Delegated Attestation: Scalable Remote Attestation of Commodity CPS by Blending Proofs of Execution with Software Attestation

Mahmoud Ammar*
Huawei Technologies
Munich, Germany
mahmoud.ammar@huawei.com

Bruno Crispo
Imec-DistriNet, KU Leuven & University of Trento
Trento, Italy
bruno.crispo@unitn.it

Ivan De Oliveira Nunes
University of California, Irvine
CA, USA
ivanoliv@uci.edu

Gene Tsudik
University of California, Irvine
CA, USA
gene.tsudik@uci.edu

ABSTRACT

Remote Attestation ($\mathcal{R}$A) is an interaction between a trusted verifier ($\mathcal{V}$rf) and one or more remote and potentially compromised devices (provers or $\mathcal{P}$rv-s) that allow the former to measure the software state of the latter. $\mathcal{R}$A is particularly relevant to safety-critical cyber-physical systems (CPS) where a set of low-end micro-controllers (MCUs), operate under the control of a remote and more powerful controller. In such cases, $\mathcal{R}$A is an effective and relatively efficient means to detect software compromise, e.g., malware infections, on these low-end MCUs that cannot support expensive security mechanisms.

Unfortunately, current $\mathcal{R}$A techniques have a major practical limitation by requiring one or more of: **(i)** customized hardware support on every $\mathcal{P}$rv; or (ii) physical presence of $\mathcal{V}$rf, which must be directly connected to all $\mathcal{P}$rv-s, or (iii) secure pre-loading of trusted software. To overcome this limitation, we construct, implement and evaluate a *Delegated Attestation* ($\mathcal{D}$A) scheme. In it, all $\mathcal{P}$rv-s remain unmodified, meaning that it applies even to already deployed CPS devices. However, there is no requirement for $\mathcal{V}$rf's physical proximity, hardware support or secure initial software pre-loading. Instead, $\mathcal{D}$A uses a recently proposed primitive (called Proofs-of-Execution or PoX) which entails a single customized low-end embedded device (called Attestation Proxy or $\mathcal{P}$rox) on the same local-area network (e.g., a CAN-bus) as $\mathcal{P}$rv-s. In doing so, $\mathcal{D}$A enables $\mathcal{R}$A of all $\mathcal{P}$rv-s. Importantly, this guarantee holds even if $\mathcal{P}$rox is itself compromised, enabling secure $\mathcal{R}$A of completely unmodified $\mathcal{P}$rv-s in an efficient manner.

CCS CONCEPTS

• **Security and privacy → Intrusion/anomaly detection and malware mitigation**.

KEYWORDS

Proofs of execution, remote attestation, Malware Detection

ACM Reference Format:
Mahmoud Ammar, Bruno Crispo, Ivan De Oliveira Nunes, and Gene Tsudik. 2021. Delegated Attestation: Scalable Remote Attestation of Commodity CPS by Blending Proofs of Execution with Software Attestation. In *Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21), June 28–July 2, 2021, Abu Dhabi, United Arab Emirates.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3448300.3467818

1 INTRODUCTION

Many everyday settings are increasingly instrumented with various embedded devices, forming so-called Cyber-Physical Systems (CPS). These "smart" or IoT devices can be found in automotive systems, homes, offices, factories, and public venues, often performing safety-critical tasks.

A typical CPS (e.g., in an automobile) is usually composed of multiple sensors and actuators, at the core of each is a low-cost micro-controller unit (MCU). Such MCUs typically run simple software, often atop "bare metal". In other words, they lack a Memory Management Unit (MMU) or a Memory Protection Unit (MPU) and consequently cannot host a microkernel or a hypervisor. They tend to be operated by a more sophisticated remote central control unit and, despite their potential importance to overall system functionality, these devices are typically designed to minimize cost, physical size and energy consumption. Unsurprisingly, their architectural support for security is rudimentary or non-existent. At the same time, burgeoning popularity of CPS/smart/IoT devices greatly increases the overall attack surface, i.e., they are attractive – and often easy – attack targets, as illustrated by massive-scale attacks, such as the Mirai botnet [3]. This prompts a fundamental problem: considering that vast numbers are already deployed, how to secure these devices, ideally, without requiring any physical modifications?

*This research was performed while the author was with KU Leuven.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WiSec '21, June 28–July 2, 2021, Abu Dhabi, United Arab Emirates
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8349-3/21/06…$15.00
https://doi.org/10.1145/3448300.3467818

Given a choice between preventing and detecting attacks, the former is obviously preferable. However, while prevention is somewhat effective on general-purpose computing platforms (smartphones, laptops, desktops, servers), it is untenable on low-end devices that simply cannot host sophisticated (and constantly evolving) anti-malware tools. Consequently, the only alternative is detection of compromise, which requires some form of periodic probing or measurement of the subject device. This is generally referred to as *Attestation*.

1.1 Attestation & Friends: Overview & Related Work

Attestation is a security service that allows a trusted verifier = $\mathcal{V}$rf (e.g., a CPS controller) to securely measure internal state of an untrusted and possibly compromised prover = $\mathcal{P}$rv (e.g., a sensor).

Based on the required hardware support on $\mathcal{P}$rv, attestation techniques are usually classified as: software, hardware or hybrid. Software attestation [5, 7, 14, 21, 23, 30–33, 36] (also known as *unconditional attestation* [16]) does not require any hardware features. Thus, it is applicable to commodity devices used in current and legacy CPS. Unfortunately, it has an important drawback by assuming direct (one-hop) constant-delay communication between $\mathcal{P}$rv and $\mathcal{V}$rf, which is impractical for many CPS applications. In contrast, hardware and hybrid attestation make no assumptions about the network or communication, allowing so-called Remote Attestation ($\mathcal{R}$A), meaning that $\mathcal{V}$rf and $\mathcal{P}$rv can be arbitrarily connected. However, hardware $\mathcal{R}$A [24–26, 28, 29] relies on $\mathcal{P}$rv having a dedicated (though, not $\mathcal{R}$A-specific) secure hardware component, such as a TPM [35], ARM TrustZone [4], or Intel SGX [18]. This is something that low-end devices cannot support. Whereas, hybrid $\mathcal{R}$A [6, 8, 9, 11, 12, 22] uses hardware-software co-design to impose minimal hardware requirements on $\mathcal{P}$rv, which means that each $\mathcal{P}$rv must be manufactured with some (however minimal) $\mathcal{R}$A hardware features. This obviously makes hybrid $\mathcal{R}$A unworkable for already deployed and already manufactured devices. SIMPLE [2] is another recent $\mathcal{R}$A type falls somewhere in between hybrid and *traditional* software $\mathcal{R}$A. It enables $\mathcal{R}$A while requiring no hardware modification or support, meaning that it is applicable to commodity devices. However, it assumes secure initial provisioning (programming) of $\mathcal{P}$rv, rendering it inapplicable to previously deployed devices.

In general, $\mathcal{R}$A by itself is only a stepping stone: if $\mathcal{V}$rf determines that $\mathcal{P}$rv is in a good state, no further action is needed. However, if compromise is detected, $\mathcal{P}$rv needs to be healed somehow. This may require one or more of $\mathcal{V}$rf-initiated actions: device reset, memory erasure, and software/firmware update. Also, it might be necessary for $\mathcal{P}$rv to verifiably execute some $\mathcal{V}$rf-supplied code. At least one of such services must be securely supported on $\mathcal{P}$rv in order for $\mathcal{R}$A to be truly useful.[1]

1.2 Focus & Contributions

This paper introduces the concept of Delegated Attestation ($\mathcal{D}$A) to bridge the gap between remote and one-hop (i.e., software) attestation, thus making attestation services more practical and relevant to real-world scenarios, including currently deployed systems. We construct a $\mathcal{D}$A technique which takes advantage of a recently proposed primitive, called Proof-of-Execution (PoX) [9]. It involves a single customized low-end embedded device, called an attestation proxy ($\mathcal{P}$rox), located on the same local-area network (e.g., a vehicular CAN-bus) as the **unmodified** and already deployed $\mathcal{P}$rv-s that are only capable of software attestation. This enables collective $\mathcal{R}$A of unmodified $\mathcal{P}$rv-s in a cost-effective, scalable, and secure manner. Notably, security guarantees hold even in case of a full software compromise on $\mathcal{P}$rox. This work makes the following contributions:

- Design of $\mathcal{D}$A: a new security service for attesting deployed commodity devices without requiring any hardware support or pre-deployment secure programming on the $\mathcal{P}$rv-s.
- Demonstration of $\mathcal{D}$A applicability, composing it with two $\mathcal{P}$rv's instances: traditional software attestation (i.e., [30]) and a novel keyless version of SIMPLE [2]. We discuss trade-offs between both approaches.
- Implementation and evaluation of the proposed $\mathcal{D}$A technique on a realistic test-bed with $\mathcal{P}$rox and multiple $\mathcal{P}$rv-s communicating over a CAN-bus. This setting emulates safety-critical CPS, e.g., in vehicular and other automation systems. We also show that $\mathcal{D}$A is both efficient and effective for attesting several types of devices.

