Reliable Assurance Protocols for Information Systems

Mahalingam Ramkumar
Computer Science and Engineering
Mississippi State University
Mississippi State, MS
Email: ramkumar@cse.msstate.edu

Somya D. Mohanty
Social Sciences Research Center
Mississippi State University
Mississippi State, MS
Email: somya.mohanty@ssrc.msstate.edu

***Abstract*—The assurances provided by an *assurance protocol* for any information system (IS), extend only as much as the integrity of the assurance protocol itself. The integrity of the assurance protocol is negatively influenced by a) the complexity of the assurance protocol, and b) the complexity of the platform on which the assurance protocol is executed. This paper outlines a holistic Mirror Network (MN) framework for assuring information systems that seeks to minimize both complexities. The MN framework is illustrated using a generic cloud file storage system as an example IS.**

Keywords: ***Clark-Wilson Model, System Integrity, Ordered Merkle Tree, Cloud Storage***

I. Introduction

Information systems (IS) are composed of a variety of hardware and software components that create, exchange, process, and dispose data. From a broad perspective, assuring the operation of an IS is a process involving *verification of self-consistency* of all critical internal states of the IS. From this perspective, the assurance mechanism (or the *assurance protocol*) itself can be seen as software that

1) verifies self-consistency of IS data, and
2) reports consistency/inconsistency to entities that interact with the IS.

For example, some of the simple self-consistency checks that will need to be performed by an assurance software for an accounting system include a) that available balance in an account is incremented by the amount deposited, or decremented by the amount withdrawn; b) that transfer of an amount a from an account A to account B results in increase in account B balance by a, and reduction in account A balance by $a + x$ (where x is a service charge), etc.

The assurances offered by the assurance software are, at best, only as good as the integrity of the assurance software itself. In general, the higher the complexity of any hardware/software component, the higher the possibility of presence of undesired (malicious or accidental) functionality [1]. Consequently, it is important to minimize both a) the complexity of the assurance software, and b) the complexity of the platform in which the assurance software is executed.

The motivation for the proposed approach to assure ISes stems from the fact that the assurance software for an IS can be *substantially simpler* than the IS software; consequently, the assurance software can be easily executed on a dedicated *high-integrity-low-complexity platform* (Figure 1), that is *completely isolated* from the actual IS. In the proposed approach, the platform for execution of the assurance software is constrained to be a homogeneous network, composed of a single type of *low complexity building block*. The mirror network (MN) model outlined in this paper involves assembling any number of such building blocks — hereinafter referred to as *MN modules* $\mathbf{T}$, into a network that a) mirrors critical IS states; and b) executes IS-specific assurance protocols for checking/reporting self-consistency of IS states.

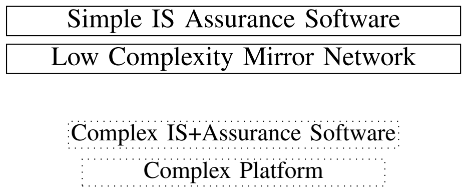
Simple IS Assurance Software
Low Complexity Mirror Network
Complex IS+Assurance Software
Complex Platform

Figure 1. MN Model (top) vs Conventional Model (bottom)

The main contributions of this paper are a) an overview of simple *generic* functional components of MN building blocks, that permit them to

1) assemble themselves into an MN, and
2) jointly execute the assurance protocol

for *any* IS, and b) illustration of the process of assurance software design, using a generic cloud file storage service as an example.

The rest of this paper is organized as follows. Section II provides a broad overview of the MN model and its relationship to the Clark-Wilson (CW) system integrity model. Specifically, while the CW model is applied directly to the IS to be secured, the MN model can be seen as a variant of the CW model, *applied to the assurance protocol* for the IS. This feature has has two important advantages. Firstly, the IS assurance software for an IS can be substantially simpler than the IS. Secondly, the assurance software for different ISes tend to be more similar than the ISes themselves — making it possible to reuse a small number of simple functional components to realize assurance software for different ISes.

Application of the MN model to an IS results in a simple *MN specification*, which *is* the assurance software for the IS, intended to be executed on a special platform — a mirror network. Section II-B outlines the mechanism for MN deployment. Section II-C reviews some simple built-in functional components in MN modules which can be leveraged to deploy MNs, and permit them to jointly execute the assurance software. Section III describes various steps involved in designing the MN specification (the assurance software) for an example IS — a cloud file storage system. Conclusions and a brief comparison between the MN approach and an alternate approach in the literature [2] (also based on a specification of low complexity modules) are offered in Section IV.

