T-CUP: A TPM-Based Code Update Protocol Enabling Attestations for Sensor Networks

Steffen Wagner[1], Christoph Krauß[1], and Claudia Eckert[2]

[1] Fraunhofer Research Institution AISEC, Garching, Germany
{steffen.wagner,christoph.krauss}@aisec.fraunhofer.de
[2] TU München, Dpt. of Computer Science, Chair for IT Security, Garching, Germany
claudia.eckert@in.tum.de

Abstract. In this paper, we propose a secure code update protocol for TPM-equipped sensor nodes, which enables these nodes to prove their trustworthiness to other nodes using efficient attestation protocols. As main contribution, the protocol provides mechanisms to maintain the ability of performing efficient attestation protocols after a code update, although these protocols assume a trusted system state which never changes. We also present a proof of concept implementation on IRIS sensor nodes, which we have equipped with Atmel TPMs, and discuss the security of our protocol.

Keywords: Wireless Sensor Network, Security, Node Compromise, TPM, Attestation, Secure Code Update.

1 Introduction

Wireless sensor networks (WSNs) [1] can be used for various security-critical applications, such as military surveillance. Sensor nodes with embedded sensing, computation, and wireless communication capabilities monitor the physical world and send data through multi-hop communication to a central base station. The resources of a sensor node are severely constrained since they are mainly designed to be cheap and battery-powered.

Since sensor nodes are often deployed in unattended and even hostile environments, node compromise is a serious issue. By compromising a sensor node, an adversary gets full access to data such as cryptographic keys stored on the node. Especially sensor nodes which perform special tasks for other sensor nodes (e.g., key management) are a valuable target. One approach to protect the cryptographic keys on such nodes is the use of a Trusted Platform Module (TPM) [15]. The TPM is basically a smartcard and can be used to create a secure storage and execution environment. The TPM additionally provides mechanisms to realize attestation protocols where the sensor nodes can prove that no adversary has tampered with their components.

However, previously proposed attestation protocols for WSNs, e.g., in [9], rely on a trusted system state which never changes. The main idea is to use the TPM to cryptographically bind certain attestation values (e.g., symmetric keys) to a trusted initial platform configuration. The platform configuration is validated during each boot process by calculating hash values for bootloader (acting as Core Root of Trust for Measurement (CRTM)), operating system (OS), and all installed applications and comparing them with references values protected by the TPM. Only if they match, access to the attestation values is possible. Thus, any code update, which might be necessary to patch security vulnerabilities or add new functionalities, would result in a different system state which prevents successful attestations.

In this paper, we present T-CUP, a secure code update protocol which enables TPM-based attestation protocols and provides mechanisms to validate the authenticity, integrity, and freshness of the wirelessly transmitted code update. We also present a proof of concept implementation and security discussion.

2 Related Work

Existing over-the-air programming (OTAP) protocols, such as Deluge [7], Infuse [2], or MNP [10], mainly focus on the (efficiency of the) update procedure, but do not consider security. In [4,3,12,13,8], code update protocols with security mechanisms have been proposed which are often based on an existing OTAP protocol, mostly Deluge. Secured hash chains are used to ensure authenticity and integrity of the individual parts of the code update. Because of the chaining, only the first hash needs to be protected by some cryptographic mechanism. However, the key used to protect this hash value must not be accessible by an adversary since this would enable him to create false code updates. In [4,3,12,14], digital signatures are used for this purpose which have much higher computational costs than symmetric approaches. The protocols proposed in [13,8] are solely based on symmetric (hash-based) mechanisms. However, all previously proposed protocols are not directly applicable to update TPM-equipped sensor nodes while maintaining the ability to perform attestations.

The use of a TPM for attestation protocols in hybrid WSNs where only a minority of special sensor nodes are equipped with a TPM has initially been introduced by Krauß et al. [9]. They propose two attestation protocols which either enable a single node (including the base station) or multiple sensor nodes to simultaneously verify the trustworthiness of a TPM-equipped sensor node. The main idea is to use the `Sealing` function of the TPM to bind certain attestation values (symmetric keys or values of a hash chain) to an initial trustworthy platform configuration. However, code updates are not considered.

Using the TPM in WSNs has been also proposed in [5] and [6] where all sensor nodes of a WSN are equipped with a TPM to perform public key cryptography.

3 Setting and Notation

In this section, we explain the setting and define the notation for our protocol.