1.3 Low-End Devices

This paper focuses on tiny $\mathcal{P}$rv-s, i.e., embedded devices with low computing power. These are some of the smallest devices based on low-power single-core MCU-s with only a few KBytes of program and data memory (such as Atmel AVR ATmega and TI MSP430), with 8- and 16-bit CPUs, typically running at 1-16MHz, with ≈ 64KBytes of addressable memory. SRAM is used as data memory, ranging in size between 2 and 16KBytes, while the rest of address space is available for program memory. Such devices usually run software atop "bare metal", execute instructions in place (physically from program memory), and have no memory management unit (MMU) to support virtual memory.

Our implementation of $\mathcal{P}$rox is based on TI MSP430. This choice is due to public availability of formally verified $\mathcal{R}$A [8] and PoX [9] architectures implemented over OpenMSP430 [15], upon which our work relies. Nevertheless, our design rationale is applicable to other low-end MCU-s of the same class. There are millions of deployed CPS using such low-end MCUs.

2 BUILDING BLOCKS

2.1 Remote Attestation ($\mathcal{R}$A)

As mentioned earlier, $\mathcal{R}$A enables detection of compromise, e.g., malware infections. It allows $\mathcal{V}$rf to remotely measure the software state of an untrusted remote $\mathcal{P}$rv. As shown in Figure 1, $\mathcal{R}$A is typically realized as a challenge-response protocol:

(1) $\mathcal{V}$rf sends an attestation request containing a challenge (Chal) to $\mathcal{P}$rv. This request might also contain a token derived from a secret that allows $\mathcal{P}$rv to authenticate $\mathcal{V}$rf.
(2) $\mathcal{P}$rv receives the attestation request and computes an *authenticated integrity check* over a pre-defined memory region (e.g., program memory) and Chal.

[1] The only alternative is to dispatch a human operator to directly (physically) fix (e.g., by reprogramming) each compromised $\mathcal{P}$rv; clearly, this is a less desirable approach.

(3) $\mathcal{P}$rv returns the result to $\mathcal{V}$rf.
(4) $\mathcal{V}$rf receives the result from $\mathcal{P}$rv, and checks whether it corresponds to a valid memory state.

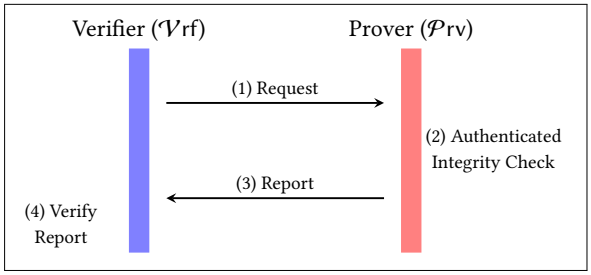


Figure 1: Remote Attestation ($\mathcal{R}$A) Interaction

The *authenticated integrity check* is usually realized as a Message Authentication Code (MAC) or a digital signature over $\mathcal{P}$rv's attested memory region. However, these primitives require $\mathcal{P}$rv to have a unique secret key ($\mathcal{K}$) either shared with $\mathcal{V}$rf (MAC-s), or for which $\mathcal{V}$rf knows the public key (signatures). This $\mathcal{K}$ must reside in secure storage, and **not** be accessible to any (potentially compromised) software running on $\mathcal{P}$rv, except for the trusted attestation code itself. Since most $\mathcal{R}$A threat models assume a fully compromised software state on $\mathcal{P}$rv, secure storage implies some level of hardware support that does not exist in our target devices.

2.2 Proofs of Execution (PoX)

PoX augments $\mathcal{R}$A capability by proving to $\mathcal{V}$rf that: (1) the expected executable is stored in $\mathcal{P}$rv's program memory, (2) this code was indeed executed, and (3) any claimed outputs were produced by its timely and authentic execution.

The first PoX architecture for low-end MCU-s was recently proposed in APEX [9]. APEX implements a hardware module controlling the value of a 1-bit flag called *EXEC*, that cannot be written by any software. $EXEC = 1$ indicates to $\mathcal{V}$rf that attested code *must* have executed successfully, between the time when the challenge was sent from $\mathcal{V}$rf (recall the $\mathcal{R}$A protocol from Section 2.1) and the time when the $\mathcal{R}$A measurement occurs (via authenticated integrity-ensuring function). Similarly, when it receives an attestation reply with $EXEC = 0$, $\mathcal{V}$rf concludes that execution of said code did not occur, or that execution (or its output) was tampered with. In APEX, the $\mathcal{R}$A measurement covers: **(i)** the *EXEC* flag itself; **(ii)** the region where this execution's output is saved (called Output Region = *OR*); and **(iii)** the executable itself (stored in Executable Region =*ER*). Security of the underlying $\mathcal{R}$A architecture guarantees that the contents of these memory regions cannot be forged to reflect something different from their values at time of attestation computation. In turn, APEX considers a code to execute properly (and sets $EXEC = 1$) if and only if:

(1) Execution is atomic (i.e., uninterrupted), from the executable's first instruction (legal entry point ER_{min}), to its last instruction (legal exit point ER_{max});
(2) Neither the executable (in *ER*), nor its produced outputs (in *OR*) are modified in between the execution and subsequent $\mathcal{R}$A computation;
(3) During execution, data-memory (including *OR*) cannot be modified, by means other than the executable itself, e.g., no modifications by other software or Direct Memory Access (DMA) controllers.

These conditions mean that $EXEC = 1$ assures $\mathcal{V}$rf that memory contents of *ER* and *OR* are consistent between *ER*'s execution and respective attestation, and that execution itself is untampered, e.g., via interruptions, or modification of intermediate results in data memory. *ER* and *OR* locations and sizes are configurable, allowing for PoX of arbitrary code and output sizes. APEX's properties are proven sufficient to imply a security definition (stated using the cryptographic security game framework [20]) for unforgeable proofs of execution. Due to space constraints, we do not overview APEX proofs and refer to [9] for details. As mentioned before, later in this paper, we use APEX's PoX functionality to construct a Delegated Attestation ($\mathcal{D}$A) scheme.

3 NETWORK & SYSTEM MODEL

Figure 2 depicts the expected scenario: a network setting with four types of devices: a verifier ($\mathcal{V}$rf), a gateway ($\mathcal{G}$w), an attestation proxy ($\mathcal{P}$rox), and a set of n provers $\mathcal{P}\text{rv}_i$ where $i \in [1, n]$.

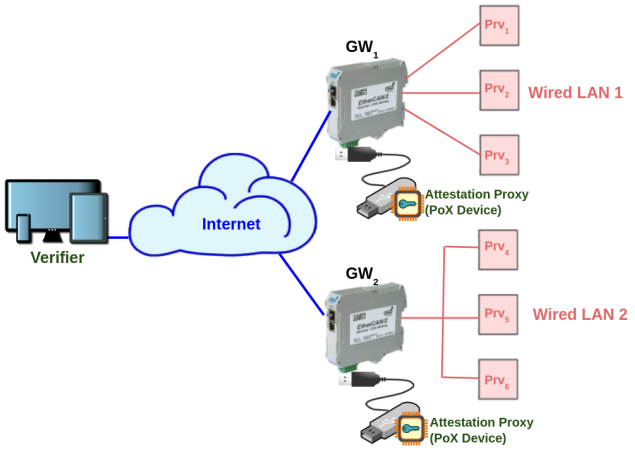


Figure 2: Delegated Attestation Network Setting

3.1 Network Model

Per Figure 2, we assume a set of n $\mathcal{P}$rv-s on a Local Area Network (LAN). A LAN gateway ($\mathcal{G}$w) serves as the entry-point to all LAN-attached devices. A remote $\mathcal{V}$rf communicates with $\mathcal{G}$w over an unreliable jitter-prone network, e.g., the Internet. We refer to the $\mathcal{V}$rf-$\mathcal{G}$w connection, as **Remote Connection (RC)**.