II. MN Model for Information Systems

The Clark-Wilson (CW) [3] model for system integrity is characterized by constrained data items (CDI), unconstrained data items (UDI), transformation procedures (TP), integrity verification procedures (IVP), and CW-tuples, where

1) **CDIs** are unambiguously labeled system states whose integrity needs to be assured;
2) **IVPs** determine if the current state of all CDIs represent a valid IS state;
3) Only "well-formed" **TPs** can *modify* or *create* CDIs;
4) **UDIs** can not be constrained as they are external inputs to the system; they are the primary triggers for creation / modification of CDIs.
5) **CW tuples** of the form (user, TP, CDIs/UDIs) specify which user process is allowed to execute which TP, and which CDIs are affected by the TP.

A TP is well-formed if it is guaranteed to always take the system from one correct state to another correct state. If at any time, the correctness of the IS state is demonstrated by an IVP, and thereafter, if only well-formed TPs are used to modify/create CDIs, it follows from induction, that the system is guaranteed to always remain in a correct state. In the CW model, the correctness of IVPs, TPs and CW-tuples are assumed to be certified by a "security officer."

A. MN Model

In the (ρ, μ, ν) MN model for an IS S with ρ CDI database types, μ message types and ν event types

1) One-way functions of crucial IS S states are grouped together into **CDI databases** of ρ different types.
2) *Events* (of ν types) trigger a) modifications to the CDI databases, and/or b) creation of **MN messages** (any of μ types).
3) Events can be external or internal. External events are UDIs. Internal events are triggered by **MN messages**, created by an external or internal event.
4) Each event type is associated with a **TP**, specifying a list of pre-conditions (for execution of the TP) and post-conditions (following execution of the TP).

To summarize, the MN model for an IS S is simply a specification of ρ types of CDI databases, ν event-types/TPs and μ types of MN messages. The designer of the MN has complete freedom in choosing convenient ρ, μ and ν, depending on the nature of the IS (in the example MN for a cloud file storage system described in a later section we choose $\rho = 3, \mu = 4$ and $\nu = 17$). The MN specification for an IS S is represented as a *static MN-rules database*, which is **the static assurance "software"** for IS S. More specifically, "execution of the assurance software" is simply execution of the ν TPs specified in the MN-rules database. A static cryptographic commitment to the MN-rules database, say S', doubles as the *identity* of the MN (the platform) deployed to execute the assurance software for the IS S.

B. Execution of Assurance Software

The MN S' (deployed for assuring IS S) is a dynamic network, composed of any number of MN modules (which become *members* of the MN). For an MN with ρ different CDI database types, members with ρ different *roles* will exist. The total number of members (MN modules) $d(t) = \sum_{i=1}^{\rho} n_i$ (or n_i members with role i) is dynamic (need *not* be specified *apriori* in the MN rules database). All MN modules possess identical functionality; the differences between modules are merely their unique identities and secrets. Within the context of MN S', each module is assigned a unique role based member identity, depending on the CDI database type maintained by the member.

Apart from the $d = \sum_{i=1}^{\rho} n_i$ members (that track CDI databases), every MN includes a special module regarded as the *creator* of the MN. The MN creator is responsible for inducting other modules into the MN as members. Unlike the $d = \sum_{i=1}^{\rho} n_i$ modules that track dynamic CDI databases, the MN creator module tracks a dynamic MN membership database. For example, if MN modules with identities $\Pi_1 \cdots \Pi_d$ have been inducted into the MN S', and assigned role based identities $(X_1 \cdots X_d)$ respectively, the membership database maintained by the MN creator module will have d records of the form (X_i, Π_i). The role-based member identities like X_i explicitly indicate (using reserved bits) the role of the member.

From the perspective of a MN member X with role i, "tracking" a CDI database of type i involves a) unambiguously identifying the TP to be executed in response to an event, b) verifying pre-conditions, and c) imposing post-conditions. Pre-conditions can be i) existence/nonexistence of specific records in X's CDI database; and/or ii) receipt of a MN message. Post-conditions can be i) updates to specific records in its CDI database; and/or ii) creation of an MN message.

During regular operation of the IS S, external events (UDIs) are conveyed to the MN. This is the *only* link between the IS S and the MN S'. A member X in the MN, triggered by an event, executes a TP, which can result in modification to one or more CDI database records of X, and/or creation of a MN message from member X to another member Y. The MN message so created, triggers execution of a TP by Y, which can trigger modification to CDI records of Y and/or creation of a message addressed to a MN member Z, and so on.