3.1 Setting

We consider a hybrid WSN consisting of a large number of common cluster sensor nodes (CNs) and very few TPM-equipped cluster head (CH) nodes. CHsperform special tasks and services for a certain number of CNs. Data is sent via multi-hop communication to one central base station (BS) which is assumed to be trustworthy, i.e., cannot be compromised.

Before node deployment, BS and all CHs are initialized in a trusted environment. The TPM of each CH is initialized by generating asymmetric key pairs which are only used within the TPM and marked as "non-migratable", i.e., the private key cannot be extracted from the TPM. These asymmetric keys are used by the sealing function of the TPM to cryptographically bind shared symmetric keys between BS and the CHs to an initial trusted system state. Likewise, a timestamp which indicates the current version of the system software is also bound to the trusted system state.

Furthermore, we assume an adversary which tries to compromise a CH by attacking the code update protocol. The adversary can either try to physically compromise a CH or by performing attacks via the wireless channel. In the first case, the adversary directly tries to read out stored data such as cryptographic keys or tries to re-program the node with his own malicious code. In the latter case, the adversary can perform attacks such as eavesdropping on the wireless communication, manipulate transmitted packets, inject new or replay old packets. However, we assume that an adversary is not able to break cryptographic algorithms, e.g., decrypting encrypted messages without knowing the key or inverting hash functions. The adversary is also not able to access cryptographic keys which are protected by the TPM or reset the TPM.

3.2 Notation

Cluster heads are denoted as CH_i with $i = 1, \ldots, a$ and the cluster nodes as CN_j with $j = 1, \ldots, b$, where $b \gg a$ (cf. [9]).

A symmetric key K between the BS and the CH_i is referred to as K_{BS,CH_i}. Since the current version of the TPM does not support symmetric cryptographic operations internally, we allow this key to be stored in RAM for a short time.

Applying a cryptographic *hash function* H on data m is denoted with $H(m)$. A one-way *hash chain* [11] stored on a CH is denoted with $C^{CH} = c_0^{CH}, \ldots, c_n^{CH}$. A hash chain is a sequence of n hash values, each of fixed length l, generated by a hash function $H : \{0,1\}^* \rightarrow \{0,1\}^l$ by applying the hash function H successively on a seed value c_0, such that $c_{\nu+1} = H(c_\nu)$, with $\nu = 0, 1, \ldots, n-2, n-1$.

A specific system state of a TPM-equipped cluster head CH_i is referred to as *platform configuration* $P_{CH_i} := (PCR[0], \ldots, PCR[p])_{CH_i}$ and stored as *integrity measurement values* μ in the *platform configuration registers* (PCRs) of the TPM. To store the value of a measured (software) component in a PCR, the existing value is not replaced, but combined with the new value using $PCRExtend(PCR[i], \mu)$, which is specified as $PCR[i] \leftarrow SHA1(PCR[i] \,||\, \mu)$ [15].

For our protocol, we define two platform configurations referred to as *full* and *reduced platform configuration.* The full platform configuration $P_{CH_i}^{(0,\ldots,p)}$ uses at least two PCRs and must consider all software layers up to the OS and application layer (cf. Fig. 1, left). Similarly, we denote the reduced platform configuration as $P_{CH_i}^{(0,\ldots,r)}$, which uses $r<p$ registers and only considers the static trusted components up to the bootloader (cf. Fig. 1, right).

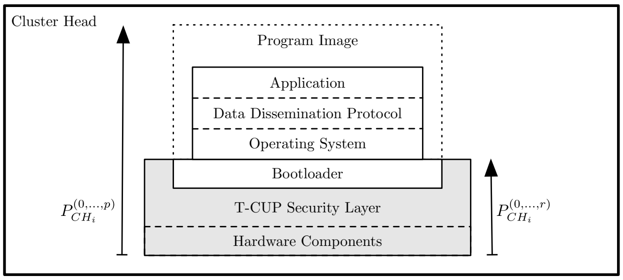

Fig. 1. T-CUP Security Layer on Cluster Head

Data m can be cryptographically bound to a specific platform configuration P by using the `TPM_Seal` command, which we call *Seal* for the sake of simplicity. Using the `TPM_Unseal` command (or simply *Unseal*), the TPM decrypts m only if the platform configuration has not been modified. Given a non-migratable asymmetric key pair (e^{CH_i}, d^{CH_i}) we denote the *sealing* of data m to the platform configuration P_{CH_i} with $\{m\}_{P_{CH_i}}^{e^{CH_i}} = Seal(P_{CH_i}, e^{CH_i}, m)$. For *unsealing* the sealed data $\{m\}_{P_{CH_i}}^{e^{CH_i}}$, it is necessary that the current platform configuration P'_{CH_i} is equal to P_{CH_i}: $m = Unseal(P'_{CH_i} = P_{CH_i}, d^{CH_i}, \{m\}_{P_{CH_i}}^{e^{CH_i}})$.

