory, the measurement chain begins in the code loaded and executed by the Boot ROM, i.e., the Secondary Program Loader (SPL).

The SPL (40KB), like the Boot ROM (24KB), is limited to 64KB as both the on-chip RAM and ROM are 64KB each. The SPL's main purpose is to initialize the memory controller to access the external SDRAM (1GB), and locate, load and execute the next stage, which is the much larger (235KB) U-Boot. The SPL was augmented to create a DevID, initialize the RTS/RTR, hash U-Boot, extent its digest and initialize an event log with that digest. The purpose of DevID is to allow an assessor to lookup the corresponding shared secret. DevID is calculated by quoting "DeviceIdentifier" before any extends (i.e. the PCR is 0). The address of the event log and DevID is passed to U-Boot via register R0.

SPL and U-Boot are both part of the Das U-Boot Universal Boot Loader [21] suite. The source was obtained from within the Altera Embedded Design Suite [22], as patched by the Terasic System CD build process [23]. U-Boot is a highly configurable, interactive, scriptable boot loader. Loading operations may occur from many locations for many purposes. Support is decentralized, making insertion of appropriate measure/log/extend operations tedious. Nevertheless, sufficient hooks were placed to permit the measured loading of a Linux operating system – in this case, Ångström. These measurements include the device tree blob (DTB) [24] and zImage [25], and are appended to the event log received from SPL.

To boot Linux, the address of zImage and the DTB are specified in the "bootz" command. After the DTB has been loaded, and measured, U-Boot inserts an "elog" node with entries for each measurement in the log into the DTB. Once Linux is booted, the event log is conveniently available in /proc/device-tree/elog/.

The attestation process was simulated using the process shown in Figure 7. After receiving an attestation response, the assessor can verify the quote as follows: (1) retrieve the shared secret from the provisioning process based on the DevID, (2) decrypt *sig*, compute *H(PCR||NONCE)* and verify that they are equal (if so, the PCR value can be trusted to have been produced by a known

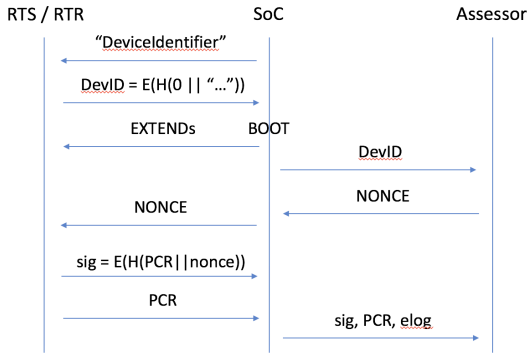

Figure 7: Summary of the attestation process in the validation prototype

device in response to this attestation request). If the PCR is unknown, (3) re-compute the DoDs from elog and compare to the reported PCR value (if equal, the event log is intact). The assessor can then make a device trust decision based on the DoDs and/or individual digests, e.g. by looking them up in known-good or known-bad databases. This process is illustrated in Figure 8.

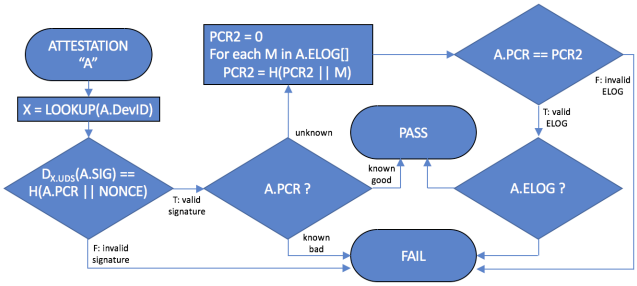

Figure 8: Notional assessment process.

6. Additional Optimizations

The prototype discussed above has been designed to be true to the TCG's modus operandi. In this section we propose resource optimizations for decreasing the impact of adding *Extend* and *Quote* as a root of trust to a device.