Inside a LAN, $\mathcal{G}$w connects directly to a set of $\mathcal{P}$rv-s: $\mathcal{P}\text{rv}_1$, ..., $\mathcal{P}\text{rv}_n$. We assume a wired connection between $\mathcal{G}$w and all $\mathcal{P}$rv-s, where the communication delay[2] is predictable and constant, i.e., the jitter is negligible. We refer to this connection as **Local Connection (LC)**.

[2]We use communication delay to refer to the sum of all sources of delay in network communication, including transmission, propagation, and in-network processing of packets, when applicable.

We do not assume reliable communication, since messages might be accidentally or maliciously delayed or discarded. Instead, we only assume the lower bound on the LAN communication delay under benign settings. In a nutshell, our LC assumptions are consistent with those in prior software attestation techniques [16]. While an adversary ($\mathcal{A}$dv) present in LC can discard or delay messages, it can never accelerate the delivery of messages due to the physical properties of the communication medium. Adversarial capabilities are discussed further in Section 3.3.

Examples of LC topologies that correspond to our targeted settings include:

- Wired Star Topology – LC-Star (Figure 2, top-right): $\mathcal{G}$w communicates directly (1-hop, constant delay) to each $\mathcal{P}\text{rv}_i$ through an individual dedicated physical circuit.
- Bus Topology – LC-Bus (Figure 2, bottom-right): $\mathcal{G}$w communicates directly to all $\mathcal{P}\text{rv}_i$ using a shared bus (e.g., CAN bus in vehicular systems). While the bus is shared, communication is still done in 1-hop with constant delay, in a broadcast fashion.

Finally, $\mathcal{G}$w is directly connected to $\mathcal{P}$rox, for instance via serial interface, with the same communication assumptions as LC. We note that, depending on the specifics of $\mathcal{D}$A implementation, $\mathcal{P}$rox might be a logical entity, e.g., a PoX-capable TEE residing in $\mathcal{G}$w, rather than a physical standalone device. Since the former requires pre-existing hardware support on $\mathcal{G}$w, we focus on the latter, where $\mathcal{P}$rox is a standalone device directly plugged into a commodity $\mathcal{G}$w. Nonetheless, the concepts discussed in the rest of this paper generalize to both settings.

3.2 Device Types

The following device types are assumed in $\mathcal{D}$A setting:

- **Provers ($\mathcal{P}\text{rv}_1$, ..., $\mathcal{P}\text{rv}_n$)** – purpose-specific low-end embedded devices (e.g., sensors and actuators), responsible for performing safety critical tasks as a part of a bigger CPS (e.g. a vehicle). Provers are locally connected to $\mathcal{G}$w inside LC.
- **Verifier ($\mathcal{V}$rf)** – a trusted remote device that wants to detect whether one or more $\mathcal{P}\text{rv}_i$ is/are compromised. $\mathcal{V}$rf is connected to $\mathcal{G}$w through an inter-networking infrastructure (e.g., over the Internet) in RC.
- **Gateway ($\mathcal{G}$w)** – a (potentially compromised) network device that communicates to both: $\mathcal{V}$rf through RC, and $\mathcal{P}\text{rv}_1$, ..., $\mathcal{P}\text{rv}_n$ through LC.
- **Attestation Proxy ($\mathcal{P}$rox)** – the only entity with secure hardware support. In particular, $\mathcal{P}$rox has a secure PoX facility (as described in Section 2). All software state in $\mathcal{P}$rox, might also be compromised. However, we assume that the minimal root of trust implementing the PoX functionality cannot be compromised.

3.3 Adversarial Model

We consider a strong adversary $\mathcal{A}$dv potentially present in both in RC and LC, as well as inside devices in the form of malware. Presence in the network allows $\mathcal{A}$dv to eavesdrop on traffic on any link (in LC and RC), thus learning transmitted messages (as well as respective metadata, i.e., timing, sizes, sources and destinations). In addition, we consider that $\mathcal{A}$dv can control the entire software state, code, and data of $\mathcal{G}$w, $\mathcal{P}$rox, and any number of $\mathcal{P}$rv-s. $\mathcal{A}$dv can modify any writable memory and read any memory that is not explicitly protected by hardware access control rules. It can also relocate malware from one memory segment to another, or between devices in order to hide it from being detected. It is noteworthy that the ability to compromise existing devices (including $\mathcal{G}$w) means that $\mathcal{A}$dv can forge, delay, and discard network messages at will. Equivalently, $\mathcal{A}$dv can compromise routers in between $\mathcal{V}$rf and $\mathcal{G}$w, positioning itself as a man-in-the-middle in RC.

Physical and hardware-focused attacks are out of scope in this paper. Specifically, we assume that $\mathcal{A}$dv cannot induce hardware faults or violate hardware security properties enforced by the underlying architecture, including the PoX capability of $\mathcal{P}$rox. (Note that the underlying PoX architecture – APEX – is secure under full software compromise of its host [9]). We assume load-time integrity of $\mathcal{P}$rv-s, i.e., their boot ROM is not corrupted. Since LC is a wired LAN (every $\mathcal{P}\text{rv}_i$ is physically attached to $\mathcal{G}$w), even though $\mathcal{A}$dv can compromise any number of $\mathcal{P}$rv-s, we assume that it cannot deploy its own additional $\mathcal{P}$rv-s or modify existing connections in LC, since doing so requires physical access to the LAN and to $\mathcal{G}$w. Protection against such attacks is an orthogonal problem. See [27] for an overview of countermeasures.

3.4 Network & Device Assumptions

Our approach relies on the following assumptions, backed up by empirical evidence.

[A1] Constant Communication Time (in LC): Communication delays in LC are measurable, predictable and deterministic. $\mathcal{V}$rf has *a priori* knowledge of the expected communication delay. Variance in communication delays (i.e., jitter) is negligible.

[A2] Algorithmic & Code Optimality: The implementation of the integrity-ensuring function (i.e., a CBC- or hash-based MAC) used to attest $\mathcal{P}$rv-s is time-optimal.

[A3] $\mathcal{P}$rv Processing Power: The upper bound on $\mathcal{P}$rv processing rate (e.g., in instruction cycles per second) is constant and known to $\mathcal{V}$rf.

[A4] $\mathcal{P}$rv Program/Code Attestation & DEP: Attestation is performed on $\mathcal{P}$rv's program memory (i.e., code). Since data memory mutates constantly, data attestation is not feasible. Instead, mechanisms for Data Execution Prevention (DEP), which are often present in MCUs, can be used to assure that no malware located in data memory can execute, obviating the need to attest data.

Assumptions **A1, A2, and A3** are used by $\mathcal{V}$rf to determine the maximum tolerated delay for a message exchange between $\mathcal{P}$rox and $\mathcal{P}$rv. See Section 4.

