HEALED: HEaling & Attestation for Low-End Embedded Devices

Ahmad Ibrahim[1](✉), Ahmad-Reza Sadeghi[1], and Gene Tsudik[2]

[1] Technische Universität Darmstadt, Darmstadt, Germany
{ahmad.ibrahim,ahmad.sadeghi}@trust.tu-darmstadt.de
[2] University of California, Irvine, CA, USA
gts@ics.uci.edu

Abstract. We are increasingly surrounded by numerous embedded systems which collect, exchange, and process sensitive and safety-critical information. The Internet of Things (IoT) allows a large number of interconnected devices to be accessed and controlled remotely, across existing network infrastructure. Consequently, a remote attacker can exploit security vulnerabilities and compromise these systems. In this context, remote attestation is a very useful security service that allows to remotely and securely verify the integrity of devices' software state, thus allowing the detection of potential malware on the device. However, current attestation schemes focus on detecting whether a device is infected by malware but not on disinfecting it and restoring its software to a benign state.

In this paper we present HEALED – the first remote attestation scheme for embedded devices that allows both detection of software compromise and disinfection of compromised devices. HEALED uses Merkle Hash Trees (MHTs) for measurement of software state, which allows restoring a device to a benign state in a secure and efficient manner.

1 Introduction

Embedded devices are being increasingly deployed in various settings providing distributed sensing and actuation, and enabling a broad range of applications. This proliferation of computing power into every aspect of our daily lives is referred to as the Internet of Things (IoT). Examples of IoT settings range from small deployments such as smart homes and building automation, to very large installations, e.g., smart factories. Similarly, an embedded or (IoT device) may constitute a low-end smart bulb in a smart home or a sophisticated high-end Cyber-Physical System (CPS) in a smart factory.

Increasing deployment and connectivity combined with the collection of sensitive information and execution of safety-critical (physical) operations has made embedded devices an attractive target for attacks. Prominent examples include: the Stuxnet worm [36], the Mirai botnet [10], the HVAC attack [1] and the Jeep hack [2]. One common feature of such attacks is that they usually involve modifying the software state of target devices. This is referred to as malware infestation.

I. Goldberg and T. Moore (Eds.): FC 2019, LNCS 11598, pp. 627–645, 2019.
https://doi.org/10.1007/978-3-030-32101-7_36

Remote attestation has evolved as a security service for detecting malware infestation on remote devices. It typically involves a standalone (or network of) *prover* device(s) securely reporting its software state to a trusted party denoted by *verifier*. Several attestation protocols have been proposed based on trusted software for securing the measurement and reporting of a prover's software state [14,17,20,32–34], on trusted hardware [19,21,22,27,29,31,35], or on software/hardware co-design [12,13,18]. In the recent years, several collective attestation schemes have been proposed that enable efficient attestation of large networks of devices [5,8,15,16].

While prior remote attestation schemes focus on detection of malware infestation on prover devices, the problem of disinfecting a prover, i.e., restoring its software to a benign state, has been totally overlooked. Prior remote attestation schemes usually focus on malware presence detection and consider the reaction policy to their presence to be out of scope. In this paper we present HEALED – HEaling & Attestation for Low-end Embedded Devices – which is the first attestation scheme that provides both detection and *healing* of compromised embedded devices. HEALED is applicable in both standalone and network settings. It allows measuring the software state of a device based on a novel Merkle Hash Tree (MHT) construction.

Main contributions of this paper are:

- Software Measurement: HEALED presents a novel measurement of prover's software state based on MHT which allows the verifier to pinpoint the exact software blocks that were modified.
- Device Healing: HEALED enables disinfecting compromised provers by restoring their software to a genuine benign state.
- Proof-of-concept Implementation: We implemented HEALED on two recent security architectures for low-end embedded devices as well as on our small network testbed composed of 6 Raspberry Pi-based drones.
- Performance Evaluation: We provide a thorough performance and security evaluation of HEALED based on our implementations and on network simulations.

2 HEALED

In this section, we present the system model, protocol goals, and a high-level overview of HEALED.

2.1 System Model

Our system model involves a group of two or more devices with a communication path between any two of them. A *device class* refers to the set of devices with the same software configuration. We denote by s be the number of devices in the smallest class. A device (regardless of its class) is denoted by D_i. Whenever a device D_v wants to attest another device D_p, we refer to the former as *prover*, and to the latter as *verifier*. As common to all current attestation schemes, we assume that D_v has prior knowledge of the expected benign software configuration of D_p. We also assume that D_v and D_p share a unique symmetric key k_{vp}.[1] Devices can be heterogeneous, i.e., have different software and hardware. However, all devices satisfy the minimal hardware requirements for secure remote attestation (see Sect. 2.2). Moreover, each device D_c can always find a *similar* device D_h with the same software/hardware configuration.

The goal of HEALED is to detect and eliminate malware on a device. HEALED consists of two protocols: (1) an attestation protocol between D_v and D_p, through which a verifier device D_v assesses the software state of a prover device D_p, and (2) a healing protocol between two similar devices D_h and D_c, through which a healing device D_h restores the software of a compromised device D_c to a benign state. Software state of a device refers to its static memory contents and excludes memory locations holding program variables.

2.2 Requirements Analysis

Threat Model. Based on a recent classification [4], we consider two types of adversaries:

1. *Local communication adversary:* has full control over all communication channels, i.e., it can inject, modify, eavesdrop on, and delay all packets exchanged between any two devices.
2. *Remote (software) adversary:* exploits software bugs to infect devices, read their unprotected memory regions, and manipulate their software state (e.g., by injecting malware).

We assume that every device is equipped with minimal hardware required for secure remote attestation, i.e., a read only memory (ROM) and a simple Memory Protection Unit (MPU) [12]. A remote software adversary cannot alter code protected by hardware (e.g., modifying code stored in ROM), or extract secrets from memory regions protected by special rules in the MPU. These memory regions are used to store cryptographic secrets and protocol intermediate variables.

Key Observation. Let $\texttt{Benign(}\texttt{t}_\texttt{a}\texttt{,}D_x\texttt{,}D_y\texttt{)}$ denote "device D_x believes that device D_y is not compromised at $\texttt{t}_\texttt{a}$, $\texttt{Equal(}\texttt{t}_\texttt{a}\texttt{,}D_x\texttt{,}D_y\texttt{)}$ denote "device D_x and device D_y have the same software state at time $\texttt{t}_\texttt{a}$. We make the following key observation:

– Healing: if two devices D_x and D_y have the same software state, then either both are benign or both are compromised.