4 Protocol Description

In this section, we describe the concept of our proposed code update protocol.

4.1 The T-CUP Header

The code update is divided into pages $pg0$ to pgT, i.e., $upd = (pg0 \,||\, \ldots \,||\, pgT)$, as depicted in Fig. 2. To ensure the authenticity of such a code update, we define a *T-CUP Header*, which is shown in detail on the right of Fig. 2. This header includes the number of pages T, a timestamp ts, a chain of hashes and an HMAC ($hmac\_upd$) for the complete code update.

The cryptographic values of the T-CUP Header are calculated as follows: First, the HMAC $hmac\_upd$, which allows to verify the authenticity and integrity of the complete code update, is calculated using the shared symmetric key K_{BS,CH_i} between BS and CH_i (here simply K):

$$hmac\_upd = HMAC(K, upd) = HMAC(K, (pg0 \,||\, \ldots \,||\, pgT)) \ . \qquad (1)$$

After that, the chain of hashes is then generated in reversed order as shown in Fig. 2 (right): For page $T-1$, the hash is created as $h_T = H(pgT \,||\, hmac\_upd)$, i.e., by concatenating the data of page T with the HMAC of the complete code

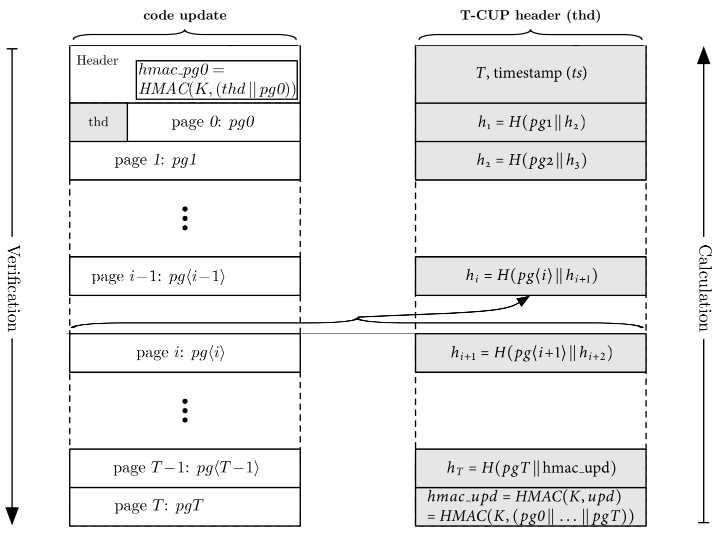

Fig. 2. Code update and T-CUP Header with chained hashes

update and hashing the result using a one way-hash function H. Starting from page $T-2$, the hash value h_i is created by concatenating the data of page i with the previously cacluated hash h_{i+1} and hashing the resulting value:

$$h_i = H(pg\langle i\rangle \,||\, h_{i+1}) \ . \tag{2}$$

Finally, a second HMAC, which is referred to as *hmac_pg0* and includes the T-CUP Header (particularly, hash h_1 and the timestamp) as well as the first page (with index 0), is generated (and stored in the code update header):

$$hmac\_pg0 = HMAC(K, (thd \,||\, pg0)) \ . \tag{3}$$

For the verification of the code update (cf. Fig. 2, left), we start with checking the HMAC *hmac_pg0*, which authenticates the T-CUP Header. The timestamp is compared with the sealed reference value to check the freshness. With the hash chain, we are able to validate the authenticity and integrity of the following parts of the code update page by page (cf. Section 4.3).

4.2 The T-CUP Security Layer

In addition to the *T-CUP Header* that protects the wirelessly transmitted code update, we also specify a *T-CUP Security Layer* beneath the OS (cf. Fig. 1). This layer protects the sensitive information stored on the CHs during a code update and maintains the ability to access sealed data even after the code update.

