

An Efficient Trusted Computing Base (TCB) for a SCADA System Monitor

A. Velagapalli, M. Ramkumar

Department of CSE, Mississippi State University, MS, USA,

Email: ramkumar@cse.msstate.edu.

Abstract—A fundamental requirement for the ability to monitor a SCADA system is a measure for ensuring that the monitoring process has an accurate picture of the current states of all sensors and actuators in the system. A misrepresentation of the state can be perpetrated either by sending misleading information (for example, by impersonating a sensor) or by preventing sensor measurements from reaching the monitor (for example, jamming). We identify a minimal trusted computing base (TCB) for an untrusted SCADA monitor, and propose a strategy to leverage the TCB efficiently to realize the assurance that “any misrepresentation of the SCADA system state (the states of *all* sensors and actuators) will be identified.” In the proposed approach the TCB is a set of well-defined and simple functions performed by a trusted module. The untrusted monitor is required to periodically offer proof to the trusted module regarding the integrity of dynamic sensor data received from all sensors.

Keywords—SCADA Security, Merkle trees, Authenticated Denial.

I. INTRODUCTION

Supervisory Control and Data Acquisition (SCADA) systems [1] are responsible for automatic control of several critical infrastructures like power grids, chemical plants, refineries, and mass transportation systems. Increased level of automation and self-regulation in SCADA systems have enabling easier management of large distributed systems. Simultaneously, this has also resulted in exposing critical infrastructures to a wide variety of attacks.

In SCADA systems a large number of sensors periodically provide process information to a central location. In practical SCADA systems each sensor - or more specifically, remote terminal units (RTU) connected to sensors, periodically report sensor measurements and actuator states to a master terminal unit (MTU).

At any instant of time, the *state* of a SCADA system is defined by the values measured by various sensors, and the states of different actuators (for example, on/off, open/closed) in the system. An important prerequisite for the ability to monitor a SCADA system is a mechanism ensure that the monitor is provided an accurate picture of the SCADA system state.

A. Attacks on SCADA Systems

Attacks on SCADA systems [1] aimed at misrepresenting the state¹ of the SCADA system can take two broad forms:

¹While there are attacks which do not involve misrepresentation of SCADA system states, in this paper we limit ourselves to attacks in this category.

- 1) impersonation of sensors to provide incorrect state information, and
- 2) preventing a sensor report from reaching the monitor.

The former category of attacks can be addressed by mandating cryptographic authentication of sensor reports. We assume that every sensor shares a secret with the trusted monitor; the reports from sensors are accompanied by a cryptographic authentication token in the form of a hashed message authentication code (MAC) computed using the shared secret.

The second category of attacks could be carried out by an attacker in a variety of ways like i) jamming the channel, ii) cutting a wire, iii) destroying a sensor, etc. In addition to malicious attacks, failure of a sensor report to reach the monitor can also result from non-malicious events like failure of a sensor or some communication equipment.

As a simple strategy to detect such attacks/failures, sensors could be required to send periodic reports; if a sensor has not been heard from for a duration longer than some threshold, then the monitor could raise an alarm. More specifically, assume that all N sensors and the monitor have loosely synchronized clocks, and that every report from each sensor indicates the time before which the next update is due. At any instant of time, the trusted monitor should store N fresh reports (the last report from each of the N sensors). At a time t no record should indicate an expiry time less than t . If any stale record exists, the trusted monitor detects a failure to receive an update, and triggers an alarm. The monitor does not care if the failure is due to an attack or due to a non malicious event.

B. Minimal TCB for a SCADA Monitor

In the approach above the monitor is simply trusted to

- 1) securely store all records,
- 2) periodically verify the expiry time of all records, and
- 3) raise an alarm if a stale record is found.

In most SCADA systems the monitor is a general purpose computer, possibly even connected to the Internet. Such a system may be vulnerable to a large number of attacks. Consequently, it is ill-advised to simply trust the monitor.

For any computing system with a desired set of assurances the trusted computing base (TCB) is “a small amount of software and hardware we rely on (to realize the desired assurances)” and that “we distinguish from a much larger amount that can misbehave without affecting security” [2].

More specifically, as long as the TCB is worthy of trust, the TCB can be amplified to realize the desired assurances. In practice, the TCB can be a set of functions executed inside a well-protected computing boundary. In this paper we investigate a minimal TCB for a SCADA system monitor.

In our approach we assume that the monitor is an untrusted computer \mathbf{U} . The monitor is however required to periodically offer proof to a trusted entity \mathbf{T} , regarding the integrity and freshness of sensor records.

In practice, the entity \mathbf{T} can be a trustworthy hardware module, which can be housed inside \mathbf{U} , or plugged into \mathbf{U} , or merely accessible by \mathbf{U} over an open network. Alternately the module \mathbf{T} could be a software module with some rigorous protection measures.

We refer to the tasks performed inside the module \mathbf{T} as the TCB for the monitor. As long as the functionality of \mathbf{T} cannot be modified, and secrets protected by \mathbf{T} cannot be revealed, the TCB can be amplified to realize the desired assurances regarding the SCADA system monitor. The contribution of this paper is the enumeration of low complexity tasks performed inside the module \mathbf{T} to offer this TCB.