[1] In the case of networks of embedded devices, we rely on the initialization protocol of existing collective attestation schemes for sharing software configurations and symmetric keys between devices [8].

$$\forall \mathtt{x}\ \forall \mathtt{y}\ \forall \mathtt{y}\ \forall \mathtt{t_a}\ \mathtt{Equal}(\mathtt{t_a}, D_y, D_z) \wedge \mathtt{Benign}(\mathtt{t_a}, D_x, D_y) \rightarrow \mathtt{Benign}(\mathtt{t_a}, D_x, D_z)$$

Consequently, healing can be supported by letting similar devices (i.e., devices having the same software configuration) attest and recover each other.

Objectives. A remote attestation protocol should not only detect presence of malware on a compromised devices, it should also identify exact regions in memory, where the malware resides in order to eliminate it. Consequently, a remote attestation protocol should have the following properties:

- Exact measurements: The measurement process on the prover should be capable of detecting software compromise and determining exact memory regions that have been manipulated.
- Healing: The protocol should allow secure and efficient disinfection of compromised devices, i.e., enable restoring the software of a compromised device to a benign state with low overhead.

Requirements. A verifier device D_v shares a symmetric key k_{vp} with every prover device D_p that it needs to attest. Similarly, every healer device D_h shares a symmetric key k_{hc} with every compromised device D_c that it heals, i.e., every device D_i shares a key with some (or all) similar devices. For brevity we assume that all devices in the group share pairwise symmetric keys. This assumption applies to small groups of device and is indeed not scalable. To achieve better scalability, keys and software configurations management might follow the design of collective attestation [8,16]. Every device that is involved in one of the protocols, i.e., D_v, D_p, D_h, and D_c supports a lightweight trust anchor for attestation, e.g., devices are equipped with a small amount of ROM and a simple MPU. During the execution of the attestation and healing protocols there should exist a communication path (or logical link) between D_v and D_p and between D_h and D_c respectively.

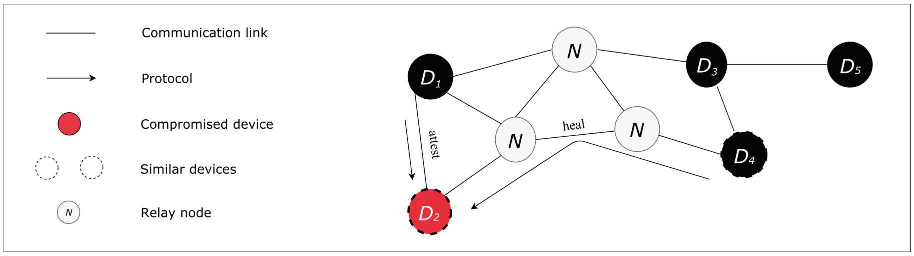


Fig. 1. HEALED in a group of 5 devices.

2.3 High Level Protocol Description

We now present a high level description of HEALED based on the example scenario shown in Fig. 1. The figure shows a group of five devices D_1–D_5, in addition to 3 communication nodes that are responsible for relaying messages between devices, e.g., routers. HEALED incorporates two protocols:

- attest: At predefined intervals, each device (e.g., D_1 in Fig. 1) acts as a verifier device and attests a random prover device (e.g., D_2). The prover uses a MHT-based measurement to report its software state. If a software compromise is detected by the verifier it initiates the healing protocol heal for the prover. The output of attest is a bit b_1 indicating whether attestation of D_p was successful.
- heal: When a compromised prover device (e.g., D_2) is detected, a benign healer device (e.g., D_4), which is similar to the prover, is identified. The healer uses the MHT-based measurement to pinpoint corrupted memory regions on the prover and restore them to their original state. The result of heal is a bit b_2 indicating whether healing by D_h was successful.

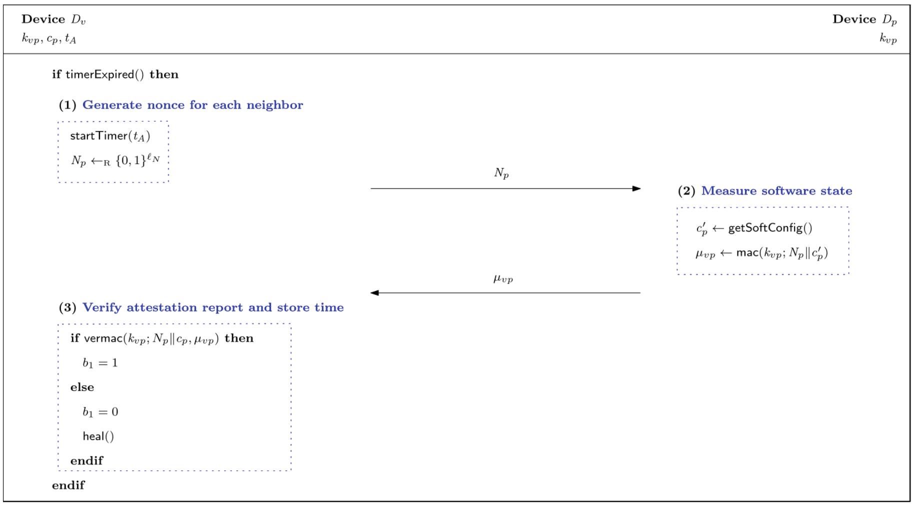

Device D_v
k_{vp}, c_p, t_A
Device D_p
k_{vp}
if timerExpired() then
(1) Generate nonce for each neighbor
startTimer(t_A)
$N_p \leftarrow_R \{0,1\}^{\ell_N}$
N_p
(2) Measure software state
$c'_p \leftarrow$ getSoftConfig()
$\mu_{vp} \leftarrow$ mac($k_{vp}; N_p \| c'_p$)
μ_{vp}
(3) Verify attestation report and store time
if vermac($k_{vp}; N_p \| c_p, \mu_{vp}$) then
$b_1 = 1$
else
$b_1 = 0$
heal()
endif
endif

Fig. 2. Protocol attest

2.4 Limitations

HEALED has some limitations in terms of system model, adversary, and application that we briefly described below:

- System model: HEALED is applicable to a set of devices under the same administrative control, e.g., devices in a smart home. Extending it to a more generic model, e.g., across multiple IoT environments, might require involving public key cryptography and using device manufacturers as certification

authorities. Moreover, gateways between multiple networks would need to be configured to exchange protocol messages.
- Adversary model: HEALED assumes that, at all times, at least one device of each class is not compromised, i.e., at most $s-1$ devices can be compromised at the same time.
- Application: HEALED provides secure and efficient detection and disinfection of compromised devices. However, it neither *guarantees* successful disinfection, nor does it prevent subsequent compromise of these devices.