Remark. *Assumption **A1** refers to the local wired communication (e.g., CAN-bus, PLC) between $\mathcal{P}$rox and $\mathcal{P}$rv. No assumptions are made about the wide-area communication (e.g., Internet) between $\mathcal{P}$rox and $\mathcal{V}$rf. Similarly, **A2** and **A3** refer to the code and processing power of $\mathcal{P}$rv-s. No such assumptions are required with respect to $\mathcal{P}$rox.*

4 DELEGATED ATTESTATION ($\mathcal{D}$A)

We start by overviewing a general $\mathcal{D}$A approach, where $\mathcal{P}$rv-s run a black-box software-based attestation algorithm, assuming that such software-based attestation method is secure with respect

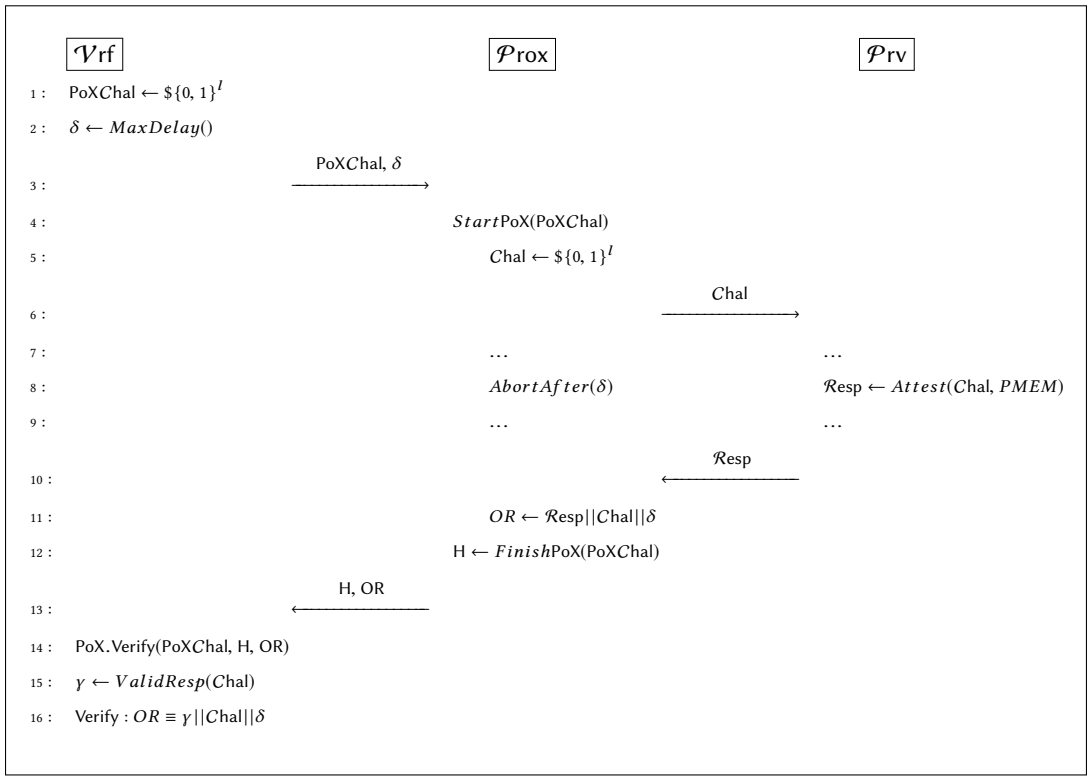


Figure 3: Delegated Attestation Protocol.

to the typical required properties discussed in Section 3.4. Then, Sections 4.2 and 4.3 present two concrete $\mathcal{P}$rv implementations, based on: (i) traditional software attestation that requires time-optimal implementation, and (ii) a keyless version of SIMPLE [2], which does not require time optimality. For clarity's sake, we first consider the case with only one $\mathcal{P}$rv. Later, Section 4.4 discusses how to extend it to multiple (and potentially heterogeneous) $\mathcal{P}$rv-s.

4.1 Protocol Overview

Figure 3 specifies the $\mathcal{D}$A protocol. It relies on $\mathcal{P}$rox to generate a PoX for the challenge-response protocol used in the underlying software attestation scheme. In turn, PoX properties (see Section 2.2) guarantee to $\mathcal{V}$rf that any deviation from the protocol (e.g., if $\mathcal{P}$rox itself is compromised) is detectable.

The protocol starts when $\mathcal{V}$rf generates a PoX challenge (PoX$\mathcal{C}$hal, at line 1) and sets the maximum tolerated delay (δ, at line 2), which is determined according to the specific setting of $\mathcal{P}$rv's is deployment, considering the LC communication delay and $\mathcal{P}$rv's time to compute attestation, as discussed in Section 3.4. $\mathcal{V}$rf forwards these two values to $\mathcal{P}$rox, which starts a PoX instance using PoX$\mathcal{C}$hal. In order to be accepted by $\mathcal{V}$rf, this PoX must contain the correct generation of a fresh challenge ($\mathcal{C}$hal), $\mathcal{P}$rox interaction with $\mathcal{P}$rv in the challenge-response protocol according to the underlying software attestation scheme, and the collection of the response. Most importantly, the code executed in this PoX, only accepts a $\mathcal{P}$rv's response received within the threshold δ after sending $\mathcal{C}$hal to $\mathcal{P}$rv. If this condition is met, $\mathcal{P}$rox sets PoX output (OR) to include the value of $\mathcal{C}$hal, $\mathcal{R}$esp, and δ. These steps correspond to lines 4-11 of Figure 3.

Upon completion, $\mathcal{P}$rox produces a proof of execution of these steps (as specified and expected by $\mathcal{V}$rf), thereby tying the produced outputs in OR (i.e., $\mathcal{R}$esp||$\mathcal{C}$hal||δ) to this execution via proof token H (line 12). All OR values are sent back to $\mathcal{V}$rf along with H (line 13). Next, $\mathcal{V}$rf checks whether token H is a valid PoX for the specified behavior of $\mathcal{P}$rox, and whether OR values were indeed produced by this execution. If the proof holds, $\mathcal{V}$rf checks if $\mathcal{R}$esp is a valid checksum computed using $\mathcal{C}$hal and $\mathcal{P}$rv's expected software state (lines 15 and 16).

Since $\mathcal{C}$hal and δ are tied to the PoX, by verifying H, $\mathcal{V}$rf can be assured that those values were indeed utilized properly in the interaction between $\mathcal{P}$rox and $\mathcal{V}$rf. Hence, $\mathcal{V}$rf securely delegates these verification steps to $\mathcal{P}$rox. (Recall that $\mathcal{P}$rox is directly connected to LC and has a constant one-hop delay to $\mathcal{P}$rv.)

We use this technique in two $\mathcal{D}$A variants as we discuss next.

4.2 $\mathcal{D}$A with Time-Optimal Software Attestation

This instance of $\mathcal{P}$rv's implementation deploys a time-optimal integrity-ensuring function (e.g., a keyed-hash) on $\mathcal{P}$rv. Upon receiving $\mathcal{C}$hal, this checksum must be computed over $\mathcal{P}$rv's entire program memory. Under the following 3 assumptions:

(1) Constant communication delay in LC (**[A1]** in Section 3.4),

(2) Time optimality ([**A2**] in Section 3.4), and
(3) Known computational power of $\mathcal{P}$rv ([**A3**] in Section 3.4).

$\mathcal{V}$rf can determine the exact response time and distrust any $\mathcal{R}$esp received thereafter. This threshold (δ in Figure 3) is given by the sum of the lower bound on LC's communication delay and the processing time for checksum computation on $\mathcal{P}$rv. Therefore, potential malware in $\mathcal{P}$rv has no time to attempt any action to avoid being detected (e.g., interrupt attestation to copy itself around memory to escape coverage by the attestation's checksum), as any such action would necessarily take more time than computing the time-optimal checksum.

While most traditional software attestation techniques [5, 7, 14, 21, 23, 30–33, 36] rely on ad-hoc optimality assumptions (e.g., the checksum's algorithm and code are carefully hand-optimized), the one recently proposed in [16] is the first that is provably time-optimal. Unfortunately, it is not appropriate for embedded devices due to its high computational complexity and its memory access model. Future work should look into similar techniques targeting more realistic embedded devices.

Instead, this part of our $\mathcal{DA}$ prototype is based on SWATT [30]. SWATT implements a checksum that traverses memory in a pseudo-random fashion, where the order is derived from $\mathcal{C}$hal. This unpredictability hardens the scheme against self-relocating malware, since malware does not have *a priori* knowledge about which particular memory locations will be covered next. At each attestation instance, $\mathcal{V}$rf uses the value of $\mathcal{C}$hal to reconstruct the memory traversal order and verify the checksum accordingly.