C. Principle of Operation

At a high level, the working of the proposed approach is as follows. Every sensor shares a secret with the module. The reports from sensors are authenticated using message authentication codes (MAC). The module is assumed to be (merely) capable of performing simple logical operations, and cryptographic hash operations.

The dynamic set of N current records form the leaves of a Merkle tree. Only the root of the tree is stored inside the module \mathbf{T} . Employing a Merkle tree permits the module to verify the integrity of any leaf by performing $\log_2 N$ hash operations (and comparing the result with the root stored inside). The N sensor records can thus be stored by the untrusted monitor \mathbf{U} .

The untrusted monitor \mathbf{U} is required to periodically (at least once in every τ seconds) prove to the module that no record is stale (to convince the module to not trigger the alarm for next τ seconds). For large N it may be impractical for the resource limited module to verify each of the N records individually (in every period of length τ) to satisfy itself that no record is stale. The module \mathbf{T} exposes two well-defined interfaces to the untrusted monitor \mathbf{U} - an interface `Update()` for submitting sensor records to the module, and interface `FProof()` to submit proof of freshness of all N records.

A significant novelty in the proposed approach lies in a strategy for indexing records added to the Merkle hash tree to ensure that the module requires to verify only one record to conclude that no record is stale. This indexing strategy is motivated by NSEC [3] - a strategy used in the domain name system (DNS) security protocol, DNSSEC [4], to provide authenticated denial of existence of DNS records.

The rest of this paper is organized as follows. In Section II we provide an overview of Merkle hash tree, and trustworthy computing modules. In Section III we outline the application

setting and provide an overview of the process of updating sensor records, and offering proof of freshness. In Section IV we provide a more in-depth description of the three functions executed by the trustworthy module - `Update()`, `FProof()` and the interface `Init()` for initializing the module. In Section V we discuss some related work, and our conclusions.

II. BACKGROUND

SCADA systems include a centralized monitor/supervisor which receives information from remote terminal units (RTU) bound to process sensors/actuators. The monitor processes such information, and may also display some or all information to human controllers. The supervisory system is also a central point from which different types of commands can be sent to RTUs associated with sensors and actuators. Communications between RTUs master stations running the supervisory system employ a wide variety of communication protocols over wired and wireless channels. Most often, the supervisory system (or the monitor) is software running on a general purpose computer, subject to a wide range of attacks.

In this paper we outline a mechanism for ensuring the integrity of sensor data stored by an untrusted monitor \mathbf{U} , by requiring the monitor to periodically prove some properties about the stored dynamic data to a trustworthy module \mathbf{T} . More specifically, the primary goal is to reduce the complexity of the module \mathbf{T} , without sacrificing the desired assurances. Furthermore, the proposed solution is transparent to the specific types of communication protocols employed both for communications between sensors and the monitor and for communications between the untrusted monitor \mathbf{U} and the trusted module \mathbf{T} .

A. Trustworthy Modules

A trustworthy computing module is trusted because it is believed that

- 1) secrets protected by the module cannot be exposed, and
- 2) the functionality of the module cannot be modified.

To ensure that such properties attributed to a module are justifiable, it is necessary to reduce the complexity of the module to the extent feasible. It is only if the modules possess low complexity, can their immutable functionality be verified consummately. It is only if they consume negligible power, and disseminate negligible heat, can we provide adequate shielding to thwart attacks intending to expose secrets protected by the module.

Practical examples of trustworthy computing modules include the IBM 4758 [5] trustworthy computing module which sports a general purpose processor inside a protected boundary, runs a specialized operating system, and can execute special application code unmolested inside the trusted boundary. Another example is the trustworthy computing group (TCG) [6] specification for trusted platform modules (TPM) [7]. Unlike the IBM modules, TPMs are nonprogrammable. With their fixed functionality, TPMs are capable of storing measures of software (hashes) in its platform configuration registers (PCR), and provide authenticated reports of such measurements to

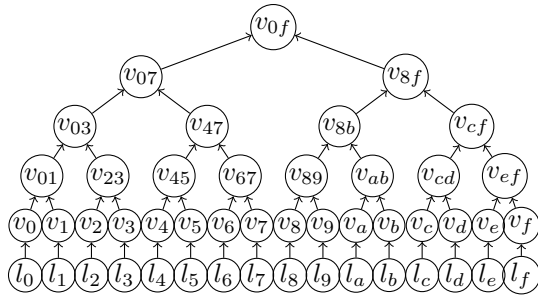


Fig. 1. A Binary Merkle tree with 16 leaves.

third parties (who then trust the measurements to the extent they trust the TPM).

In the approach proposed in this paper, we do not rely on the integrity of the general purpose computer \mathbf{U} (which is fallible to numerous attacks) which receives and stores measurements from sensors - we rely only the integrity of a low complexity module \mathbf{T} to achieve the desired goals: viz., i) accept only authenticated records; ii) securely store all records, iii) periodically verify the expiry time of all records, and iv) raise an alarm if a stale record is found.

Ideally, the module \mathbf{T} should be a dedicated hardware module. Alternately, the functionality of the module can be implemented as a separate software module with some special protections afforded to the module. For example, the current TPM (version 1.2), in conjunction with new instructions supported by Intel and AMD processors (Intel-TXT and AMD-SKINIT), can provide an assurance that a “small piece of application logic (PAL)” [8] will run unmolested on a general purpose platform. More recently, it has been suggested [9] that secure obfuscation techniques can be used as a trust anchor for a SCADA system monitor.