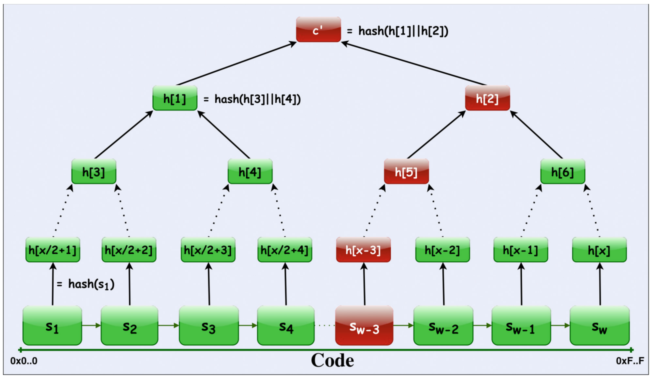


Fig. 3. Merkle Hash Tree of software configurations

3 Protocol Description

As mentioned earlier, HEALED includes the following protocols executed between devices acting as verifier D_v, prover D_p, healer D_h, and compromised device D_c.

Attestation. As shown in Fig. 2, each device D_v periodically acts as a verifier and attests a random device D_p acting as prover. Specifically, every t_A amount of time, D_v sends D_p an attestation request containing a random nonce N_p. Upon receiving the request D_p measures its software state, and creates a MAC μ_{vp} over the generated measurement c'_p and the received nonce based on the key k_{vp} shared with D_v. The MAC μ_{vp} is then sent back to D_v. Having the reference benign software configuration c_p of D_p and the shared key k_{vp}, D_v can verify μ_{vp}. Successful verification of μ_{vp} by D_v implies that D_p is in a benign software state. In this case attest returns $b_1 = 1$. On the contrary, if μ_{vp}'s verification failed, D_v deduce that D_p is compromised and initiates the healing protocol for D_p. In this case attest returns $b_1 = 0$.

The measurement of software state on D_p is created as a root of a Merkle Hash Tree (MHT) [23], as shown in Fig. 3. In particular, D_p divides the code to be attested into w segments: $s_1, \ldots, s_w$ of equal length, and computes hashes:

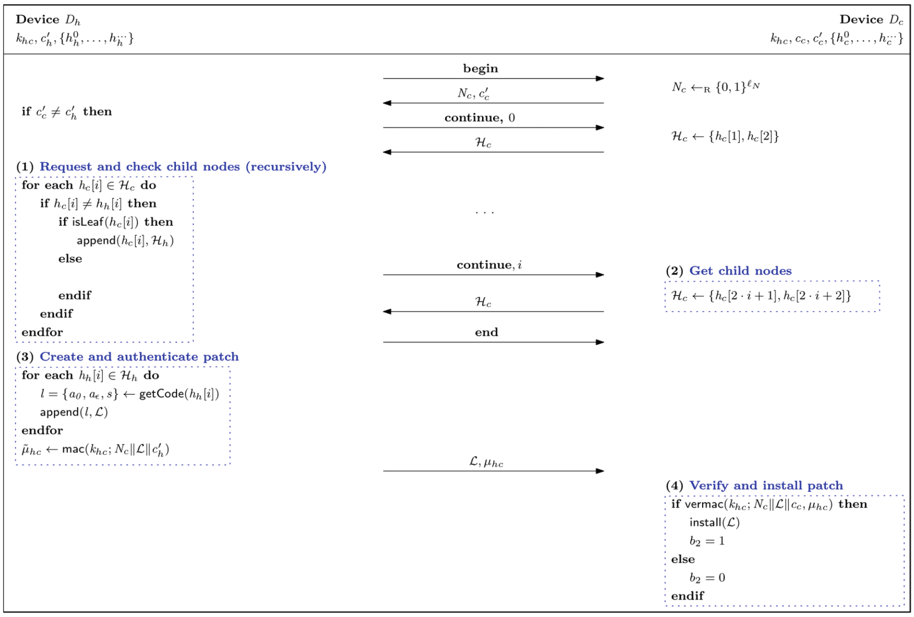
$h_p[\frac{x}{2}+1], \ldots, h_p[x]$ of each segment. A MHT is then constructed, with $h_p[\frac{x}{2}+1]$, $\ldots$, $h_p[x]$ as leaves and c'_p as the root, where x denotes the number of nodes in the MHT excluding the root node. Note that, a malware-infected code segment (e.g., s_{w-3}), leads to generation of false hash values along the path to the root. attest is formally:

$$\mathsf{attest}\big[D_v \;:\; k_{vp}, c_p, t_A; D_p \;:\; k_{vp}; * \;:\; -\big] \rightarrow \big[D_v \;:\; b_1; D_p \;:\; N_p\big].$$

Based on attest the compromise of any device will be detected.

Healing. Whenever a device D_v detects a compromised device D_c through attest, it searches for a healer device D_h, whose reference software configuration c_h is identical to that of D_c, i.e., a D_h that has the same version of the same software of D_c. Note that, if D_v and D_c are similar D_v directly initiates heal with D_c acting as healer device. Otherwise, D_v broadcasts the reference software configuration c_c of D_c along with a constant (protocol specific) Time-to-Live (TTL), and a random nonce N. Every device D_i that receives this tuple (1) checks TTL, and (2) compares c_c to its reference software configuration c_i. If c_c and c_i do not match, and TTL is not equal to zero, D_i re-broadcasts the tuple after TTL is decremented. Consequently, this tuple is flooded across devices until TTL is exceeded or a healer device D_h is found.

When a device D_h, whose reference software configuration c_h matches c_c, receives the tuple it sends a reply to D_v, which includes its *current* software configuration c'_h, authenticated along with the received nonce N, using a MAC based on the key k_{vh} shared with D_v.