4.3 $\mathcal{DA}$ with a Non-time-optimal Software Attestation

The main drawback of the above is the time-optimality requirement. Even the scheme in [16] only proves optimality at the algorithmic level. This proof does not carry on to the implementation level, which is important, considering different compilation tool-chains of different processors and MCUs. To get rid of the time-optimality assumption we propose another construction based on a recent $\mathcal{RA}$ scheme, called SIMPLE [2].

SIMPLE Overview. Figure 4 illustrates SIMPLE's functionality. It emulates hybrid $\mathcal{RA}$ schemes, implementing a trust anchor entirely in software. SIMPLE relies on a software memory isolation primitive – the Security MicroVisor (SμV) [1] – to logically isolate a segment of the MCU's program memory for storing the attestation trusted code, protecting this region against reads and writes coming from any untrusted software in the MCU. Execution of the trusted code can only be triggered through legal entry points, preventing arbitrary execution of chunks of this code, and associated code re-use attacks. To guarantee such safety rules, the SIMPLE software component must be installed on $\mathcal{P}$rv at deployment time by a trusted party. Thereafter, untrusted software can be installed and updated "over the air", passing through a *Loader/Verifier* function that is part of SμV, to check its adherence to the safety rules that verify each instruction's binary. If any instruction is found to be unsafe, the application is rejected. An application instruction is considered unsafe if it tries to illegally access the isolated memory segment, i.e., read/write from/to it.

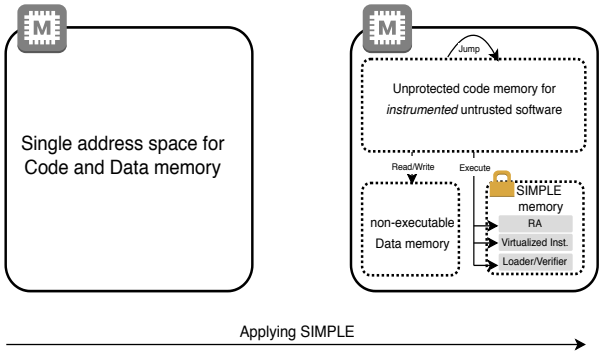


Figure 4: MCU memory layout. Left: without SIMPLE. Right: with SIMPLE access control. Dashed boxes represent logical isolation (enforced by the underlying SμV primitive).

While instructions with static addressing mode (for which the target memory address is known) can be checked at load-time, target addresses of dynamic instructions are only determined at runtime. Therefore, SIMPLE depends on a SμV-enabled toolchain to replace these instructions with safe virtualized equivalents, which can be accessed via a call to a subroutine stored in the isolated memory segment. Whenever the subroutine is called at runtime, it validates the corresponding instruction by checking its target address: if it is safe, normal execution continues; otherwise, a soft MCU reset is triggered. Notably, SIMPLE does not trust the modified tool-chain, since load-time verification of software on $\mathcal{P}$rv itself prevents $\mathcal{A}$dv from using her own tool-chains or writing hand-crafted assembly instructions.

The original design of SIMPLE requires a symmetric key that has to be physically installed, along with the the attestation code, at each $\mathcal{P}$rv, during deployment (usually product manufacture/assemble time). This requirement introduces key management and provisioning challenges. First, enforcing confidentiality, at scale, during deployment, is difficult. Second, $\mathcal{V}$rf must securely store and manage symmetric keys for a large number of $\mathcal{P}$rv-s (one key per $\mathcal{P}$rv), thus being a single point of failure. Finally, if key updates are ever needed (e.g., if one or more keys are accidentally leaked by $\mathcal{V}$rf), it is not clear how to do so, since in various use-cases (e.g., sensors and actuators inside vehicles), $\mathcal{P}$rv-s are hard to access physically at post-manufacture stages.

To address these issues and alleviate the optimality assumption ([**A2**]) of traditional software attestation schemes (Section 4.2), we propose a key-less version of SIMPLE.

Key-Less SIMPLE (`KL-SIMPLE`). We modify SIMPLE in two ways. As secrets no longer exist in the protected memory region, no runtime read protection is required. Thus, we eliminate all related safety-checks from SμV. To eliminate the strict optimality dependence of unconditional schemes (e.g., our SWATT-based approach), at deployment time, `KL-SIMPLE` also programs a *secure timer-based interruption*. Similar to runtime protection checks performed by the original SIMPLE, `KL-SIMPLE` will enforce that this interrupt can not be disabled or misconfigured by untrusted applications (i.e., by protecting writes to the interrupt handler implementation,

the interrupt vectors, and the interrupt configuration registers corresponding to this particular timer). Importantly, *only* KL-SIMPLE *trusted software (the attestation software) can disable this interrupt.*

The purpose of this guaranteed interruption is straight-forward: it periodically delays the execution of untrusted software using a busy-wait. Specifically, we say that this guaranteed interruption is triggered on fixed periods of T causing a time delay of t.

To see how this strategy alleviates optimality requirements, let $\mathsf{T}_{\mathcal{A}\mathsf{TT}}$ denote the time consumed by the checksum implemented by KL-SIMPLE attestation function. The approach works as follows:

- The period between guaranteed interrupts is set to $T = \mathsf{T}_{\mathcal{A}\mathsf{TT}} - t$. Where t is the delay caused by each such interrupt.
- When KL-SIMPLE attestation is called, it uses its privileged control to disable this interrupt during its own attestation, enabling it back upon completion. Hence, KL-SIMPLE attestation is the only software in the MCU that is not delayed by this interrupt.

With this approach, any $\mathcal{A}$dv's optimized checksum that is potentially faster than KL-SIMPLE's checksum by less than t is still detectable. This approach is similar to (though not exactly the same as) a *secure software-based clock divider*. As such, a runtime overhead of $\frac{t}{\mathsf{T}_{\mathcal{A}\mathsf{TT}}-t}$ is incurred to all untrusted applications, which is the price of not requiring strict time optimality. This represents an interesting trade-off, whereas applications that require stronger runtime security may choose to implement $\mathcal{D}$A with KL-SIMPLE, while those with strict runtime requirements still have the traditional software attestation option. For instance, if an MCU might has a default clock speed of 16MHz and its applications only require 8MHz, KL-SIMPLE can be used to implement key-less attestation that is secure even if $\mathcal{A}$dv manages to implement attestation forgery that is up to 2× faster than KL-SIMPLE checksum, which itself can be implemented using highly optimized software attestation checksums.

Note that, despite the introduced delay periods, KL-SIMPLE is still friendly to real-time task scheduling, as the delays and the intervals between them are deterministic. Clearly, with a delay of $t = 1s$, this implementation would not guarantee an application response time of $0.5s$. However, delays that are too long for specific applications requiring faster responses can be amended by breaking them down into multiple shorter delays within the period T.

4.4 $\mathcal{D}$A with Multiple Provers

Both proposed schemes can be extended to support scalable multi-$\mathcal{P}$rv attestation.

In settings where multiple $\mathcal{P}$rv-s are on the same bus (LC), our scheme can benefit from direct one-hop communication between $\mathcal{P}$rv-s and $\mathcal{G}$w/$\mathcal{P}$rox by broadcasting $\mathcal{C}$hal to all $\mathcal{P}$rv-s simultaneously. $\mathcal{P}$rv-s must be configured to report their responses at different times taking into account two conditions: (i) no more than one $\mathcal{P}$rv reports at the same time to avoid network collisions, and (ii) the gap between the first reporting $\mathcal{P}$rv and the last one is not enough for the former to be compromised and assist the latter in forging an attestation response. Both conditions can be met by incorporating a per-$\mathcal{P}$rv proof-of-work based delay function. One example of such proof-of-work consists of requiring the attestation response to be re-hashed several times. In settings where some $\mathcal{P}$rv-s have faster CPUs than others, longer proofs-of-work can be assigned to faster devices to assure that all $\mathcal{P}$rv-s cannot assist each other in forging attestation responses, as all computational resources in the network are simultaneously required in order for all $\mathcal{P}$rv-s to meet the attestation deadline. That way, sets of compromised $\mathcal{P}$rv-s can not assist a slower $\mathcal{P}$rv in forging attestation responses. Importantly, given that the $\mathcal{G}$w is an untrusted device, our design considers keeping it busy (by a proof-of-work computation function) during the waiting period for attestation reports to prevent it from collusion with other compromised $\mathcal{P}$rv-s.