C. Generic MN Module Functions

To reduce the complexity of the platform (the MN), MN modules are *deliberately* constrained to be able to perform only simple sequences of logical and hash operations that demand only modest and constant memory size for execution. Fortunately, the versatility of cryptographic hash functions renders them more than adequate for

1) realizing simple security protocols for tracking the integrity of dynamic databases, and

2) facilitating authentication and privacy of a) MN messages between MN modules, b) UDIs from external entities to MN modules; and c) state reports from MN modules to external entities.

In other words, simple generic (IS-independent) protocols built-in into MN modules provide the foundation for richer protocols necessary for deployment of MNs, *and* execution of IS-specific TPs.

1) Index Ordered Merkle Tree: From the perspective of MN modules, any database is seen as a collection of (index,value) tuples (or records). Protocols for maintaining an index ordered Merkle tree (IOMT) [4], [5]-[7] permit resource limited modules that store only a single hash — the *root* of the IOMT — to perform reliable database operations for reading/ updating/ inserting/ deleting uniquely indexed records/tuples in a *virtually* stored database. In other words, the actual database of records can be stored in any convenient (and possibly untrusted) location – for example, by the untrusted IS. For a virtually stored database with N records, each basic database operation will only require $\mathbb{O}(\log_2 N)$ hash evaluations by the module (for example, 40 hashes for a database with a trillion records). IOMTs can also be used to represent nested tuples — where the value v in tuple (a, v) can itself be the root of an IOMT.

In databases represented using an IOMT, a record of the form $(idx, val = 0)$ is a *place-holder*, indicating "absence of information" regarding index idx. The main difference between an IOMT and the better known "plain" Merkle hash tree [8] is that the IOMT includes protocols for *insertion/deletion* of place holders to guarantee uniqueness of indexes. Protocols for updating/reading records using an IOMT are, however, identical to that of a "plain" erkle tree.

Simple built-in capability to execute IOMT protocols confer MN modules (which store only a single hash) with the ability to a) verify pre-conditions like existence of a record (f, v), non-existence of a record for index f (or equivalently, existence of place-holder $(f, 0)$), in the virtually stored CDI database and b) modify the IOMT root stored inside in accordance with modifications made to one or more virtually stored CDI tuples, as demanded by post-conditions of a TP.

Specifically, MN modules use their ability to execute IOMT protocols to reliably perform

1) read/write/insert/delete operations in dynamic CDI databases for purposes of verifying pre-conditions and imposing post-conditions; each MN member (module) tracks one CDI database;
2) read/write/insert/delete operations in the dynamic MN-membership databases for inducting/ejecting modules into/from the MN; there is only one such database for each MN, maintained by the MN creator module;
3) read operations on the static MN-rules databases; the same database is referred to by every member of the MN. As all members of the MN S' are initialized with the same value S', they will only honor TPs in this common database.

2) Authentication and Privacy: Several key distribution schemes [4], [9] – [11] for establishment of pairwise secrets have been explicitly designed for scenarios involving severely resource limited participants. For example, the MLS protocol [11] will require every module to store only a single secret, and evaluate a single hash to compute a pairwise secret with any other entity. Two modules X and Y with secrets K_X and K_Y respectively can compute a common secret $h(K_X, Y)$ or $h(K_Y, X)$ depending on which entity has access to a pair-wise *public* value

$$P_{XY} = h(K_X, Y) \oplus h(K_Y, X) \tag{1}$$

If X has access to the public value the pair-wise secret is computed by X as $h(K_X, Y) \oplus P_{XY} = h(K_Y, X)$, which can be computed by Y by hashing its secret. The number of pair-wise public values required is not a serious concern as they can be stored virtually (outside the module).

Pair-wise secrets facilitated by schemes like MLS can be used for computing hashed message authentication codes (HMAC) for a) mutual authentication, and b) protecting privacy of secret components in messages. In the MN model, the built-in ability of MN modules to compute pairwise secrets are leveraged for the following specific purposes:

1) mutual authentication of message exchanges between MN modules (potential members and the MN creator) to join an MN;
2) mutual authentication of MN messages between MN-members,
3) mutual authentication and privacy of communications between MN members and external entities (who convey UDIs, and may query a MN member for the state of the MN)

A MN message can also be a *self-message* — from a member to itself. Self messages from a member X are authenticated using a self-secret S_X known only to X (randomly generated by X). Self secrets can also be used by a MN module to encrypt *other* secrets entrusted to the module (and store encrypted secrets virtually).