The general idea behind the *T-CUP Security Layer* results from the need to protect the sensitive information during a code update and preserve the trusted system state in order to access these information after the code update. The reason for preserving the trusted state is that the data is sealed to the initial trusted system state, which is changed by the code update. As a consequence, if the information was still sealed to the old platform configuration, it could not be unsealed after the update, which would make attestations impossible. But obviously, if the sensitive information was unsealed before the code update is performed, the sensor node and the attestation could be easily compromised. That is why all sensitive information need to be sealed even during a code update.

Thus, we define the *T-CUP Security Layer* as a reduced platform configuration for sealing data during a code update, which only considers those components that are not affected and modified by the code update, i.e., the CRTM, the bootloader, and the hardware components. Note that for our proof of concept implementation we assume that (one of the components of) TOSBoot is trustworthy since it acts as CRTM. For real implementations, we suggest a hardware CRTM to increase security

4.3 The T-CUP Protocol Steps

In this section, we describe the protocol steps of T-CUP in detail. The T-CUP protocol can be divided into three phases: (P1) Initialization and Dissemination, (P2) Validation and Preparation, and (P3) Verification and Processing. In the first phase, the code update is generated on the base station and distributed to CHs. In the second and third phase, CHs checks the authenticity, integrity, and freshness, prepares for the necessary reboot, and processes the code update after an additional verification. During the code update, all sensitive information is sealed to the security layer. It is resealed to the new full platform configuration after the code update is installed.

In Phase 1 (cf. Table 1), the base station first generates the program binary (P1.1) and then creates the T-CUP Header as described in Section 4.1 by setting the number of pages and timestamp in the T-CUP Header (P1.2) and by calculating the HMAC for the complete code update *hmac_upd*, the hash chain for the code update pages, and the HMAC for the first page *hmac_pg0* (P1.3 – P1.5). After that, the code update is disseminated in the network (P1.6).

In Phase 2 (cf. Table 2), the cluster head validates the code update (P2(a)) and prepares for the reboot (P2(b)). Thus, when the dissemination is initiated by the base station, CH eventually receives *hmac_pg0* and page 0 (P2.1). To verify the HMAC for the first page, CH first unseals the shared key K_{BS,CH_i} (P2.2), which is only possible if the node is still in a trustworthy system state:

$$K_{BS,CH_i} = Unseal(P_{CH_i}^{(0,\ldots,p)}, d^{CH_i}, \{K_{BS,CH_i}\}_{P_{CH_i}^{(0,\ldots,p)}}^{e^{CH_i}}) \ . \tag{4}$$

With the unsealed key, the cluster head can recalculate the HMAC and compare it with the reference value *hmac_pg0* from the global header (P2.3):

$$hmac\_pg0 \stackrel{?}{=} HMAC(K_{BS,CH_i}, thd \,||\, pg0) \ . \tag{5}$$

If the values are identical, the authenticity and integrity of page 0, the head of the hash chain, and the timestamp in the T-CUP Header is successfully validated.

Table 1. Phase 1: Initialization on Base Station

Step	Node	Data	Action/Description
P1(a): Initialization			
P1.1	*BS*	*binary code*	creates program binary
P1.2	*BS*	*#pages, timestamp*	sets number of pages and timestamp
P1.3	*BS*	*hmac_upd* $= MAC_{upd}^{K_{BS,CH_i}}$	creates a HMAC for the complete code update with the symmetric key K_{BS,CH_i}
P1.4	*BS*	h_i	creates hash values for each page
P1.5	*BS*	*hmac_pg0*	creates a HMAC for page 0
P1(b): Dissemination			
P1.6	$BS \rightarrow CH_i$	*upd*	disseminates code update

Otherwise, the code update protocol stops. To verify the freshness of the code update, the CH extracts the authenticated timestamp in step P2.4 and compares it with the sealed reference value. If the extracted timestamp indicates a more recent program binary, the current reference value is replaced with the timestamp from the code update (after sealing it). Otherwise, the protocol aborts since the code update is outdated.

After the validation of the T-CUP Header, CH requests the complete code update page by page and verifies the elements of the hash chain (P2.5), which ensure the integrity and authenticity of the included pages, by recalculating each value and comparing it with the expected result (cf. Section 4.1).