6.1 Resource Reduction

- Elimination of conventional hash: The largest component of our initial prototype is the SHA-256 hash engine. Since it has been shown that block ciphers can be made to support hashing [29], the complexity of our solution can be greatly reduced by eliminating the conventional hash engine, and rewiring the state machine to use the block cipher for hashing. For that, we used the Davies-Meyer (DM) approach as illustrated in Figure 9. Message fragments, M_i, are used as encryption keys in this scheme.
- Use of lightweight cipher: While AES is very popular, it was not specifically designed for resource constrained environments. We selected the Simon cipher [28] for the optimized prototype as it requires few hardware resources, and was readily available from, though never required by, our sponsor. Its performance also happens to be tuned for implementing in hardware.
- Compression Consolidation: The method by which hash algorithms, such as SHA, produce message digests is via Merkle-Damgård (MD) construction [30]. MD operates on a function that compresses two inputs into one output. To hash a sequence of message fragments (M), the process is repeated where one input (D) to the compression function (C) is the output of the previous round. This can be summarized by D_i =

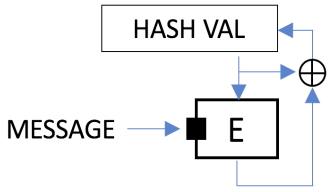

Figure 9: Using the Davies-Meyer approach for hash implementation, where $H_i = E_{Mi}(H_{i-1}) \oplus H_{i-1}$

$C(D_{i-1}, M_i)$. The TCG extends digests to PCRs in a similar manner, as follows: $PCR_i = H(PCRi_{-1} \| D_i)$ with $PCR_0 = 0$. In the case of SHA256, the *H* operation is $C(C(IV, PCRi_{-1} \| D_i)$,

PAD). That means that for every extend, an IV (initialization value) and PAD are compressed into each intermediate result when they should only be used once. To do so, we consolidate the algorithms with: $PCR_i = C(PCRi_{-1}, D_i)$ where $PCR_0 = IV$. Padding could be compressed in when the PCR is used, i.e. during *Quote*.

- Elimination of Decrypt Logic: If a device only requires the abilities to *Extend* and *Quote* for measurement and attestation, and does not require full, general-purpose crypto acceleration, then the decrypt function could also be eliminated. Only the encrypt portion of the cipher is required for signing and to implement the DM hash.

6.2 Optimized Prototype

Figure 10 illustrates the data path for the optimized prototype. The most noticeable difference between this approach and our initial prototype is the integration of the wrappers, where only one state machine controls the various components. The state machine arbitrates signals *hash, extend,* and *quote* from the host processor. It reads in those signals, and then controls the data slicer, counter (not shown), and multiplexer to feed in the necessary inputs to the Simon encrypt engine to perform hash, *Extend* and *Quote* operations.

To perform a hash, *selDI* and *selCV* are used to control the multiplexers that feed into *din* (plaintext) and *cvin* (key) inputs of the Simon engine. These signals control the multiplexers to feed *DSR* (Digest and Signature Register) data to *din* while *hdat* input from the host processor is fed to *cvin* input. The Data slicer and Counter components are used to feed the 128-bit messages to Simon in 32-bit blocks in the correct order. This reduces the size of multiplexers as stated earlier. Once the *Ciphertext* output is calculated, the *selDI* signal is set so that the XOR unit is fed *DSR* as one of the inputs. Thus, *DSR* is written with $E_{mi}(H_{i-1}) \oplus H_{i-1}$.

Executing an *Extend* resembles the hash operation except for two differences. First, *selDI* and *selCV* are set so that PCR data is fed to the Simon engine's *din* input, while *DSR*, which contains the digest to extend, is fed to *cvin* input. Second, once *dout* is calculated, it is XORed with and written to the *PCR* instead of *DSR*. This is accomplished through signals *selDI* and *en_PCR*.

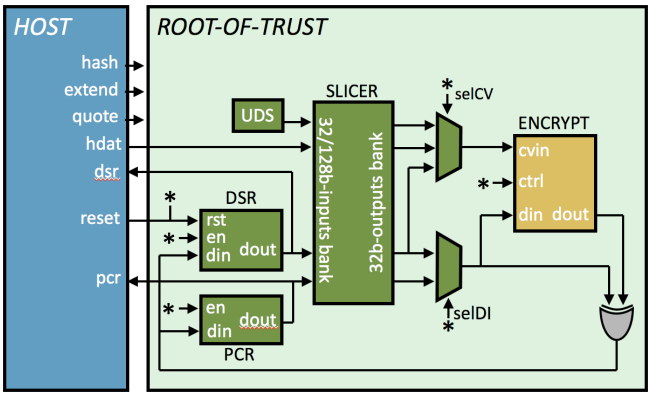

Figure 10: Data Path of Optimized Implementation

However, a *Quote operation* is more complex because it requires both hash and encrypt operations. It requires the hash of PCR and a NONCE that is provided by the host via hdat. This is accomplished by setting *selDI* and *selCV* to feed PCR to Simon engine's *din* and hdat to *cvin*. The *Ciphertext* output is then fed to the XOR gate with *DSR* to produce the necessary digest, which will be stored in DSR. The *DSR* data is then encrypted with the Unique Device Secret (UDS) through *selDI* and *selCV*. After feeding *DSR* to the Simon engine, *DSR* data is reset to 0's. This is done so that once the *Ciphertext* output is valid, we can set *selDI* so that output is XORed with the *DSR* data, which is all zeros. Any value XORed with zeros will return that value unchanged. In other words, the output of the Simon engine, which is the encrypted signature, will not be altered by the XOR unit and stored in *DSR*. The register will contain the encrypted signature, which is then read by the host.