Device D_h
$k_{hc}, c'_h, \{h^0_h, \ldots, h^{\cdots}_h\}$
Device D_c
$k_{hc}, c_c, c'_c, \{h^0_c, \ldots, h^{\cdots}_c\}$
begin
N_c, c'_c
continue, 0
$\mathcal{H}_c$
...
continue, i
$\mathcal{H}_c$
end
$\mathcal{L}, \mu_{hc}$
(1) Request and check child nodes (recursively)
(2) Get child nodes
(3) Create and authenticate patch
(4) Verify and install patch

```
$N_c \leftarrow_R \{0,1\}^{\ell_N}$
if $c'_c \neq c'_h$ then
$\mathcal{H}_c \leftarrow \{h_c[1], h_c[2]\}$
```

(1) Request and check child nodes (recursively)

```
for each $h_c[i] \in \mathcal{H}_c$ do
    if $h_c[i] \neq h_h[i]$ then
        if isLeaf($h_c[i]$) then
            append($h_c[i], \mathcal{H}_h$)
        else

        endif
    endif
endfor
```

(2) Get child nodes

```
$\mathcal{H}_c \leftarrow \{h_c[2 \cdot i + 1], h_c[2 \cdot i + 2]\}$
```

(3) Create and authenticate patch

```
for each $h_h[i] \in \mathcal{H}_h$ do
    $l = \{a_0, a_\epsilon, s\} \leftarrow$ getCode($h_h[i]$)
    append($l, \mathcal{L}$)
endfor
$\tilde{\mu}_{hc} \leftarrow$ mac($k_{hc}; N_c \| \mathcal{L} \| c'_h$)
```

(4) Verify and install patch

```
if vermac($k_{hc}; N_c \| \mathcal{L} \| c_c, \mu_{hc}$) then
    install($\mathcal{L}$)
    $b_2 = 1$
else
    $b_2 = 0$
endif
```

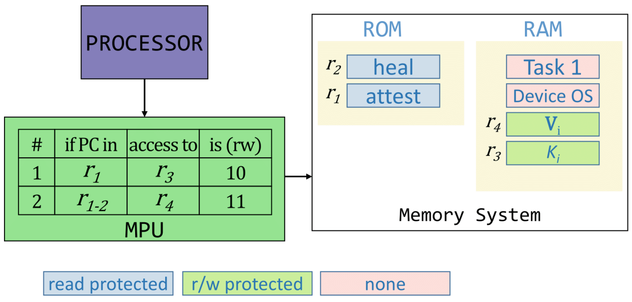
Fig. 4. Protocol heal

After proving its software trustworthiness, D_h initiates heal with D_c (as shown in Fig. 4). Note that, messages between D_h and D_c may go through D_v using the newly established route between D_h and D_v. D_h may also exploit an existing routing protocol to find a shorter path to D_c.

In details, D_h sends a protocol message **begin** to D_c. Upon receiving **begin**, D_c sends its software configuration c'_c and a fresh nonce N_c to D_h. D_h compares c'_c to its own software configuration c'_h. If the two configurations did not match, D_h replies requesting children $h_c[0]$ and $h_c[1]$ of c'_c in the Merkle Hash Tree (MHT) rooted at c'_c (protocol message **continue**). D_h continues recursively requesting child nodes of every hash that does not match its reference value (i.e., the value at the same position in D_h's tree) until leaf nodes are reached. Next, D_h sends a protocol message **end** indicating that it has reached leaf nodes. Finally, D_h adds a code segment l, for each modified leaf node, to the *patch* $\mathcal{L}$, authenticates $\mathcal{L}$ with a MAC based on k_{hc} and sends it back to D_c. A code segment $l = \{a_0, a_\epsilon, s\}$ is identified by its starting address a_0, its end address a_ϵ, and its code s. D_c, in turn verifies $\mathcal{L}$. If the verification was successful, it installs the patch, i.e., replaces segments indicated by $\mathcal{L}$ with the code in $\mathcal{L}$, and outputs $b_2 = 1$. Otherwise, D_c outputs $b_2 = 0$. heal is formally:

$$\begin{aligned}\mathsf{heal}\ \big[D_h : k_{hc}, c'_h, \{h_h[0], \ldots, h_h[x]\};\\ D_c : k_{hc}, c_c, c'_c, \{h_c[0], \ldots, h_c[x]\}; * : -\big] \rightarrow \big[D_h : N_c; D_c : \mathcal{L}, b_2\big].\end{aligned}$$

Device healing allows devices that have the same software configuration to recover from malware. By refusing to participate in the healing process (e.g., not installing the patch), D_c remains malicious and would not be able to prove its trustworthiness to other devices.



#	if PC in	access to	is (rw)
1	r_1	r_3	10
2	$r_{1\text{-}2}$	r_4	11

Fig. 5. Implementation of HEALED on SMART [12]

4 Implementation

In order to demonstrate viability and evaluate performance of HEALED we implemented it on two lightweight security architectures for low-end embedded devices that provide support for secure remote attestation: SMART [12] and TrustLite [18]. We also implemented HEALED on a testbed formed of six autonomous drones in order to demonstrate its practicality. In this section we present the details of these implementations.

4.1 Security Architectures

SMART [12] and TrustLite [18] are two lightweight security architectures for low-end embedded devices that enable secure remote attestation based on minimal hardware requirements. These two architectures mainly require: (1) A Read-Only Memory (ROM), which provides emutability and ensures integrity of the code it stores; and (2) A simple Memory Protection Unit (MPU), which controls access to a small region in memory where secret data is stored. Memory access control rules of MPU are based on the value of the program counter.

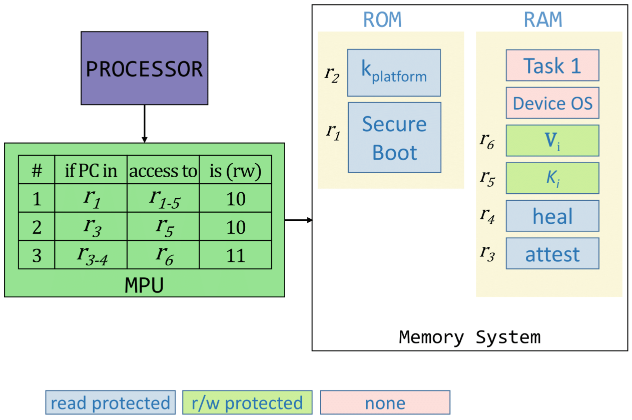
In SMART, the ROM code stores the attestation code and an attestation key, and the MPU ensures that the attestation code has exclusive access to the attestation key. As a consequence, only unmodified attestation code can generate an authentic attestation report. TrustLite exploits ROM and MPU to provide isolation of critical software components. In particular, ROM is used to ensure the integrity of a secure boot code which has exclusive access to a securely stored platform key. TrustLite enables isolation by initiating critical components via secure boot, which sets up appropriate memory access rules for each component in the MPU. We implemented HEALED on SMART replacing the attestation code in ROM, and on TrustLite as two isolated critical components. Our prototype implementations for SMART and TrustLite are shown in Figs. 5 and 6 respectively.