This multi-$\mathcal{P}$rv scheme also assigns a unique ID for each $\mathcal{P}$rv, to be included in the hash-based checksum's input. The IDs allow $\mathcal{V}$rf to differentiate between responses from multiple provers, that are installed with the same software, even though their checksums are computed using the same challenge. Unless $\mathcal{V}$rf receives one correct checksum per ID, it will distrust the overall result of multi-prover attestation.

5 IMPLEMENTATION

We prototyped $\mathcal{D}$A and experimented with it over a CAN-enabled testbed consisting of 16 $\mathcal{P}$rv-s and one $\mathcal{G}$w/$\mathcal{P}$rox, as shown in Figure 5.

5.1 Proxy ($\mathcal{P}$rox)

$\mathcal{P}$rox is implemented atop the publicly available verified architecture for proofs-of-execution (APEX) [9]. APEX yields the proof-of-execution required by $\mathcal{D}$A as specified in Figure 3, in particular, to prove execution of lines 4–12 in Figure 3. This code is implemented as an embedded application on the TI MSP430 microcontroller, which is connected directly to the gateway device in the local network (LC), where $\mathcal{P}$rv-s are located. The MSP430 hardware design is synthesized along with the APEX architecture using the commodity FPGA Basys3. Per protocol specifications, software in $\mathcal{P}$rox has three main functions: (1) random number generation (line 5); (2) challenge-response communication (lines 6 and 10); and (3) delay function (line 8). We elaborate on them below.

(1) is implemented using the technique described in [34]. This technique is proposed and recommended by Texas Instruments. It is primarily based on the time difference between edge transitions

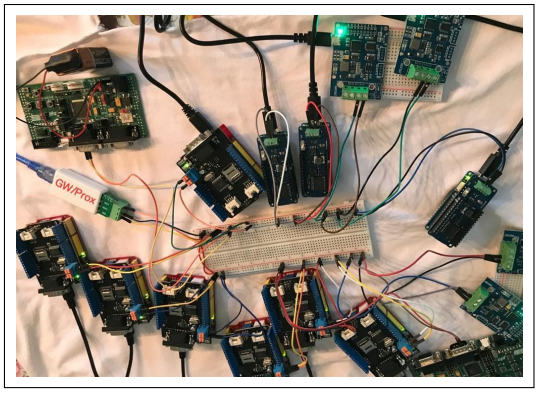


Figure 5: A snapshot of our prototype.

Table 1: Specifications of $\mathcal{P}$rv-s hardware

MCU type	RAM size (kB)	Flash size (kB)	Clock speed (MHz)	Quantity	Platform
AVR ATmega 328P	2	32	16	7	Arduino UNO
AVR ATmega32U4	2.5	32	8	4	Arduino Leonardo
SAMD21 Cortex-M0+	32	256	48	3	Arduino MKR1000
AVR AT90CAN128	4	128	16	2	DVK90CAN1

of two independent clock sources (very-low frequency oscillator – VLO, and digitally controlled oscillator – DCO), which can be exploited to generate a stream of random bits. In one VLO clock cycle, there is always approximately the same number of DCO pulses. However, because the two clock sources are independent, whether this number of pulses is even or odd is not predictable. More importantly, this number is not predictable even if the previous result is known. This method is evaluated in [13], where NIST statistical tests show that it indeed generates random numbers with sufficient entropy for cryptographic applications. The time required by this step is linear in the number of bits in $\mathcal{C}$hal. We use $|\mathcal{C}\text{hal}| = 128$ bits, which incurs the generation time of $\approx$ 11ms. This time does not affect the attestation threshold δ, since $\mathcal{C}$hal's generation is done before sending the request to $\mathcal{P}$rv (-s).

$\mathcal{P}$rox also forwards the randomly generated challenge to $\mathcal{P}$rv (-s). This in implemented as a simple communication function using the MSP430's UART interface. In our case, UART is connected to external communication hardware – CAN bus module.

The delay function determines the maximum time threshold that $\mathcal{P}$rox should wait for a response from $\mathcal{P}$rv (-s). It is implemented as a naïve busy-wait function, i.e., a simple counter loop. Once the delay elapses, $\mathcal{P}$rox checks the UART receive interface for a response from $\mathcal{P}$rv. If not present, $\mathcal{P}$rox interprets the result as an attestation failure and reports back to $\mathcal{V}$rf accordingly. Responses received thereafter are ignored.

5.2 Provers ($\mathcal{P}$rv-s)

We implemented the integrity-ensuring function in both $\mathcal{P}$rv approaches using SHA-256. For the first approach (Section 4.2), we use a verified implementation from the HACL* library [37] and assume its time-optimality. In the second approach (Section 4.3), we use a compact implementation[3] to minimize the memory footprint on $\mathcal{P}$rv. We considered 4 types of low-end MCUs that are widely used in various sensors and actuators. Table 1 shows the hardware specification of each. All $\mathcal{P}$rv-s are integrated with a CAN transceiver, allowing them to send and receive CAN packets at speed of 125kbps. Our testbed mimics the network settings found in contemporary automobiles and some automation systems.

6 EXPERIMENTAL EVALUATION

We now report on experimental results.

6.1 DA with Single $\mathcal{P}$rv

Time of computing $\mathcal{R}$esp and memory footprint. Considering specifications in Table 1, Figure 6 shows the time to compute $\mathcal{R}$esp on each of $\mathcal{P}$rv type using both $\mathcal{D}$A variations. Results of SWATT-based variation (in blue) are reported according to a pseudorandom

[3]From: https://github.com/cantora/avr-crypto-lib

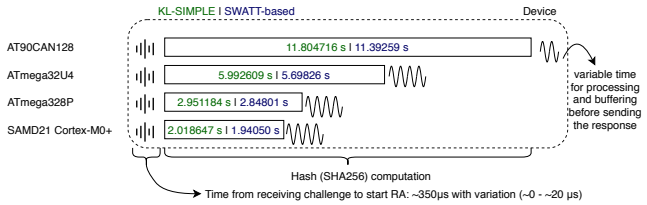


Figure 6: The required time to compute $\mathcal{R}$esp along with other pre- and post-computation time for each of the considered platforms.

traversal of memory per page by $O(n)$, where n is the number of memory pages in the corresponding $\mathcal{P}$rv. In practice, to access all memory pages with high probability, increasing the number of accesses to $O(n \ln n)$ is recommended, based on the Coupon Collector's Problem [10].

The size of $\mathcal{R}$A function was under 5kB in the first approach, whereas, the entire trusted code was less than 3.3kB in the second approach, utilizing under 15.6% and 10.3%, respectively, of the smallest-size Flash; see Table 1.

Runtime overhead. The SWATT-based approach does not incur any runtime overhead for the untrusted application, while KL-SIMPLE incurs $\approx (0.01\% - 6\%) + \frac{t}{T_{\mathcal{ATT}} - t}$. The former is an application-dependent value, due to the instrumentation required by the underlying SμV primitive as reported in [1]. The latter is because of activating a timer-based interrupt as discussed in Section 4.3, where $T_{\mathcal{ATT}}$- t is the active period of the untrusted software, and t is the specified delay.

Response delay. By running $\mathcal{D}$A with individual benign $\mathcal{P}$rv-s (shown in Table 1) using both $\mathcal{D}$A variants, 10 times each, a maximum of 25μs has been recorded for the vast majority of $\mathcal{P}$rv-s as a time variation when delivering $\mathcal{R}$esp-s. This value is increased, up to 100μs, with some $\mathcal{P}$rv-s as explained later. In the worst-case scenario, this variation is not sufficient for malware to hide its presence or forge a correct $\mathcal{R}$esp, as discussed in Section 6.4.

6.2 DA with Multiple $\mathcal{P}$rv

Please note that in the benign settings, both $\mathcal{P}$rv approaches considered behave similarly w.r.t. the variations of delivering responses. Therefore, due to the limited space, in this section, we report only the results considering KL-SIMPLE as $\mathcal{P}$rv's instance.