After CH has received the complete code update, it starts preparing for the reboot in order to program the new image by sealing the shared key K_{BS,CH_i} to the security layer in step P2.6:

$$\{K_{BS,CH_i}\}^{e^{CH_i}}_{P^{(0,\dots,r)}_{CH_i}} = Seal(P^{(0,\dots,r)}_{CH_i}, e^{CH_i}, K_{BS,CH_i}) \ . \tag{6}$$

CH also seals the HMAC for the complete code update to the reduced platform configuration (P2.7) to be able to verify the image after the reboot:

$$\{hmac\_upd\}^{e^{CH_i}}_{P^{(0,\dots,r)}_{CH_i}} = Seal(P^{(0,\dots,r)}_{CH_i}, e^{CH_i}, hmac\_upd) \ . \tag{7}$$

The second HMAC *hmac_upd* allows for an efficient verification of the complete code update again after the reboot, because it 1) is already implicitly authenticated, 2) requires only one calculation instead of calculating again all values of the hash chain, and 3) occupies less space. Thus, *hmac_upd* effectively preserves the effort already invested in authenticating and verifying the complete hash chain page by page.

As the final step of the preparation, CH reseals all sensitive information m, e.g., the attestation values such as a symmetric key, to the security layer (P2.8):

Table 2. Phase 2: Validation and Preparation

Step	Node	Data	Action/Description
P2(a): Validation			
P2.1	CH_i	*hmac_pg0*, page 0	receives page 0 and *hmac_pg0*
P2.2	CH_i	K_{BS,CH_i}	unseals the symmetric key
P2.3	CH_i	*hmac_pg0*	checks the HMAC
P2.4	CH_i	*timestamp*	checks timestamp
P2.5	CH_i	*upd*	receives complete *upd* page by page and validates each hash value
P2(b): Preparation			
P2.6	CH_i	$\{K_{BS,CH_i}\}^{e^{CH_i}}_{P^{(0,\ldots,r)}_{CH_i}}$	reseals the symmetric key to the security layer
P2.7	CH_i	$\{hmac\_upd\}^{e^{CH_i}}_{P^{(0,\ldots,r)}_{CH_i}}$	seals the HMAC for the complete code update to the security layer
P2.8	CH_i	$\{m\}^{e^{CH_i}}_{P'^{(0,\ldots,p)}_{CH_i}} \to \{m\}^{e^{CH_i}}_{P'^{(0,\ldots,r)}_{CH_i}}$	reseals the sensitive information to the security layer

$$\begin{aligned}\{m\}^{e^{CH_i}}_{P^{(0,\ldots,r)}_{CH_i}} &= \mathit{Seal}(P^{(0,\ldots,r)}_{CH_i}, e^{CH_i}, m) \\ &= \mathit{Seal}(P^{(0,\ldots,r)}_{CH_i}, e^{CH_i}, \mathit{Unseal}(P^{(0,\ldots,p)}_{CH_i}, d^{CH_i}, \{m\}^{e^{CH_i}}_{P^{(0,\ldots,p)}_{CH_i}}))\ .\end{aligned} \tag{8}$$

After resealing, CH reboots and executes the bootloader.

In Phase 3 (cf. Table 3), CH verifies the code update again to check if it is still unmodified (P3(a)) and processes the verified update (P3(b)). For the verification, the CRTM starts with measuring the security layer to create a reduced platform configuration $P'^{(0,\ldots,r)}_{CH_i}$ (P3.1). This platform configuration has to match the platform configuration, which has been specified to seal the shared key before the reboot, in order to unseal it (P3.2):

$$K_{BS,CH_i} = \mathit{Unseal}(P'^{(0,\ldots,r)}_{CH_i}, d^{CH_i}, \{K_{BS,CH_i}\}^{d^{CH_i}}_{P'^{(0,\ldots,r)}_{CH_i}})\ . \tag{9}$$

That is only the case if the security layer is still unmodified, i.e., if the equation $P'^{(0,\ldots,r)}_{CH_i} = P^{(0,\ldots,r)}_{CH_i}$ holds. CH also unseals the HMAC for the complete update (P3.3), where the same condition applies:

$$\mathit{hmac\_upd} = \mathit{Unseal}(P'^{(0,\ldots,r)}_{CH_i}, d^{CH_i}, \{\mathit{hmac\_upd}\}^{d^{CH_i}}_{P'^{(0,\ldots,r)}_{CH_i}})\ . \tag{10}$$