Irrespective of the specific approach used it is indeed desirable to reduce the complexity of the module \mathbf{T} - the hardware module \mathbf{T} , or the PAL \mathbf{T} , to as low an extent as feasible. In the proposed approach, the module functionality can be realized as simple sequences of hash (for example SHA-1) and logical operations, and can thus be realized efficiently in either hardware or software.

B. Merkle Hash Tree

A binary Merkle tree [10] is constructed using a pre-image resistant hash function $h(\cdot)$ (for example, SHA-1). Figure 1 depicts a Merkle tree of height $L = 4$ with $N = 2^L = 16$ leaves. At level 0 (above the leaves) are 2^L nodes - one corresponding to each leaf, obtained by hashing the leaf using $h(\cdot)$. For example, $v_1 = h(l_1)$, $v_6 = h(l_6)$, etc.. At level 1 are 2^{L-1} nodes obtained by hashing of a pair of nodes in level 0. For example, $v_{45} = h(v_4 \parallel v_5)$, where \parallel represents concatenation. Similarly, the 2^{L-i} values in level i are hashes of 2^{L-i} pairs in level $(i-1)$. The lone node at level L , obtained by hashing two nodes in level $L-1$, is the root of the tree.

The root of the tree is a commitment to all N leaves. More specifically, corresponding to any leaf l_i is a node v_i at level 0 which is commitment for l_i , and a set of L

“complementary” nodes bfv_i (one from each level) which are commitments for *all nodes except* l_i . For example, v_4 is a commitment for l_4 , and the set of complementary nodes is $\mathbf{v}_4 = \{v_5, v_{67}, v_{03}, v_{8f}\}$. There now exists a one way function $f(v_i, \mathbf{v}_i) = r$ where r is the root of the tree. The function $f(\cdot)$ is simply a sequence of L hash operations.

A verifier who stores only the root of the tree, and is able to execute the one-way function $f(\cdot)$, can easily verify the integrity of a leaf l_i by checking that $f(v_i, \mathbf{v}_i) = r$. Typically, the prover stores all leaves and internal nodes, and provides the leaf l_i along with L the complementary nodes \mathbf{v}_i as inputs to the verifier. The verifier computes $v_i = h(l_i)$, followed by $f(v_i, \mathbf{v}_i)$. If the output of $f(\cdot)$ is the same as the root stored by the verifier, the verifier is convinced of the integrity of i) the leaf l_i , and ii) the set of complementary nodes \mathbf{v}_i (to the extent the verifier trusts the integrity of the root, and that the hash function $h(\cdot)$ is pre-image resistant).

If it is desired to modify l_i to l'_i , the verifier computes the new the root as $r' = f(v'_i, \mathbf{v}_i)$ where $v'_i = h(l'_i)$. Once the root is modified (to replace the old l_i with l'_i), the verifier will not recognize the old leaf as valid. Thus, the root of the Merkle tree can be seen as a commitment to a dynamic set of N leaves.

For our purposes, a resource limited trustworthy module \mathbf{T} is the verifier which stores the root of a Merkle tree. The prover is an untrusted computer \mathbf{U} which stores all N leaves and the $2N - 1$ internal nodes. The N leaves are records specifying the states of N sensors - each record indicating the latest sensor measurement and the time of expiry of the record. The untrusted \mathbf{U} is expected to maintain the current state of all sensors. However, only the trustworthy module \mathbf{T} is trusted to enforce this requirement.

III. TRUSTWORTHY SCADA MONITOR

We consider a SCADA system with N sensors/actuators and a central monitor. In the rest of this paper we refer to both sensors and actuators as sensors. Each sensor has a unique identity, represented as S_1, S_2, \dots, S_N . Our approach especially targets large SCADA deployments where the number of sensors may be several thousands.

The monitor is a software running on an untrusted general purpose computer \mathbf{U} , which has access to a trustworthy module \mathbf{T} . The module \mathbf{T} has the following limited abilities: i) store a single value (root of a Merkle tree); ii) perform simple sequences of hash and logical operations; and iii) trigger an alarm when a timer fires inside the module.

A sensor S_i is associated is a value d_i , which is the duration of validity of reports from S_i . In general, different sensors may be associated with different durations of validity. For example, a critical sensor S_i which needs to provide updates frequently may have a short validity time like $d_i = 2$ seconds, while S_j , a not-so-critical sensor (or a sensor measuring a slowly varying process) may be associated with a value d_j of the order of a few minutes. All sensors and the module \mathbf{T} have loosely

synchronized clocks².

The sensors send sensed data in the form of “sensor records” which are stored by \mathbf{U} . A fresh report from a sensor should replace the earlier report from the same sensor. Consequently, at any time t , \mathbf{U} is required to store N records - one for each sensor. Each record is a leaf of a Merkle tree; the root of the tree is stored inside \mathbf{T} .

Records sent from sensors are authenticated for verification by \mathbf{T} . The untrusted computer \mathbf{U} is expected to submit records to \mathbf{T} on arrival. If satisfied with the integrity of the record, the module \mathbf{T} accepts the new record and updates its root. Every new record from every sensor will lead to a modification of the root.