4.2 Implementation Details

Let $\mathcal{K}_i$ denote the set of all symmetric keys shared between a device D_i and any other device, and V_i denote the protocol variables processed and stored by HEALED. These include all nodes in the Merkle Hash Tree (MHT), including the root c_i. Integrity of HEALED code is protected through ROM of SMART (see Fig. 5), or secure boot of TrustLite (see Fig. 6). The secrecy of the set $\mathcal{K}_i$ of D_i is protected by the MPU of SMART and TrustLite (rule #1 in Fig. 5 and rule #2 in Fig. 6 respectively). Further, rules #2 in SMART and #3 in TrustLite ensure that variables processed and produced by HEALED are exclusively read- and write-accessible to HEALED's code.

4.3 Autonomous Testbed

In order to test and demonstrate the practicality of HEALED, we implemented and tested it on our autonomous drones testbed. The testbed is formed of six Raspberry Pi-based drones forming an ad-hoc network, where four of the drones are involved in HEALED while the remaining two drones act as relay drones. The Pi-s are equipped with a 1.2 GHz Quad-core 64-bit CPU and they are connected through a 150 MBit/s WiFi link. Our setup is shown in Fig. 7. Our implementation uses C programming language and is based on mbed TLS [6] cryptographic library.



#	if PC in	access to	is (rw)
1	r_1	$r_{1\text{-}5}$	10
2	r_3	r_5	10
3	$r_{3\text{-}4}$	r_6	11

Fig. 6. Implementation of HEALED on TrustLite [18]

5 Performance Evaluation

HEALED was evaluated on SMART [12], TrustLite [18], and on the drones testbed. The results of evaluation on TrustLite and the runtimes on our drones testbed are presented in this section. Results for SMART are very similar to those of TrustLite and will therefore be omitted.

Hardware Costs. A comparison between the hardware costs of our implementation of HEALED and that of the existing implementation of TrustLite [18] is shown in Table 1. As shown in the table, HEALED requires 15324 LUTs and 6154 registers in comparison to 15142 LUTs and 6038 registers required by TrustLite. In other words, HEALED incurs a negligible additional increase of 1.20% and

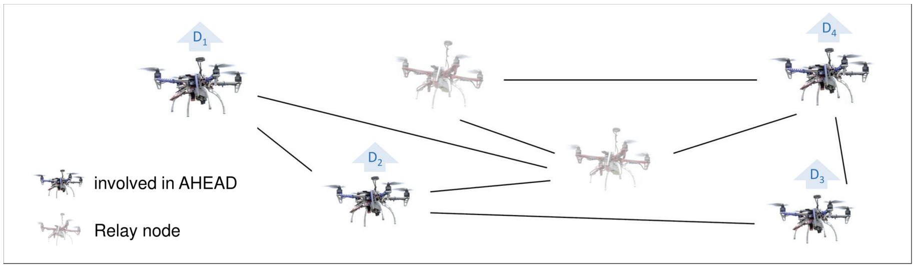


Fig. 7. Testbed setup

1.92% on the original hardware costs of TrustLite in terms of number of LUTs and registers respectively.

Memory Requirements. TrustLite already includes all the cryptographic operation that are involved in HEALED. Implementing HEALED on TrustLite required incorporating the code that is responsible for handling protocol messages and generating the Merkle Hash Tree (MHT). Further, every device D_i needs to securely store g_i symmetric keys (20 bytes each), where g_i corresponds to the number of devices D_i is expected to attest or heal. For every device D_i, g_i is upper bounded by the total number n of devices involved in HEALED. Furthermore, D_i should store the entire MHT that represents its benign software configuration. MHT size depends on the size of the code and the number of code segments. Each hash value is represented by 20 bytes.

Table 1. Hardware cost of HEALED

	Look-up Tables	Registers
TrustLite	15142	6038
HEALED	15324	6154
% of increase	1.20%	1.92%

Energy Costs. We estimated the energy consumption of HEALED based on reported energy consumption for MICAz and TelosB sensor nodes [24].[2] Note that, SMART [12] and TrustLite [18] support the same class of low-end devices that these sensor nodes belong to. Figure 8 shows the estimated energy consumption of `attest` and `heal` as function of the number of attested and healed devices respectively. We assume 100 KB of code divided into 128 segments.

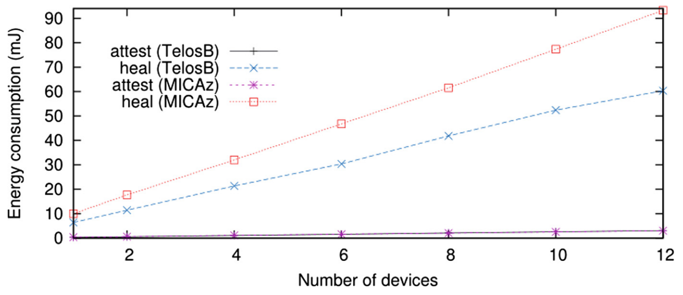


Fig. 8. Energy consumption of HEALED

[2] It is not possible to provide accurate measurements of the energy consumption of HEALED since our FPGA implementations of SMART and TrustLite tend to consume considerably more energy than manufactured chips.

Energy consumption of both the healing and attestation protocols increases linearly with the number of attested/healed devices. Moreover, this consumption can be as low as 21 mJ for attesting then healing 4 devices.

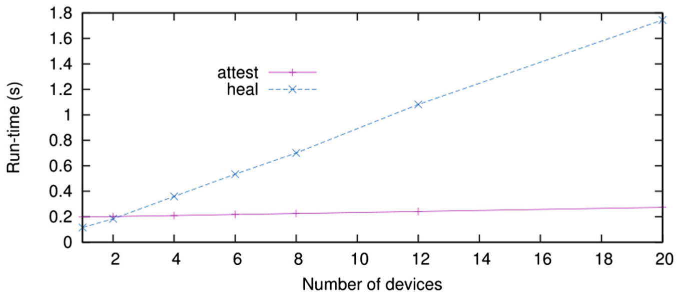


Fig. 9. Runtime of HEALED

Simulation Results. In order to measure the runtime of HEALED we used network simulation. We based our simulation on OMNeT++ [25] network simulator, where we emulated cryptographic operations as delays based on measurements we made for these operations on SMART [12] and TrustLite [18]. We measured the runtime of attest and heal for different number of attested/healed devices. We also varied the number of hops between the compromised device and the healer, as well as the number w of segments the attested code is divided into. The results of our simulation are shown in Figs. 9, 10, and 11.