For the verification of the code update stored in memory, a fresh HMAC is calculated and compared with the unsealed HMAC reference value:

$$\mathit{hmac\_upd} = MAC^{K_{BS,CH_i}}_{upd} \stackrel{?}{=} HMAC(K_{BS,CH_i}, upd)\ . \tag{11}$$

Once the trustworthiness of the security layer and the code update is verified, the bootloader copys the binary to the program memory (P3.5). After that,

Table 3. Phase 3: Verification and Processing

Step	Node	Data	Action/Description
P3(a): Verification			
P3.1	CH_i	$P'^{(0,\dots,r)}_{CH_i}$	measures the security layer
P3.2	CH_i	K_{BS,CH_i}	unseals the symmetric key
P3.3	CH_i	$MAC^{K_{BS,CH_i}}_{upd}$	unseals the HMAC
P3.4	CH_i	$upd, MAC^{K_{BS,CH_i}}_{upd}$	uses the symmetric key to compare the unsealed MAC with a freshly calculated HMAC of the *upd*
P3(b): Processing			
P3.5	CH_i	upd	copies update to program memory
P3.6	CH_i	$P'^{(0,\dots,p)}_{CH_i}$	measures remaining components for a full platform configuration
P3.7	CH_i	$\{K_{BS,CH_i}\}^{e^{CH_i}}_{P^{(0,\dots,p)}_{CH_i}}$	CH_i seals K_{BS,CH_i} to the new trusted full platform configuration
P3.8	CH_i	$\{m\}^{e^{CH_i}}_{P'^{(0,\dots,r)}_{CH_i}} \rightarrow \{m\}^{e^{CH_i}}_{P'^{(0,\dots,p)}_{CH_i}}$	reseals the sensitive information to the new trusted full platform configuration P'

it measures the remaining software components and creates the full platform configuration, which includes the OS and application components. Using this new trusted full platform configuration, CH finally reseals the shared symmetric key (P3.7) as well as all other sensitive information (P3.8).

5 Implementation

As proof of concept, we implemented T-CUP on IRIS sensor nodes, which we connected with Atmel *AT97SC3204T* TPMs via I^2C, by extending the current de-facto standard code dissemination protocol *Deluge* [7] and the boot loader *TOSBoot* from TinyOS [16]. The *T-CUP Image Format* extends the specification of a Deluge image with the cryptographic information of the T-CUP Header to enable the verification of the authenticity, integrity, and freshness of the distributed code update. Based on the T-CUP Image Format specification, we have implemented the T-CUP protocol as (1) an interface script for the base station and (2) T-CUP components for CHs. The new T-CUP interface script *tos-tcup* is based on the Deluge interface script *tos-deluge* and can be used to initialize CHs prior to deployment, i.e., the cryptographic keys are generated and symmetric keys and initial timestamps are sealed to the initial trusted platform configuration. The T-CUP components for CHs consists of the TPM driver and extended Deluge and TOSBoot components for the dissemination and reprogramming.

6 Security Discussion

In this section, we evaluate the security of T-CUP. We first discuss an adversary performing attacks via the wireless channel and then an adversary that physically tampers with a CH (cf. Section 3.1).

To compromise a CH via wireless channel, an adversary can try to send his own malicious code update to CH. Lets assume that an adversary is able to do this. A code update which is accepted by CH must contain a valid *hmac_upd*. Since we assume an adversary is not able to break cryptographic algorithms (cf. Section 3.1), the adversary must be in possession of the symmetric key shared between BS and CH. To get access to the required key, the adversary must have either compromised BS or CH. However, this is a contradiction to the assumption that BS is trustworthy and that all keys on a CH are protected by the TPM. Thus, an adversary cannot inject his own malicious code update. The same applies to manipulations of eavesdropped valid code update sent by BS.

An adversary could also try to replay and install a valid old code update which is known to possess certain weaknesses, e.g., possible buffer overflows. However, CH verifies the freshness by comparing the timestamp in the header, which is protected by *hmac_pg0*, with the sealed reference value. Thus, an adversary would have to manipulate that timestamp and create a valid *hmac_pg0* which is a contradiction to our assumptions already mentioned above.

Now we consider the case where an adversary has physical access to CH and tries to compromise it. The adversary can try to manipulate the software components of a CH (cf. Fig. 1) to get access to the cryptographic keys. However, we assume that runtime attacks such as buffer overflows are not possible. Thus, an adversary has to install his malicious code and reboot CH. But after the reboot, the platform configuration has changed and the TPM denies access to the sealed cryptographic keys preventing a successful compromise.