The digest and PCR lengths were reduced from 256 to 128 bits, not in an attempt to optimize by sacrificing security, but to fit more naturally with the available accelerator's block size. Smaller devices that provide crypto acceleration may only provide, for example, AES-128. Also, the largest block size in the reference Simon implementation is 128 bits.

7. Results

The designs of the crypto cores and wrappers supporting hardware-rooted-trust measurement and remote attestation were implemented on an Intel Cyclone V SoC FPGA, as peripherals for the in-built ARM processor. The measurement units used for quantifying the resource consumption of the FPGA design are the Adaptive Logic Module (ALM) and Block Memory Bits (BMB). ALMs are the building blocks of the FPGA. Each ALM comprises an 8-input adaptive look table (ALUT) to implement general combinatorial logic, four storage registers, and two adders. On the other hand, BMB is a measure of the amount of *embedded memory* block resources used. For Intel FPGAs, *embedded memory* refers to the dense dedicated memory structures provided on FPGAs, which provide an alternative to exhausting a large amount of ALMs for implementation of memory (i.e. distributed memory). In Table 1, the required resources to implement the *Extend* and *Quote* wrappers for the initial (conventional) and the optimized prototypes are provided. For the design of the conventional approach that includes AES and SHA hardware that could be found on a microcontroller with hardware crypto support, the size of the additional wrappers was 442 ALMs, representing a 26.7% increase over its crypto accelerator engines' 1657 ALMs. For the design of the optimized approach using a lighter Simon-Encrypt hardware module, the ALM count was reduced by 87%, while reducing the number of BMB to 0. For reference, the Simon hardware with wrappers together would be roughly 1/3 the size of an OpenMSP430 [31].

APPROACH	ENGINE	CORE	WRAPPER	TOTAL
CONVENTIONAL	SHA-256	1009\ 384	256\0	1265\ 384
	AES-256	648\75776	186\0	834\75776
	Total	1657\76160	442\0	2099\76160
OPTIMIZED	Simon	106\ 0	165\0	271\ 0

Table 1: ALM\BMB Consumption of Conventional and Optimized Prototypes

8. Conclusions

Advances in microelectronics have made it possible to develop small-scale computational platforms that can also be internetworked to form an IoT and serve numerous applications.

The increased popularity of IoT motivates the need for lightweight, yet robust, attestation of the integrity of the computing platform of the involved devices. Existing trust attestation technologies, such as the TPM, would impose unacceptable cost and power overhead to the IoT devices and would be thus deemed unsuitable. The TPM alternatives proposed by the technical community, such as DICE, leave out key security features, namely RTS and RTR, in order to reduce complexity. In this paper we have proposed viable, hardware-based approaches for protecting the measurement and attestation process on IoT devices with low overhead. Our methodology leverages crypto accelerators found on many microprocessors and microcontroller based systems and incorporates a lightweight state machine to drive them securely.

9. ACKNOWLEDGMENTS

Our thanks to the Information Assurance Research Group in the National Security Agency's Research Directorate who sponsored this work.

10. REFERENCES

[1] Wang, J., et al., Survey on key technology development and application in trusted computing. China Communications, 13, (2016), 70-90.

[2] Kinney, S. Trusted Platform Module Basics 1st Ed.: Using TPM in Embedded Systems. (2006) Elsevier.

[3] Shepherd, C. et al., Secure and Trusted Execution: Past, Present, and Future - A Critical Review in the Context of the Internet of Things and Cyber-Physical Systems. in Proceedings TRUSTCOM'16, (Tianjin, China, 2016), IEEE.

[4] Trusted Platform Module Library, Part 1: Architecture, Family "2.0", Level 00 Revision 01.38, (September 29, 2016), section 9.5.5 Integrity Measurement and Reporting

[5] Trusted Platform Architecture Hardware Requirements for a Device Identifier Composition Engine. https://www.trustedcomputinggroup.org

[6] ARM, ARM Security Technology, Building a Secure System using TrustZone Technology, ARM White Paper, 2009.