As shown in Fig. 9 runtimes of attest and heal increase linearly with the number of attested and healed devices respectively. Further, these runtimes can be as low as 0.6 s for attesting then healing 4 devices.

Figure 10 shows the runtime of heal when the attested code is divided into 128 segments. As can be seen in the figure, the runtime of heal increases linearly with the number of hops between the healer D_h and the compromised device D_c. Finally, Fig. 11 shows the run-time of heal and getConfig (i.e., time needed to create the Merkle Hash Tree) when D_h and D_c are 10 hops away. As shown in the figure, the runtime of heal is logarithmic in the number of segments, while getConfig has a low run-time which is linear in the number of segments.

Note that, runtime of heal decreases with the number of segments, due to consequent decrease in code that should be transferred to D_c. Increasing the number of segments indeed increases the number of rounds of heal by increasing the size of MHT. However, the effect of this increase on the performance of heal is overshadowed by the huge reduction in the communication overhead.

Our simulation results also show that the runtimes of heal and attest are constant in the size of the network. These results are omitted due to space constraints. On the other hand, increasing the size of the network while keeping the number of similar devices constant could increase the expected number of hops between a healer D_h and a compromised device D_c. This would indeed lead to an increase in the runtime of heal (see Fig. 10).

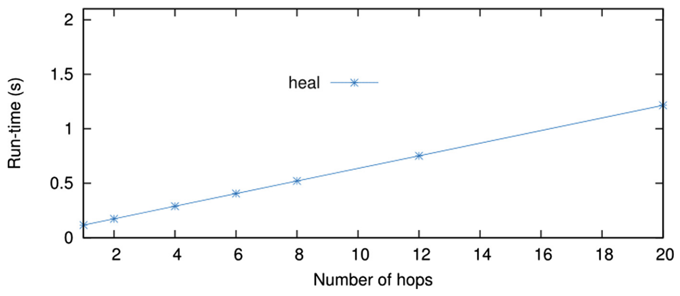


Fig. 10. Runtime of heal as function of number of hops

Drones Testbed. We also measured the runtime of HEALED on our drones testbed shown in Fig. 7. These runtimes are smaller than those of TrustLite since our Raspberry Pi-s utilize a much more powerful processor. The runtime of attest on drone D_2 attesting drones D_1 and D_3 is 11 ms, and the runtime of heal on drone D_1 healing drone D_4 through one relay node is 34 ms. Note that, the attested code is 100 KB in size and is divided into 128 segments. Further, these runtimes are averaged over 100 executions.

6 Security Consideration

Recall that the goal of HEALED is to allow secure detection and disinfection of compromised devices. We formalize this goal as a security experiment $\mathbf{Exp}_{\mathcal{A}}$, where the adversary $\mathcal{A}$ interacts with involved devices. In this experiment $\mathcal{A}$ compromises the software of two similar devices D_c and D_h. Then, after a polynomial number (in ℓ_{mac}, ℓ_{hash}, and ℓ_N) of steps by $\mathcal{A}$, one verifier device D_v outputs its decision b_1 signifying whether D_c is benign. The compromised device D_h executes heal with D_c which outputs b_2 signifying whether healing was successful. The result of the experiment is defined as the OR of outputs b_1 and b_2 of D_v and D_c respectively, i.e., $\mathbf{Exp}_{\mathcal{A}} = b \mid b = b_1 \vee b_2$. A secure attestation & healing scheme is defined as follows:

Definition 1 (Secure attestation & healing). *An attestation & healing scheme is secure if* $\Pr\left[b = 1 | \mathbf{Exp}_{\mathcal{A}}(1^{\ell}) = b\right]$ *is negligible in* $\ell = f(\ell_{\mathsf{mac}}, \ell_{\mathsf{hash}}, \ell_N)$*, where function* f *is polynomial in* ℓ_{mac}*,* ℓ_{hash}*, and* ℓ_N*.*

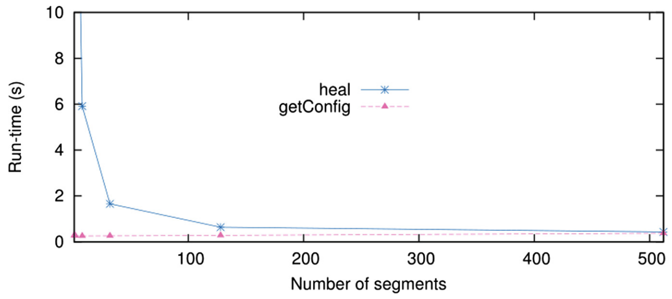


Fig. 11. Runtime of heal vs. getConfig

Theorem 1 (Security of HEALED). *HEALED is a secure attestation & healing scheme* (Definition 1) if the underlying MAC scheme is selective forgery resistant, and the underlying hash function is collision resistant.

Proof sketch of Theorem 1. $\mathcal{A}$ can undermine the security HEALED by either tricking D_v into returning $b_1 = 1$ or tricking D_c into returning $b_2 = 1$ We distinguish among the following two cases:

– $\mathcal{A}$ *attacks* attest: In order for D_v to return $b_1 = 1$ it should receive an attestation report containing a MAC $\mu_{vc} = \mathsf{mac}(k_{vc}; N_c \| c_c)$, where k_{vc} is the symmetric key shared between D_v and D_c, N_c is the fresh random nonce sent from D_v to D_c, and c_c is a benign software configuration of D_c. Consequently, $\mathcal{A}$ can try to: (1) extract the symmetric k_{vc} and generate such a MAC, (2) modify the measurement process on D_c to return a MAC over benign software configuration regardless of the software state on D_c, (3) replay an old attestation report containing a MAC $\mu_{\text{old}} = \mathsf{mac}(k_{vc}; N_{\text{old}} \| c_c)$ over a benign software configuration c_c and an old nonce N_{old}, (4) forge a MAC $\mu_{vc} = \mathsf{mac}(k_{vc}; N_c \| c_c)$ over a benign software configuration c_c and the current nonce N_c, or (5) modify the code on D_c in a way that is not detectable by the measurement process. However, the adversary is not capable of performing (1) and (2) since the secrecy of the key k_{vc} and the integrity of the measurement code are protected by the hardware of the underlying lightweight security architecture. Moreover, since D_v is always sending a fresh random nonce, the probability of success of (3) is negligible in ℓ_N. Furthermore, the probability of $\mathcal{A}$ being able to forge a MAC as in (4) is negligible in ℓ_{mac}. Finally, modifying the value of one bit of D_c's code would change the hash value of the segment containing this bit. This will change the hash value on the higher level in the Merkle Hash Tree and so on leading to a different root value, i.e., a different software configuration. Consequently, in order to perform (5) $\mathcal{A}$ should find at least on collision of the hash function that is used for constructing the MHT which is negligible in ℓ_{hash}.
– $\mathcal{A}$ *attacks* heal*:* In order for D_c to return $b_2 = 1$ it should receive a healing message containing a patch $\mathcal{L}$ and a MAC $\mu_{hc} = \mathsf{mac}(k_{hc}; N_c \| \mathcal{L} \| c_c)$, where k_{hc}