Associated with the module \mathbf{T} is a timer of duration τ . Expiry of the timer will trigger an alarm. To prevent the alarm from being triggered, verifiable proof of integrity and freshness of all N records should be submitted to the module. If the proof is satisfactory, the module resets the timer to zero, and restarts the timer. Thus, *at least once* in every interval of time τ the module \mathbf{T} requires to be “convinced” of the freshness of all N records.

A. Sensor Data

To bootstrap the system, a trusted entity is assumed to provide some values to the module \mathbf{T} , to every sensor, and the untrusted monitor \mathbf{U} . Specifically,

- 1) \mathbf{T} is provided a secret K , and a value r_0 as the *initial* root of a Merkle tree.
- 2) the untrusted computer \mathbf{U} is provided a set of N *initial* records (one corresponding to each sensor). The N records are leaves of a Merkle tree with root r_0 (which was provided to \mathbf{T});
- 3) each sensor is provided a secret; the secret K_i provided to sensor S_i is

$$K_i = h(K \parallel S_i \parallel d_i). \quad (1)$$

The sensor data from S_i is a record of the form

$$\mathbf{R}_i^{\mathbf{T}} = S_i \parallel \xi_i \parallel t_i \parallel \mu_i \quad (2)$$

where ξ_i is the measured value, and t_i is the current time (as perceived by the sensor S_i), and μ_i is a MAC computed as

$$\mu_i = h(\xi_i \parallel t_i \parallel K_i). \quad (3)$$

The sensor data is received by \mathbf{U} , which is expected to submit the record to \mathbf{T} . More specifically, module \mathbf{T} is provided the record \mathbf{R}_i along with the value d_i . The module \mathbf{T} (which has access to the secret K) computes $K_i = h(K \parallel S_i \parallel d_i)$, and verifies the integrity of the MAC μ_i .

If the MAC is valid, the record $\mathbf{R}_i^{\mathbf{T}}$ is interpreted by \mathbf{T} to be valid till time $\tau_i = t_i + d_i$. More specifically, the record

²The extent of clock synchronization required is related to the least d_i value d_{min} - the clock drift should be substantially lower the d_{min} . We assume some mechanism in place for the sensors to periodically synchronize their clocks to \mathbf{T} 's clock.

$\mathbf{R}_i^{\mathbf{T}}$ from S_i is added to the Merkle tree as a record \mathbf{R}_i of the form

$$\mathbf{R}_i = S_i \parallel \xi_i \parallel \tau_i \parallel a_i, \quad (4)$$

where a_i is some auxiliary information required to verify the freshness of records, as discussed in the following section.

B. Auxiliary Sensor Data

At any time t , the set of N records stored by \mathbf{U} is of the form

$$\begin{array}{cccc} S_1 & \xi_1 & \tau_1 & a_1 \\ S_2 & \xi_2 & \tau_2 & a_2 \\ & \vdots & & \\ S_N & \xi_N & \tau_N & a_N \end{array} \quad (5)$$

where ξ_i is the reported value for sensor S_i , and τ_i is the time till which the sensor record is valid. The last field - the “auxiliary” value - a_i in a record for sensor S_i is obtained by sorting the values $\tau_1, \tau_2, \dots, \tau_N$ of all N records in an ascending order. More specifically, the value that follows τ_i is a_i . If τ_i happens to be the highest value (last value in the sorted order) then the value that follows is the first value - or a_i is the first (smallest) value in the sorted list.

Shown below is an example depicting τ_i and a_i values for a set of $N = 8$ sensor records.

$$\begin{array}{ccc} \xi & \tau & a \\ S_1 & 5 & 1002 & 1008 \\ S_2 & 6.78 & 845 & 848 \\ S_3 & 0 & 850 & 1002 \\ S_4 & 5 & 840 & 842 \\ S_5 & 4.44 & 848 & 850 \\ S_6 & 0 & 1008 & 835 \\ S_7 & 0.76 & 835 & 840 \\ S_8 & 0 & 842 & 845 \end{array} \quad (6)$$

The least τ value (835 for S_7) is accompanied by $a_7 = 840 = \tau_4$ - the next higher value. Similarly, $a_4 = 842 = \tau_8$, and so on. Corresponding to the record S_6 with the highest $\tau_6 = 1008$ is $a_6 = 835 = \tau_7$, the least τ value.

The reason that the sensor data is stored with an auxiliary field is to be able to easily offer proof to the module \mathbf{T} that “no record is stale.” Recall that at least once in every τ units of time \mathbf{U} needs convince the module \mathbf{T} that “all N sensor records are fresh.” Without the additional value a_i , \mathbf{U} has to submit each of the N records to the module \mathbf{T} which will then need to verify integrity of each record individually, and check that no record has a value τ smaller than the current time. Such an approach is obviously impractical for large N .

Now, with the auxiliary value a_i , \mathbf{U} has to provide only one sensor record - the record which includes a wrapped-around value (τ_l, a_l) with $a_l < \tau_l$ (in the specific example above, the record for S_6 with values (1008, 835)). This record indicates that $a_l = 835$ is the earliest time of expiry. Thus, as long as $a_l = 835$ is greater than the current time t , the module is convinced that all records are fresh. Thus, once in every τ

units of time, the untrusted computer has to submit just *one* record to the module **T** to suppress the alarm.