Incremental performance analysis. To evaluate the performance in various network settings, we run $\mathcal{D}$A with a different number of $\mathcal{P}$rv-s, 10 times each, separated by a 15-minute time window. We then computed the end-to-end[4] time variation w.r.t. the time reported in the first trial. In our experiment, $\mathcal{P}$rv-s are configured to iteratively perform, as a normal task, random computations for 1 second and then sleep for 3 seconds. The experiment is performed in benign settings without considering any delay function, leveraging the CSMA/RA feature of the underlying CAN transceivers that allows for re-transmitting responses, in case of collisions, until they are received by the designated party. Figure 7

[4]End-to-end means the time period from sending the last bit of $\mathcal{C}$hal until receiving the last bit of the last responding $\mathcal{P}$rv.

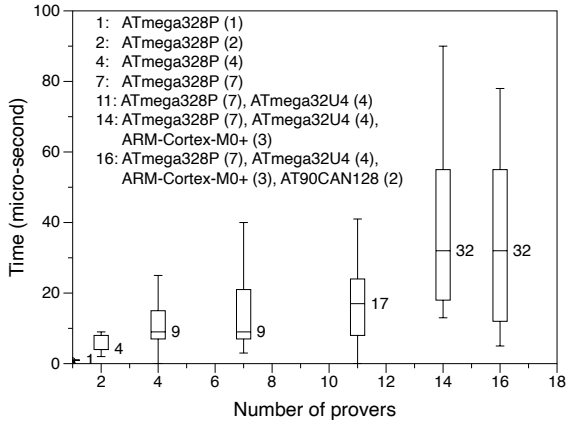


Figure 7: The end-to-end time variation (μs) when running $\mathcal{D}$A with a different number of $\mathcal{P}$rv-s, 10 times each.

shows the time variations in microseconds when considering networks with 1, 2, 4, 7, 11, 14, or 16 $\mathcal{P}$rv-s. The types of underlying MCUs of selected $\mathcal{P}$rv-s in each experiment are illustrated in the figure itself. The results show that, in the worst-case scenario, the end-to-end time variation of $\mathcal{D}$A does not exceed 90 μs when considering a network of 16 heterogenous $\mathcal{P}$rv-s.

Individual responses. In order to analyze the time variation of individual responses, we run $\mathcal{D}$A 10 times on the entire 16-$\mathcal{P}$rv testbed. To limit the number of re-transmission cases, we configured homogeneous $\mathcal{P}$rv-s to run a simple proof-of-work delay function (iteratively hashing the output result for a number of times) with different delay periods, ranging from 5 to 15 ms, taking into account that sending $\mathcal{R}$esp over CAN requires $\approx$ 0.9 ms. Figure 8 visualizes the time variation of responses for each considered $\mathcal{P}$rv. The figure is divided into four parts to maintain the readability of it. It shows that, for the majority of considered $\mathcal{P}$rv-s, the maximum variation is not more than 23 μs, except for a group of $\mathcal{P}$rv-s (3 $\mathcal{P}$rv-s, shown in Figure 8c), which can reach until 95.2 μs. The careful experimental analysis demonstrated that this is an exceptional case as these $\mathcal{P}$rv-s are accommodated with inaccurate/unstable crystal oscillator.

6.3 $\mathcal{P}$rox Evaluation

As discussed in Section 5, $\mathcal{P}$rox performs three main functions: random number generation, challenge-response communication, and delay. The network delay was discussed in the previous sections. Meanwhile, the time in the delay function is determined by the protocol itself (δ), depending on the particular network setting. Hence, we now consider the time required by the random number generation, as described in Section 5. Figure 9 presents this time as a function of the random number bit-size in our implementation. In terms of code size, $\mathcal{P}$rox behavior was implemented in `C` in 176 LoC-s.

6.4 Security Evaluation

$\mathcal{D}$A with `KL-SIMPLE` approach. Malware can only follow two directions in order to evade detection. The first is to forge $\mathcal{R}$esp using a faster hash function and send it within the expected time window. The chance that malware can benefit from this option is very low due to the delay caused by the timer-based interrupt, giving the correct $\mathcal{R}$A call an edge of t seconds, where t is an application-dependent value. Note that, the higher t, the higher the runtime overhead. However, this means stronger security guarantees.

The second direction is for malware to erase itself and restore $\mathcal{P}$rv memory to its original content before redirecting $\mathcal{C}$hal to the correct $\mathcal{R}$A. Table 2 illustrates the time required to erase or write a single 2-byte word in Flash of all considered MCUs/$\mathcal{P}$rv types. These experimentally obtained numbers are in line with MCUs' specifications in their respective datasheets. The smallest action malware can take is to erase itself from Flash before giving control to $\mathcal{R}$A. We assume that only a single word is being erased, even though it would be much more in practice.[5] Results in Table 2 show that the actual time is much longer than the maximum variation detected for each $\mathcal{P}$rv type.

$\mathcal{D}$A with a time-optimal SWATT-based approach. Assuming a time optimal integrity-ensuring function, malware can not use a faster function to compute $\mathcal{R}$esp. Furthermore, erasing or relocating itself causes at least the same delay as reported in Table 2, thus assuring that malware presence would be detected.

The only choice left for malware is to use the interruptibility feature by inserting a conditional statement to check whether the next memory block to be read is the block where it is located. If so, it would redirect the read to some other memory location, i.e., SRAM, that contains expected contents. Our experiments confirm that malware would definitely lose this race, because of the pseudorandom memory traversal technique.

To clarify this claim, we consider the best case for malware when $\mathcal{P}$rv is based on AVR ATmega328P MCU, which has the smallest Flash memory (32 kB) and the fastest clock (16 MHz). The MCU has 256 memory blocks. To cover all n blocks with high probability, the number of accesses should be at least O($n \ln n$). This means that 1, 408 memory block traversals must be performed. For malware t insert a simple conditional statement (as above) would add at least 6 clock cycles to each round. This results in a total of 8, 448 **extra** clock cycles, prolonging the execution time by 528 μs, which is 26 times greater than the maximum variation detected for this $\mathcal{P}$rv type.

[5] In Flash memory, erasure only occurs per block (memory page). The smallest size of a memory page in our MCUs is 64 words (128 bytes), meaning that erasure of 1 or 64 contiguous words takes the same time.

Table 2: Time to erase/write a single word.

MCU type	Erasing one word (two bytes) (ms)	Writing one word (two bytes) (ms)
AVR ATmega 328P	2.64	5.29
AVR ATmega32U4	5.27	10.54
SAMD21 Cortex-M0+*	0.84	1.15
AVR AT90CAN128	2.64	5.29

*This is considered a half-word for this MCU.

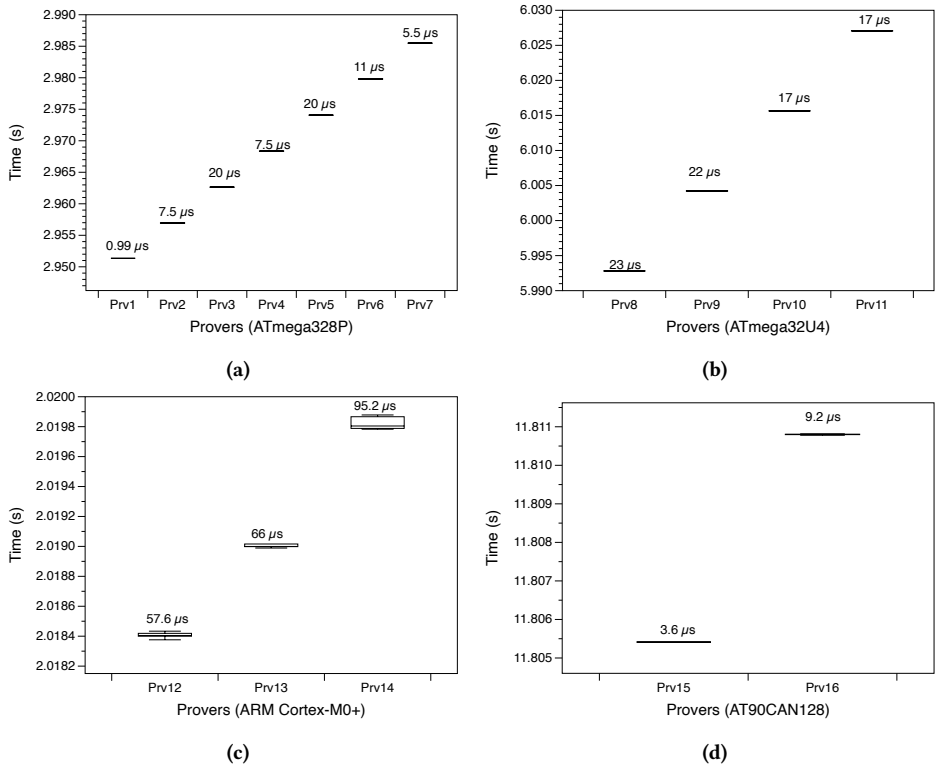


Figure 8: Variation of individual responses (in μs) when running $\mathcal{D}$A with multiple $\mathcal{P}$rv-s 10 times.