is the symmetric key shared between D_h and D_c, N_c is the fresh random nonce sent from D_c to D_h, and c_c is a benign software configuration of D_c. Similar to attest $\mathcal{A}$ may try to extract k_{hc}, modify the code responsible for generating the healing message, replay an old healing message, forge μ_{hc}, or compromise D_h in a way that is not detectable by the measurement process. However, because of the security of the underlying hardware and cryptographic primitives the success probabilities of these attacks are negligible in ℓ_{mac}, ℓ_{hash}, and ℓ_N. Indeed D_c may refuse to execute the healing protocol or install the patch, thus remaining compromised. However, the compromise of D_c will be detected by any subsequent attestation. One remedy for this problem could incorporate performing a subsequent attestation for healed devices and reporting devices that do not comply to the healing protocol.

This means that the probability of $\mathcal{A}$ bypassing the attestation protocol or infecting a benign device through the healing protocol is negligible in ℓ_{mac}, ℓ_{hash}, and ℓ_N. Consequently, HEALED is capable of securely detecting and disinfecting compromised devices. □

7 Related Work

Attestation. Attestation is a security service that aims at the detection of (malicious) unintended modifications to the software state of a device. Attestation is typically realized as an interactive protocol involving two entities: a verifier and a prover. Through this protocol the prover sends the verifier an attestation report indicating its current software state. Existing attestation schemes can be categorized into their main classes: (1) software-based attestation [14,17,20,32–34] which does not requires hardware support, but is based on strong assumptions and provides weak security guarantees; (2) hardware-based attestation [19,21,22,27,29,31,35] which provides stronger security guarantees based on complex and expensive security hardware; and (3) hybrid attestation [12,13,18] which aims at providing strong security guarantees while imposing minimal hardware costs. Additionally, recent advances have lead to the development of attestation schemes for verifying the intergrity of networks of embedded devices – collective attestation [5,8,15], and for detecting runtime attacks – control-flow attestation [3,11,37]. All existing attestation schemes, regardless of the type, aim at the detection of software compromise and overlook the problem of disinfecting compromised devices. These schemes usually consider the reaction policy to malware detection to be out of scope. HEALED is, to the best of our knowledge, the first attestation scheme that allows the detection and elimination of software compromise in both single-device and group settings.

Software Update and Healing. There is not much of prior work on attestation that allows the disinfection of compromised devices. SCUBA [33] leverages verifiable code execution based on software-based attestation to guarantee an untampered execution of a software update protocol. While SCUBA is built on top of a software-based attestation scheme that is based on unrealistic assumptions [7] to perform software update, HEALED leverages a lightweight security architecture to provide security guarantees regarding efficient disinfection of compromised devices. POSH [28] is a self-healing protocol for sensor networks which enables collective recovery of sensor nodes from compromise. The core idea of POSH is to enable sensor nodes to continuously compute new keys that are unknown to the adversary based on randomness provided by other sensors. Consequently, an adversary that compromises a device and extracts its current key would not be capable of extracting its future keys. TUF [30] is a software update for embedded systems that aims at reducing the impact of key compromise on the security of software update. TUF is based on role separation and multisignatures, where particular signatures using distinct private keys ensure different properties of the software update, e.g., timeliness or authenticity. ASSURED [9] enables applying secure update techniques, such as TUF, to the IoT setting while providing end-to-end security and allowing the verification of successful updates. In HEALED, we rely on a lightweight security architecture for protecting the secrecy of the keys and leverage MHT to restore the software state of compromised devices. Finally, PoSE [26] presents a secure remote software update for embedded devices via proof of secure erasure. The protocol allows restoring a device to its benign software state by ensuring the erasure of all code on that device. However, PoSE imposes a high communication overhead which is linear in the size of the genuine software. Moreover, similar to all existing software-based attestation protocols, PoSE assumes adversarial silence during the execution of the update protocol.

8 Conclusion

Most of the prominent attacks on embedded devices are at least started through malware infestation [1,2,10,36]. Remote attestation aims at tackling the problem of malware infestation by detecting device software compromise. However, current attestation schemes focus on the detection of malware, and ignore the problem of malware removal. These schemes usually consider the reaction to software compromise to be an orthogonal problem. In this paper, we present HEALED – the first attestation scheme for embedded devices which is capable of disinfecting compromised devices in a secure and efficient manner. The core of HEALED is a software measurement process based on Merkle Hash Tree (MHT) which allows identifying infected memory regions, and a healing protocol that efficiently restores these regions to their benign state. We implemented HEALED on two lightweight security architectures that support remote attestation and on an autonomous drones testbed. Moreover, we evaluated the energy, runtime, and hardware costs of HEALED based on measurements of real execution and on network simulation.

Acknowledgements. We thank the anonymous reviewers and, in particular, Alvaro Cardenas for his constructive feedback. This research was co-funded by the German

Science Foundation, as part of project S2 within CRC 1119 CROSSING, HWSec, and Intel Collaborative Research Institute for Collaborative Autonomous & Resilient Systems (ICRI-CARS). Gene Tsudik was supported in part by: (1) DHS under subcontract from HRL Laboratories, (2) ARO under contract W911NF-16-1-0536, and (3) NSF WiFiUS Program Award 1702911.