That a leaf l_i indicates a value (τ_i, a_i) is proof that “no leaf exists in the tree with a value of τ covered by (τ_i, a_i) ”. Such an approach is used in NSEC [3] records in DNSSEC (DNS security) [4] for providing authenticated denial of queried records. An NSEC record of the form `(abc.example.com, add.example.com)` proves that no record pertaining to a name `ac.example.com` exists.

C. Updating a Sensor Record

As stored sensor records are updated (with every received sensor record), it is necessary to ensure the consistency of the τ_i, a_i values in all N records at all times. To achieve this, updating a sensor typically requires *three* records to be modified. More specifically, apart from the record for which an update is received, the a_i values in two *other* records may need to be modified. In some (non-typical) cases, the a_i value in one other record will need to be modified.

For example, consider an update for S_5 with a new validity time 851 (to replace the old validity time 848). As a result, the auxiliary value in records S_2 and S_3 need to be modified. Specifically,

- 1) the value a_3 needs to be changed from 1002 to 851
- 2) the value a_2 needs to be changed from 848 to 850

As another example (of a non typical case), if an update for S_5 causes the validity time to be modified from 848 to 849, then a_2 should be modified from 848 to 849.

More generally, to update a record for an index u with current values (τ_u, a_u) to a new expiry time τ'_u , the algorithm is as follows:

- 1) determine the record index p such that $a_p = \tau_u$ (this record index p “points” to the record indexed u to be updated).
- 2) determine the record index c such that (τ_c, a_c) covers τ'_u . More specifically,
 - a) if $\tau_c < a_c$ then (τ_c, a_c) covers τ'_u if $\tau_c \leq \tau'_u \leq a_c$;
 - b) if $\tau_c > a_c$, then (τ_c, a_c) covers τ'_u if $\tau'_u > \tau_c$, or if $\tau'_u < a_c$;
 - c) if $\tau_c = a_c$, then (τ_c, a_c) covers τ'_u if $\tau'_u = \tau_c = a_c$.
- 3) If $u == c$, or in other words, if (τ_u, a_u) itself covers the new value τ'_u , then set $a_p = \tau'_u$; the value a_u remains unchanged in the updated record;
- 4) If $u \neq c$, then set i) $a_p = a_u$; ii) $a_u = a_c$; and iii) $a_c = \tau'_u$;

For the example where S_5 was updated to a new validity time 851, the indices u, p and c are respectively $u = 5, p = 2$ and $c = 3$. For the second example where S_5 was updated to a new validity time 849, the indices are $u = c == 5$, and $p = 2$.

IV. MODULE **T** INTERFACES

The module **T** exposes three fixed interfaces. The interface `Init()` is used to initialize the module - to provide the initial root value. The interface `Update()` is used by the untrusted

monitor **U** to submit new authenticated reports from sensors to the module **T** to cause the module to update the root of the tree. The interface `FProof()` should be periodically called by **U** (at least once every τ seconds) to inhibit the module from sounding the alarm.

In this section we provide a more in-depth description of the three interfaces.

A. Interface `Init()`

The function `Init()` can be called only by the trusted authority to provide i) the value r_0 (the initial root), and ii) a value τ (which is the timer duration). In response, the module **T** i) sets its clock counter to zero; ii) stores r_0 and τ in an internal register, and iii) starts the τ -timer. From this point onwards, if the timer fires, an alarm will result.

The trusted authority provides a sets of initial records - one for each sensor, to **U**. In the record for S_i (with values ξ_i, τ_i, a_i), $\tau_i = d_i$. As explained earlier the value a_i is the next higher value of τ . At the time the module **T** is initialized, all sensors are also instructed to set their clock to zero.

B. Interface `FProof()`

The monitor **U** uses this interface to submit proof that no record is stale. Specifically, the values provided as input are i) a leaf l_i with (τ_i, a_i) where $a_i < \tau_i$, and ii) a set of complementary values \mathbf{v}_i such that $f(v_i, \mathbf{v}_i) = r$. The module first verifies the integrity of the leaf l_i ; verifies that $a_i < \tau_i$ (which occurs only for one leaf), and that $a_i > t$ where t is the current time. If so, the module **T** resets its τ -timer to zero.

C. Interface `Update()`

Typically, a Merkle tree is used in applications where each leaf needs to be changed independently of other leaves. However, as discussed in Section III-C, to update a leaf l_u , two other leaves may have to be modified.

To update the record for a sensor S_u the prover provides the following inputs:

- 1) the current leaf l_u for S_u (which indicates values ξ_u, τ_u and a_u), and the value d_u ;
- 2) a new record for sensor S_u (authenticated by S_u through MAC μ_u), indicating time of creation as t_u . This record will be verified by the module before it replaces the old leaf l_u . Recall that the new leaf l'_u will indicate expiry time $\tau'_u = t_u + d_u$.
- 3) a current leaf l_p (for sensor S_p) with $a_p = \tau_u$,
- 4) a current leaf l_c , with the pair (τ_c, a_c) which covers $\tau'_u = t_u + d_u$.
- 5) a set of “complementary” hashes $\mathbf{v}_{u,p,c}$ to permit **T** verify the validity of the three current records against the root.

For now we shall assume that the module **T** is capable of executing a function of the form $f_3(v_u, v_p, v_c, \mathbf{v}_{u,p,c}) = r$ to verify the validity of the leaves l_u, l_p and l_c , and the validity of the complementary hashes $\mathbf{v}_{i,j,s}$ themselves. (the specific algorithm for $f_3()$ is discussed in the next section).