Instead of manipulating the installed software, the adversary might tamper with a code update stored in the flash memory before it gets installed. However, CH verifies *hmac_upd* before the code update is installed. Thus, an adversary would have to forge the correct HMAC for the manipulated code update. But this would also require the adversary to break cryptography, compromise BS, or access TPM-protected keys which is contradictory to our assumptions.

The adversary might also try to exploit the (re)sealing to different platform configurations and the security layer. First, keys are sealed to the initial platform configuration which is assumed to be trustworthy. Thus, an adversary cannot perform successful manipulations during the unsealing and resealing of the keys to the security layer before a new update is installed. After a reboot, only the integrity of the security layer, including the CRTM and all necessary security services such as the HMAC engine, is checked. Thus, an adversary could theoretically manipulate the other software components above the security layer, i.e., OS and application components. However, this would have no effect, because the new trusted code update (since *hmac_upd* is valid) is installed by the security layer and overwrites the malicious code. Thus, also the resealing to the new platform configuration is performed when CH is in a trustworthy state.

7 Conclusion

In this paper, we presented T-CUP, a TPM-based code update protocol to secure distributed program images while still enabling attestation protocols based on binding keys to a trusted initial platform configuration. T-CUP provides mechanisms to validate the authenticity, integrity, and freshness of the wirelessly transmitted code update. To enable attestations, we introduced a new "virtual" security layer beneath the OS where attestation values are temporarily bound to during an update. Our protocol is based on efficient cryptographic primitives such as hash functions and MACs to avoid computational intensive digital signatures and unnecessary large messages. We also presented the feasibility of T-CUP in a proof of concept implementation and discussed the security of our protocol. T-CUP can handle an adversary attacking via the the wireless channel as well as an adversary which directly tampers with a CH using physical access.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A survey on sensor networks. IEEE Communications Magazine 40(8), 102–114 (2002)
2. Arumugam, M.U.: Infuse: a TDMA based reprogramming service for sensor networks. In: SenSys (2004)
3. Deng, J., Han, R., Mishra, S.: Secure code distribution in dynamically programmable wireless sensor networks. In: IPSN (2006)
4. Dutta, P.K., Hui, J.W., Chu, D.C., Culler, D.E.: Securing the Deluge Network Programming System. In: IPSN (2006)
5. Hu, W., Corke, P., Shih, W.C., Overs, L.: secFleck: A Public Key Technology Platform for Wireless Sensor Networks. In: Roedig, U., Sreenan, C.J. (eds.) EWSN 2009. LNCS, vol. 5432, pp. 296–311. Springer, Heidelberg (2009)
6. Hu, W., Tan, H., Corke, P., Shih, W.C., Jha, S.: Toward trusted wireless sensor networks. TOSN 7(1) (2010)
7. Hui, J.W., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale. In: SenSys (2004)
8. Kim, D.H., Gandhi, R., Narasimhan, P.: Castor: Secure code updates using symmetric cryptosystems. In: Real-Time Systems Symposium (2007)
9. Krauß, C., Stumpf, F., Eckert, C.: Detecting Node Compromise in Hybrid Wireless Sensor Networks Using Attestation Techniques. In: Stajano, F., Meadows, C., Capkun, S., Moore, T. (eds.) ESAS 2007. LNCS, vol. 4572, pp. 203–217. Springer, Heidelberg (2007)
10. Kulkarni, S.S., Wang, L.: MNP: Multihop Network Reprogramming Service for Sensor Networks. In: ICDCS (2005)
11. Lamport, L.: Password authentication with insecure communication. Communications of the ACM 24(11), 770–772 (1981)
12. Lanigan, P.E., Gandhi, R., Narasimhan, P.: Secure dissemination of code updates in sensor networks. In: SenSys (2005)
13. Lee, S., Kim, H., Chung, K.: Hash-based secure sensor network programming method without public key cryptography. In: Worksh. on World-Sensor-Web (2006)
14. Liu, A., Oh, Y.-H., Ning, P.: Secure and dos-resistant code dissemination in wireless sensor networks using seluge. In: IPSN (2008)
15. Trusted Computing Group. Trusted Platform Module (TPM) Specifications, `https://www.trustedcomputinggroup.org/specs/TPM`
16. University of California Berkeley: TinyOS, `http://www.tinyos.net/`