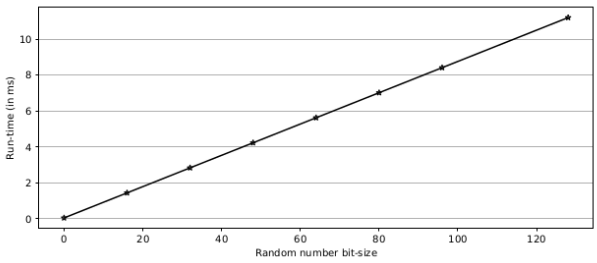


Figure 9: Runtime of $\mathcal{P}$rox random number generation on MSP430 as a function of random number bit-size.

6.5 $\mathcal{D}$A Feasibility

$\mathcal{D}$A's goal is to enable scalable and efficient remote attestation without requiring the hardware of $\mathcal{P}$rv-s to be modified. Introducing a small (portable) device ($\mathcal{P}$rox) [6] to the network is affordable and logistically much easier than modifying or replacing all $\mathcal{P}$rv-s. The $\mathcal{D}$A design obviates the need for secret keys on $\mathcal{P}$rv-s, avoiding key management issues. This is an important feature that makes $\mathcal{D}$A applicable to several domains that currently lack practical solutions. For instance, elevator manufacturers can attest, through their technicians, various MCUs ($\mathcal{P}$rv-s) of deployed elevators using only one portable $\mathcal{P}$rox device that can be plugged in the elevator through a USB (or some other) port [17].

Finally, our $\mathcal{D}$A design is applicable to other types of wired networks. Our choice of the CAN bus setting is motivated by the fact that over one billion CAN network are already deployed in vehicles and other robotic systems [19].

7 CONCLUSION

This paper proposed, implemented, and evaluated a protocol for delegated attestation ($\mathcal{D}$A). It is the first technique for secure *remote* attestation ($\mathcal{R}$A) of commodity CPS devices that lack hardware support for $\mathcal{R}$A. It exploits a recently proposed primitive for proofs-of-execution to enable secure remote verification of the software state of potentially compromised remote embedded devices. $\mathcal{D}$A's practicality and low computational costs were showcased using its implementation over a CAN bus on an experimental testbed.

ACKNOWLEDGMENTS

Research of the first two authors was supported by KU Leuven and imec, a research institute of the Flemish Government, as well as by EU H2020-SU-ICT-03-2018 Project 830929 CyberSec4Europe. Research of the second two authors was supported by funding from Semiconductor Research Corporation (SRC) Contract 2019-TS-2907, NSF Awards SATC-1956393 and CICI-1840197, as well as a subcontract from Perspecta Labs.

[6] In some platforms where $\mathcal{G}$w is lacked, we require introducing $\mathcal{G}$w as well.

REFERENCES

[1] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. 2019. SμV—The Security MicroVisor: A Formally-Verified Software-Based Security Architecture for the Internet of Things. *IEEE Transactions on Dependable and Secure Computing* 16, 5 (2019), 885–901.

[2] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. 2020. SIMPLE: A Remote Attestation Approach for Resource-constrained IoT devices. In *ACM/IEEE ICCPS*. 247–258.

[3] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. 2017. Understanding the mirai botnet. In *USENIX Security*.

[4] Arm Ltd. 2018. Arm TrustZone. https://www.arm.com/products/security-on-arm/trustzone

[5] Frederik Armknecht et al. 2013. A Security Framework for the Analysis and Design of Software Attestation. In *ACM CCS*.

[6] F. Brasser et al. 2015. TyTAN: Tiny Trust Anchor for Tiny Devices. In *DAC*. ACM.

[7] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. 2009. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM conference on Computer and communications security*. 400–409.

[8] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. 2019. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *28th USENIX Security Symposium (USENIX Security 19)*. 1429–1446.

[9] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2020. APEX: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise. In *29th USENIX Security Symposium (USENIX Security 20)*.

[10] Aristides V Doumas and Vassilis G Papanicolaou. 2012. The coupon collector's problem revisited: asymptotics of the variance. *Advances in Applied Probability* 44, 1 (2012), 166–195.

[11] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. 2017. HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In *Proceedings of the 10th ACM Conference on Security and Privacy in wireless and Mobile Networks*. ACM, 99–110.

[12] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. 2012. SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust. In *NDSS*, Vol. 12. 1–15.

[13] Radek Fujdiak, Jiri Misurec, Petr Mlynek, and Ondrej Raso. 2015. Analysis of random number generator from texas instrument in MSP430 x5xx families. In *2015 38th International Conference on Telecommunications and Signal Processing (TSP)*. IEEE, 653–656.

[14] R.W. Gardner et al. 2009. Detecting Code Alteration by Creating a Temporary Memory Bottleneck. *IEEE TIFS* (2009).

[15] Olivier Girard. 2009. openMSP430.

[16] Virgil D Gligor and Shan Leung Maverick Woo. 2019. Establishing Software Root of Trust Unconditionally.. In *NDSS*.

[17] Senad Huseinbegovic, Sead Kreso, and Omer Tanovic. 2009. Design and implementation of the CAN based elevator control system. In *2009 XXII International Symposium on Information, Communication and Automation Technologies*. IEEE, 1–6.

[18] Intel. [n.d.]. Intel Software Guard Extensions (Intel SGX). https://software.intel.com/en-us/sgx

[19] Karl Henrik Johansson, Martin Törngren, and Lars Nielsen. 2005. Vehicle applications of controller area network. In *Handbook of networked and embedded control systems*. Springer, 741–765.

[20] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to modern cryptography*. CRC press.

[21] Rick Kennell et al. 2003. Establishing the Genuinity of Remote Computer Systems. In *USENIX Security*.

[22] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. 2014. TrustLite: A security architecture for tiny embedded devices. In *EuroSys*. ACM.

[23] X. Kovah et al. 2012. New Results for Timing-Based Attestation. In *IEEE S&P '12*.

[24] Jonathan McCune et al. 2008. Flicker: An Execution Infrastructure for TCB Minimization. *SIGOPS Operating Systems Review* (2008).

[25] J Noorman et al. 2013. Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base. In *USENIX*.

[26] Jr. Petroni et al. 2004. Copilot — A Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX Security*.

[27] Srivaths Ravi, Anand Raghunathan, and Srimat Chakradhar. 2004. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design*.

[28] t Sailer et al., Reiner. 2004. Design and Implementation of a TCG-based Integrity Measurement Architecture.. In *SSYM'04*.

[29] Dries Schellekens et al. 2008. Remote attestation on legacy operating systems with trusted platform modules. *Science of Comp. Programming* (2008).

[30] A. Seshadri et al. 2004. SWATT: Software-based Attestation for Embedded Devices. In *IEEE S&P '04*.

[31] A. Seshadri et al. 2005. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. In *ACM SOSP*.

[32] Arvind Seshadri et al. 2006. SCUBA: Secure Code Update By Attestation in Sensor Networks. In *ACM WiSe*.

[33] Arvind Seshadri et al. 2008. SAKE: Software Attestation for Key Establishment in Sensor Networks. In *DCOSS*.

[34] Texas Instruments. 2018. Random Number Generation Using MSP430 MCUs. https://www.ti.com/lit/an/slaa338a/slaa338a.pdf

[35] Trusted Computing Group. 2017. Trusted Platform Module (TPM). http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module/

[36] Yi Yang et al. 2007. Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks. In *SRDS'07*.

[37] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1789–1806.