The module **T** processes the inputs as follows:

- 1) Verify MAC μ and compute $\tau'_u = t_u + d_u$;
- 2) Verify $f_3(v_u, v_p, v_c, \mathbf{v}_{u,p,c}) = r$;
- 3) If $v_u == v_c$, create
 - a) a new leaf l'_p by replacing value a_p in leaf l_p with τ'_u ; and
 - b) a new leaf l'_u by replacing values ξ_u and τ_u in l_u with values ξ'_u and τ'_u .
- 4) If $v_u \neq v_c$, create
 - a) a new leaf l'_u by replacing ξ_u with ξ'_u , τ_u with τ'_u , and a_u with a_c ;
 - b) a new leaf l'_p by replacing a_p with τ'_u ; and
 - c) a new leaf l'_c by replacing a_c with τ'_u .
- 5) Compute $v'_u = h(l'_u)$, $v'_p = h(l'_p)$, $v'_c = h(l'_c)$;
- 6) Compute the new root as $r' = f_3(v'_u, v'_p, v'_c, \mathbf{v}_{u,p,c})$.

D. Merkle Tree Primitives

At the lowest level, every operation for creating, verifying and updating a Merkle hash tree is a hash function $h()$. At the next level is an elementary operation to traverse one level of the tree. Specifically, starting from any internal node x , to traverse the tree one-level up, we require an ‘‘instruction’’ of the form (y, b) , where y is a node adjacent to x (at the same level as x) and b is a one-bit value. When the instruction (y, b) is applied to a value x , the result is $h(x \parallel y)$ if $b = 0$, or $h(y \parallel x)$ if $b = 1$. For example, an instruction $(v_{67}, 0)$ maps v_{45} to $v_{47} = h(v_{45} \parallel v_{67})$; to map v_{67} to v_{47} the instruction is $(v_{45}, 1)$.

We shall use the notation $h_e(x, (y, b))$ to represent this elementary Merkle-tree primitive of traversing one-level up the tree, which can be expressed algorithmically as

$$h_e(x, (y, b)) = \begin{cases} h(x \parallel y) & \text{if } b = 0 \\ h(y \parallel x) & \text{if } b = 1 \end{cases} \quad (7)$$

This elementary instruction is typically applied repeatedly. The process denoted by $h_{ve}()$ takes a *sequence* of instructions $\mathbf{I} = \{(y_1, b_1) \cdots (y_n, b_n)\}$ as input, and can be described algorithmically as

```

 $h_{ve}(x_0, \{(y_1, b_1) \cdots (y_n, b_n)\}) \{$ 
  FOR  $i = 1 \cdots n$ 
     $x_i = h_e(x_{i-1}, (y_i, b_i));$ 
  RETURN  $x_i;$ 
 $\}$ 

```

In Section II-B we loosely represented the process of mapping a leaf l_4 to the root r as a function $r = f(v_4, \mathbf{v}_4)$ where $v_4 = h(l_4)$ and $\mathbf{v} = \{v_5, v_{67}, v_{03}, v_{8f}\}$. A more accurate description is as $r = h_{ve}(v_4, \mathbf{I})$, where $\mathbf{I} = \{(v_5, 0), (v_{67}, 0), (v_{03}, 1), (v_{8f}, 0)\}$.

Thus the inputs to the interface FProof() are

- 1) a leaf l_i with (τ_i, a_i) where $a_i < \tau_i$; and
- 2) an instruction vector \mathbf{I}_i such that $r = h_{ve}(v_i, \mathbf{I})$ (where $v_i = h(l_i)$).

1) *Common Parent of Two Nodes:* Any two internal nodes - say x_l and x_r (where x_l cannot be to the right of x_r) have a common parent x_p . For example, v_{47} is the common parent of $x_l = v_4$ and $x_r = v_6$; v_{8f} is the common parent of v_9 and v_{ef} . If $x_l = x_r$ (if both actually refer to the same internal node) then the parent is also the same node $x_p = x_l = x_r$.

To map two internal nodes, say $x_l = v_1$ and $x_r = v_{67}$, to a common parent $x_p = v_{07}$, two sets of instruction vectors are required. In this case, the instruction vector $\{(v_0, 1), (v_{23}, 0)\}$ maps v_0 to the v_{03} (which is the left child of $x_p = v_{07}$). The instruction vector $(v_{45}, 1)$ maps v_{67} to v_{47} (the right child of x_p). The two children are then combined to yield the parent. In this paper we use the notation $h_{cp}()$ to represent the process of reaching a common parent of two nodes. Algorithmically, $h_{cp}()$ can be described as follows:

```

 $h_{cp}(x_l, \mathbf{I}_l, x_r, \mathbf{I}_r) \{$ 
  IF  $(x_l \neq x_r)$ 
     $\xi_l = h_{ve}(x_l, \mathbf{I}_l);$ 
     $\xi_r = h_{ve}(x_r, \mathbf{I}_r);$ 
    RETURN  $h(\xi_l \parallel \xi_r);$ 
  ELSE  $\parallel x_l == x_r$ 
    RETURN  $x_l;$  //parent of  $x_l$  and  $x_l$  is  $x_l$ 
 $\}$ 

```

2) *Common Parent of Three Nodes:* To map three nodes x_l, x_m, x_r (where x_m is to the right of x_l and to the left of x_r) to a common parent x_p we first identify the common parent of (x_l, x_m) and (x_m, x_r) (say x_{lm} and x_{mr} respectively). If x_{lm} is at a lower level we then proceed to find the common parent of x_{lm} and x_r . On the other hand, if x_{mr} is at a lower level (compared to x_{lm}) we proceed to determine the common parent of x_l and x_{mr} .