References

1. Target attack shows danger of remotely accessible HVAC systems (2014). http://www.computerworld.com/article/2487452/cybercrime-hacking/target-attack-shows-danger-of-remotely-accessible-hvac-systems.html
2. Jeep Hacking 101 (2015). http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101
3. Abera, T., et al.: C-FLAT: control-flow attestation for embedded systems software. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016, pp. 743–754. ACM, New York (2016), https://doi.org/10.1145/2976749.2978358
4. Abera, T., et al.: Invited - things, trouble, trust: On building trust in iot systems. In: Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, pp. 121:1–121:6. ACM, New York (2016). https://doi.org/10.1145/2897937.2905020
5. Ambrosin, M., Conti, M., Ibrahim, A., Neven, G., Sadeghi, A.R., Schunter, M.: SANA: secure and scalable aggregate network attestation. In: Proceedings of the 23rd ACM Conference on Computer & Communications Security, CCS 2016 (2016)
6. ARM Limited: SSL library mbed TLS/polarssl (2016). https://tls.mbed.org/
7. Armknecht, F., Sadeghi, A.R., Schulz, S., Wachsmann, C.: A security framework for the analysis and design of software attestation. In: ACM Conference on Computer and Communications Security (2013)
8. Asokan, N., et al.: SEDA: scalable embedded device attestation. In: Proceedings of the 22nd ACM Conference on Computer & Communications Security, CCS 2015, pp. 964–975 (2015)
9. Asokan, N., Nyman, T., Rattanavipanon, N., Sadeghi, A., Tsudik, G.: Assured: architecture for secure software update of realistic embedded devices. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **37**(11), 2290–2300 (2018)
10. Botnet, M.: Website (2016). https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html
11. Dessouky, G., et al.: LO-FAT: low-overhead control flow attestation in hardware. In: 54th Design Automation Conference (DAC 2017), June 2017
12. Eldefrawy, K., Tsudik, G., Francillon, A., Perito, D.: SMART: secure and minimal architecture for (establishing a dynamic) root of trust. In: Network and Distributed System Security Symposium (2012)
13. Francillon, A., Nguyen, Q., Rasmussen, K.B., Tsudik, G.: A minimalist approach to remote attestation. In: Design, Automation & Test in Europe (2014)
14. Gardner, R., Garera, S., Rubin, A.: Detecting code alteration by creating a temporary memory bottleneck. IEEE Trans. Inf. Forensics Secur. **4**(4), 638–650 (2009)
15. Ibrahim, A., Sadeghi, A.R., Tsudik, G.: DARPA: device attestation resilient against physical attacks. In: Proceedings of the 9th ACM Conference on Security and Privacy in Wireless and Mobile Networks. WiSec 2016 (2016)
16. Ibrahim, A., Sadeghi, A.R., Tsudik, G.: US-AID: unattended scalable attestation of IOT devices. In: Proceedings of the 37th IEEE International Symposium on Reliable Distributed Systems, SRDS 2018 (2018)

17. Kennell, R., Jamieson, L.H.: Establishing the genuinity of remote computer systems. In: USENIX Security Symposium (2003)
18. Koeberl, P., Schulz, S., Sadeghi, A.R., Varadharajan, V.: TrustLite: a security architecture for tiny embedded devices. In: European Conference on Computer Systems (2014)
19. Kovah, X., Kallenberg, C., Weathers, C., Herzog, A., Albin, M., Butterworth, J.: New results for timing-based attestation. In: IEEE Symposium on Security and Privacy, pp. 239–253 (2012)
20. Li, Y., McCune, J.M., Perrig, A.: VIPER: verifying the integrity of peripherals' firmware. In: ACM Conference on Computer and Communications Security (2011)
21. McCune, J.M., et al.: TrustVisor: efficient TCB reduction and attestation. In: Proceedings of the 2010 IEEE Symposium on Security & Privacy, S&P 2010, pp. 143–158 (2010)
22. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: an execution infrastructure for TCB minimization. SIGOPS Operating Syst. Rev. **42**(4), 315–328 (2008)
23. Merkle, R.C.: Protocols for public key cryptosystems. In: IEEE Symposium on Security and Privacy, pp. 122–134. IEEE Computer Society (1980). http://dblp.uni-trier.de/db/conf/sp/sp1980.html#Merkle80
24. de Meulenaer, G., Gosset, F., Standaert, O.X., Pereira, O.: On the energy cost of communication and cryptography in wireless sensor networks. In: IEEE International Conference on Wireless and Mobile Computing (2008)
25. OpenSim Ltd.: OMNeT++ discrete event simulator. http://omnetpp.org/ (2015)
26. Perito, D., Tsudik, G.: Secure code update for embedded devices via proofs of secure erasure. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 643–662. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15497-3_39
27. Petroni, Jr., N.L., Fraser, T., Molina, J., Arbaugh, W.A.: Copilot – a coprocessor-based Kernel runtime integrity monitor. In: USENIX Security Symposium, pp. 13–13. USENIX Association (2004)
28. Pietro, R.D., Ma, D., Soriente, C., Tsudik, G.: POSH: proactive co-operative self-healing in unattended wireless sensor networks. In: 2008 Symposium on Reliable Distributed Systems, October 2008, pp. 185–194 (2008)
29. Sailer, R., Zhang, X., Jaeger, T., Van Doorn, L.: Design and implementation of a TCG-based integrity measurement architecture. In: Proceedings of the 13th USENIX Security Symposium, pp. 223–238 (2004)
30. Samuel, J., Mathewson, N., Cappos, J., Dingledine, R.: Survivable key compromise in software update systems. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 61–72. CCS 2010. ACM, New York (2010). https://doi.org/10.1145/1866307.1866315
31. Schellekens, D., Wyseur, B., Preneel, B.: Remote attestation on legacy operating systems with trusted platform modules. Sci. Comput. Program. **74**(1), 13–22 (2008)
32. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: SWATT: software-based attestation for embedded devices. In: IEEE Symposium on Security and Privacy (2004)
33. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: SCUBA: secure code update by attestation in sensor networks. In: ACM Workshop on Wireless Security (2006)
34. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In: ACM Symposium on Operating Systems Principles (2005)

35. Trusted Computing Group (TCG): Website. http://www.trustedcomputinggroup.org (2015)
36. Vijayan, J.: Stuxnet renews power grid security concerns, June 2010. http://www.computerworld.com/article/2519574/security0/stuxnet-renews-power-grid-security-concerns.html
37. Zeitouni, S., et al.: ATRIUM: runtime attestation resilient under memory attacks. In: 2017 International Conference on Computer Aided Design, ICCAD 2017, November 2017