[7] Samsung Electronics, White Paper: Samsung KNOX Premium, Samsung Electronics, Sep. 2014.

[8] Kim, M., D. G. Lee, and J. Ryou. Compact and unified hardware architecture for SHA-1 and SHA-256 of trusted mobile computing. Personal Ubiquitous Comput. 17, 5 (June 2013), 921-932.

[9] Aaraj, N. A. Raghunathan, and N. K. Jha. Analysis and design of a hardware/software trusted platform module for embedded systems. ACM Trans. Embed. Comput. Syst. 8, 1, Article 8 (January 2009), 31 pages.

[10] Kursawe, K. and D. Schellekens. Flexible µTPMs through disembedding. In Proceedings of ASIACCS '09 (Sydney, Australia, March 2009), ACM, 116-124.

[11] Das, S., W. Zhang and Y. Liu, Reconfigurable Dynamic Trusted Platform Module for Control Flow Checking. in Proceedings of the Annual Symposium on VLSI (Tampa, FL, 2014), IEEE press, 166-171.

[12] LeMay, M., and C. A. Gunter, Cumulative Attestation Kernels for Embedded Systems, IEEE Transactions on Smart Grid, 3, 2, (2012), 744-760.

[13] Gu, L., et al.. Remote attestation on program execution. In Proceedings of STC '08 (Alexandria, VA, October 2008). ACM, 11-20.

[14] Akram, R. N., K. Markantonakis and K. Mayes, Trusted Platform Module for Smart Cards. in Proceedings of NTMS'14 (Dubai, UAE 2014), IEEE press, 1-5.

[15] England, P., et al. RIoT - A Foundation for Trust in the Internet of Things, Technical report MSR-TR-2016-18, Microsoft Corp, April 2016.

[16] Tan, H., G. Tsudik and S. Jha, MTRA: Multiple-tier remote attestation in IoT networks. in Proceedings of CNS'17 (Las Vegas, NV, 2017), IEEE press, 1-9.

[17] Hamadeh, H., S. Chaudhuri and A. Tyagi, Area, Energy, and Time Assessment for a Distributed TPM for Distributed Trust in IoT Clusters. In Proceedings of iNIS'15 (Indore, India, December 2015), *IEEE press,* 225-230.

[18] Foundational Trust for IoT and Resource Constrained Devices. https://trustedcomputinggroup.org

[19] Implicit Identity Based Device Attestation (draft). https://trustedcomputinggroup.org, Nov 2017.

[20] Maat: A Platform Service for Measurement and Attestation, arXiv:1709.10147 [cs.CR], https://arxiv.org/abs/1709.10147

[21] Das U-Boot -- the Universal Boot Loader, https://www.denx.de/wiki/U-Boot/WebHome, DENX Software Engineering

[22] Altera SoC EDS Standard edition, v17.0, http://dl.altera.com/soceds/17.0/?edition=standard, host_tools/altera/preloader/uboot-socfpga.tar.gz

[23] DE1-SoC CD-ROM (rev.F Board), v5.1.1, ../Demonstrations/SOC_FPGA/de1_soc_GHRD/software/spl_bsp/Makefile. http://www.terasic.com.tw

[24] u-boot-2017.11-rc3, arch/arm/dts/socfpga_cyclone5_de1_soc.dtb

[25] https://releases.rocketboards.org/release/2017.10/gsrd/bin/linux-socfpga-gsrd-17.1std-cv.tar.gz (sdimage.tar.gz)

[26] BIOS Integrity Measurement Guidelines (Draft), NIST SP 800-155.

[27] Secure Hash Standard, FIPS PUB 180-4, https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[28] The Simon and Speck Families of Lightweight Block Ciphers, https://eprint.iacr.org/2013/404.pdf

[29] Winternitz. R., A secure one-way hash function built from DES. In Proceedings of the IEEE Symposium on Information Security and Privacy, (1984) IEEE Press, 88-90.

[30] Merkle. R.C., Secrecy, authentication, and public key systems. Stanford Ph.D. thesis 1979, pages 13-15.

[31] openMSP430 :: Area and speed analysis, https://opencores.org/project,openmsp430,area%20and%20speed%20analysis

[32] Avalon AES ECB-Core (128, 192, 256 Bit), https://opencores.org/project/avs_aes