For example, to map $x_l = v_0, x_m = v_2$ and $x_r = v_7$ to the common parent $x_p = v_{07}$ we first map v_0 and v_2 to the common parent $x'_{lp} = v_{03}$ and then map v_{03} and v_7 to the parent $x_p = v_{07}$. To map v_2, v_6 and v_7 to the common parent v_{07} we first map $x_m = v_6$ and $x_r = v_7$ to $x'_{rp} = v_{67}$, and then map v_2 and x'_{lp} to the parent $x_p = v_{07}$.

The instructions required to map three nodes x_l, x_m, x_r to a common parent x_p include i) a one bit instruction b (if $b = 0$ then x'_{lp} is the common parent of x_l and x_m ; if $b = 1$ then x'_{rp} is the common parent of x_m and x_r), and four instruction vectors $\mathbf{I}_l, \mathbf{I}_m, \mathbf{I}_r$ and $\mathbf{I}_{p'}$. The algorithm $x_p = h_{3cp}()$ for determining the common parent of three nodes is then

```

 $p = h_{3cp}(x_l, \mathbf{I}_l, x_m, \mathbf{I}_m, x_r, \mathbf{I}_r, b, \mathbf{I}_{p'}) \{$ 
  IF  $(b == 0)$ 
     $x'_{lp} = h_{cp}(x_l, \mathbf{I}_l, x_m, \mathbf{I}_m);$ 
    RETURN  $h_{cp}(x'_{lp}, \mathbf{I}_{p'}, x_r, \mathbf{I}_r);$ 
  ELSE
     $x'_{rp} = h_{cp}(x_m, \mathbf{I}_m, x_r, \mathbf{I}_r);$ 
    RETURN  $h_{cp}(x_l, \mathbf{I}_l, x'_{rp}, \mathbf{I}_{p'});$ 
 $\}$ 

```

3) *Implementation of Function $f_3()$:* Finally, three nodes at level 0, say v_l, v_m, v_r can be mapped to the root by first mapping them to the common parent by using $p = h_{3cp}(v_l, \mathbf{I}_l, v_m, \mathbf{I}_m, v_r, \mathbf{I}_r, b, \mathbf{I}_{p'})$, and an additional instruction \mathbf{I}_p which maps the common parent to the root (if the common parent is not already the root). Such a function will take the form

```

 $r = h_{3lr}(v_l, v_m, v_r, \mathbf{I}_l, \mathbf{I}_m, \mathbf{I}_r, b, \mathbf{I}_{p'}, \mathbf{I}_p) \{$ 
   $p = h_{3cp}(v_l, \mathbf{I}_l, v_m, \mathbf{I}_m, v_r, \mathbf{I}_r, b, \mathbf{I}_{p'})$ 
  RETURN  $h_{ve}(p, \mathbf{I}_p)$ 
 $\}$ 

```

Recall that the interface Update() discussed in Section IV-C assumed a function of the form $f_3(v_u, v_p, v_c, \mathbf{v}_{u,p,c})$ to map three nodes (v_u, v_p, v_c) at level 0 to the root. The function is actually realized as $h_{3lr}()$. In translating from $f_3()$ to $h_{3lr}()$, the values $\mathbf{v}_{u,p,c}$ in Section IV-C actually represent

the instruction vectors $\mathbf{I}_l, \mathbf{I}_m, \mathbf{I}_r, b, \mathbf{I}_{p'}, \mathbf{I}_p$, and an additional parameter ν (the purpose of which will be explained soon).

In general, the values (v_u, v_p, v_c) input to $f_3()$ is a *permutation* of the values (v_l, v_m, v_r) input to the function $h_{3lr}()$. Note that the function $h_{3lr}()$ expects values (v_l, v_m, v_r) which are pre-constrained to have specific relative positions (among the three, v_l should be the left-most and v_r should be the right-most). Thus to employ the function $h_{3lr}()$ to compute $r, \mathbf{v}_{u,c,p}$ should also specify a value ν which can take values 1 to 6 - each corresponding to a specific permutation of (v_u, v_p, v_c) to obtain (v_l, v_m, v_r) required for $h_{3lr}()$.

Recall that for an example where S_5 was updated to a new validity time 851, the indices u, c and p were respectively $u = 5, p = 2$ and $c = 3$. Thus in this case $x_l = v_p = v_2, x_m = v_c = v_3$ and $x_r = v_u = v_5$. For the second example where S_5 was updated to a new validity time 849, the indices were $u = c = 5$, and $p = 2$. Thus, $x_l = v_p = v_2$, and $x_m = x_r = v_u = v_c = v_5$.

V. RELATED WORK AND CONCLUSIONS

The ability to monitor SCADA systems controlling critical infrastructure is an important requirement for the security of any State. As general purpose computers cannot be trusted to perform this task it is necessary to identify a minimal amount of trusted hardware/software to enforce this requirement.

Mandating that only trusted platforms be used for running the monitoring software can alleviate some concerns. In the trusted computing group (TCG) specification for trusted platforms, trusted platform modules (TPM) are used to attest measures of loaded software, thereby permitting third parties to verify that only authorized software has gained control of a platform. However, practical deployments of TCG trusted platforms have been hindered by several factors like i) several attacks that have been identified to violate the envisioned security goals of the TCG architecture [11]; and that ii) thorough review necessary to pre-certify software (BIOS, boot-loader, operating system and application software) is far from practical. Chavez [9] suggests using secure obfuscation techniques to run protected code which can then be used a trust anchor for a SCADA monitor. However, [9] does not describe the exact nature of the tasks performed by the trust anchor.

The ongoing IEEE effort to standardize SCADA cryptographic modules (SCM) [12] that comply with Serial SCADA Protocol Protection (SSPP) protocol, intended to permit SCMs to be retrofitted in existing SCADA deployments, is a good first step towards reducing the scope of attacks involving impersonation of sensors/actuators. However, such an approach does not address attacks that prevent sensor data from reaching the monitor.

In the proposed solution a trustworthy module \mathbf{T} requires to perform only simple sequences of hash and logical operations to ensure that any misrepresentation of the SCADA system state will be detected by the trustworthy module. If a high level of security is desired the module \mathbf{T} can be a dedicated hardware module. As any one-way communication channel

is sufficient for the interactions between \mathbf{U} and \mathbf{T} (in all "interactions" \mathbf{U} sends some values - inputs for the interfaces Update() and FProof() - to \mathbf{T}) it is also possible to house the module \mathbf{T} in a more secure (possibly remote) location if necessary.

The primary advantage of a Merkle trees is a single cumulative verification point for all its data records, which makes it easier to store that one value in a secure location (instead of storing all the data records). This useful feature has been taken advantage of in various application scenarios. The secure co-processor AEGIS [13] utilizes this feature to expand trust in the root hash of the tree which is stored inside its secure memory to provide a trusted boundary for the values stored outside. In [14] a Merkle tree is used to leverage one trusted monotonic counter to realize a large number of virtual monotonic counters.

One of the main novelty in the proposed approach is the use of a special indexing strategy for leaves of a Merkle tree. A simple Merkle tree can *not* be used to provide a reliable response to a query like "what is the minimal (or maximal) value among all leaves," without checking every leaf. Unlike traditional applications of Merkle trees where each leaf is processed independently, our indexed tree requires a plurality of leaves to be verified/updated simultaneously due to the interdependency between leaves.

Ensuring detection of misrepresentations of sensor states is however just one step towards securing SCADA systems. Once it is ensured that the untrusted monitor cannot misrepresent states $\{\xi_1 \cdots \xi_N\}$ of any sensor/actuator, a mechanism is required to verify that the state $\{\xi_1 \cdots \xi_N\}$ is a "valid" one. One of our ongoing work is the identification of simple TCB functions performed by a trusted module to verify proof of validity of the state.

REFERENCES

- [1] J. Stamp, J. Dillinger, W. Young, J. DePoy, "Common Vulnerabilities in Critical Infrastructure Control Systems," Sandia National Laboratories report SAND2003-1772C, Albuquerque, New Mexico (2003).
- [2] B. Lampson, M. Abadi, M. Burrows, E. Wobber, "Authentication in Distributed Systems: Theory and Practice," ACM Transactions on Computer Systems, 1992.
- [3] S. Weiler, J. Ihen, "RFC 4470: Minimally Covering NSEC Records and DNSSEC On-line Signing," April 2006.
- [4] R. Arends, R. Austein, M. Larson, D. Massey, S. Rose "RFC 4033: DNS Security: Introduction and Requirements," March 2005.
- [5] S.W. Smith, S. Weingart, "Building a High-Performance Programmable Secure Coprocessor," IBM Technical Report RC21102, Feb 1998.
- [6] TCG Specification: Architecture Overview, Specification Revision 1.4, 2nd August 2007.
- [7] Trusted Computing Group, "Trusted platform module main specification," Version 1.2, Revision 94, Mar. 2006.
- [8] J. M. McCune, B. Parno, A. Perrig, A. Seshadri, "How Low Can You Go? Recommendations for Hardware-Supported Minimal TCB Code Execution," ASPLOS08, March 15, 2008, Seattle, Washington, USA.
- [9] A. R. Chavez, "Protecting Process Control Systems against Lifecycle Attacks Using Trust Anchors," DHS Workshop on Future Directions in Cyber-physical Systems Security, July 22-24, 2009.
- [10] R.C. Merkle "Protocols for Public Key Cryptosystems," In Proceedings of the 1980 IEEE Symposium on Security and Privacy, 1980.
- [11] E. Sparks, "A Security Assessment of Trusted Platform Modules," Computer Science Technical Report TR2007-597, Dartmouth College, 2007.

- [12] IEEE Trial Use Standard for SCADA Serial Link Cryptographic Modules and Protocol, 2008, P1711 Draft 3, 2008-08-16.
- [13] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," In Proceedings of the 17th International Conference on Supercomputing, New-York, June 2003.
- [14] L. F. G. Sarmanta, M van Dijk, C W. ODonnell, J. Rhodes, S. Devadas, "Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS," In Proceedings of the 1st ACM CCS Workshop on Scalable Trusted Computing, 2006.