For example, external entities can employ the MN for distributing secrets. Specifically, let an external entity u share a secret K_{xu} with a MN member X. Entity u can utilize the following simple protocol to share a secret K with any number of entities, specified indirectly through a *context* f. The entity u sends values c, s_u, f related as

$$c = h(K, f) \text{ and } s_u = h(K_{xu}, c) \oplus K \tag{2}$$

The value c is a commitment to both the secret K and the context f, and serves as a public identifier for the secret K; s_u is the link-encrypted version of the secret K. The module (which can readily compute K_{xu}) computes $K = h(K_{xu}, c) \oplus s_u$, verifies that $c = h(K, f)$, and uses its self-secret K_x, to re-encrypt the secret K for storage as $s = h(K_x, c) \oplus K$ (more specifically, a tuple (c, s) is added to the CDI database tracked by the module).

An entity w with whom the module shares a secret K_{xw} may receive the secret K under some conditions. Firstly, the module X should be a) convinced of the existence of the record (c, s), and b) provided a value f satisfying

$$c = h(h(K_x, c) \oplus s, f). \tag{3}$$

In addition, if a MN-specific rule relates w and the context f (for example, a rule can be "existence of a record for index w in an IOMT with root f.") the module may output values c and $s_w = h(K_{xw}, c) \oplus K$ to entity w. Entity w (who has access to secret K_{xw}) can decrypt the secret and check its integrity by verifying that $c = h(K, f)$.

Given that simple protocols to support such generic functions can be easily implemented even in severely resource limited modules, in the rest of this paper, we focus on the process for designing the MN rules database (the "assurance software" for the IS) with an illustrative example.

III. Mirror Network Design Example

The creation of the MN rules database for an IS S can be seen as a process consisting of the several steps like 1) identification of desired assurances; 2) identification of the subset of data items (or one-way functions of data items) that need to be constrained in order to realize the desired assurances; 3) choice of ρ types of CDI databases, each possibly with a different interpretation of the index and value fields. 4) enumeration of ν event types and μ message types; and 5) specification of TP for each event in the form of pre/post-conditions. All components of the MN specification become leaves of a static IOMT with root S'.

A. Desired Assurances for a Cloud Storage Service

Cloud storage services offer a convenient way for users to share files between multiple platforms – even platforms owned by different users. From a security perspective, users of such a service desire assurances regarding the *integrity*, *privacy*, and *availability* of files.

In such a system, software running on end-user platform may periodically upload new files, or newer versions of existing files to the service. Users may also be able to specify access control lists (ACL) for every file they own, indicating read/write access restrictions for other users. For ensuring privacy of files, the files may be encrypted by users. As users who share an encrypted file will need to share the file-encryption secret, and as users may not have an out-of-network strategy for exchanging such secrets, the file storage service itself should cater for secure mechanisms for conveying file encryption secrets to authorized users.

The desired assurances for a remote file storage service [6] can be summarized as follows:

A1 The service will not alter files; only users explicitly granted the permission (by the owner) to modify the file can do so.
A2 File encryption secrets will not be abused by the service.
A3 The service will not modify ACLs, and strictly abide by ACL permissions.
A4 Only the latest version of the file will be provided by the service to authorized users (except when explicitly queried for an earlier version).
A5 After an ACL has been modified by an authorized user, the older ACL should not be used to determine the access privileges.
A6 A user with legitimate access rights will not be improperly denied access to the file.

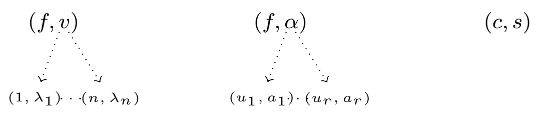
(f, v)
$(1, \lambda_1) \cdots (n, \lambda_n)$
(f, α)
$(u_1, a_1) \cdot (u_r, a_r)$
(c, s)

Figure 2. Structure of records in CDI databases. File database (left), ACL database (middle) and encryption-secret database (right).

B. Constrained Data Items and MN Roles

The problem of assuring the integrity of a file can be reduced to that of assuring the integrity of the cryptographic hash of the file. Likewise, assuring the privacy of contents of a file can be reduced to assuring the privacy of a file-encryption secret, and that it is made available only to authorized users included in the ACL for the file. Thus, from the perspective of realizing the desired assurances, the CDIs for the MN are 1) file hashes corresponding to every version of every file; 3) the ACL for every file; and 3) all file encryption secrets. The CDIs can be seen as three types of databases, with possibly different interpretations of the index field and value field in the database records.

1) File databases, indexed by unique file indexes;
2) ACL databases, also indexed by file indexes;
3) File-encryption-secret database where the index is a "key" identifier c.

In a record (f, v) in the file database (Figure 2, left), f is a unique file index, and v is the root of a nested IOMT. A record (n, λ_n) in the nested IOMT provides information λ_n regarding the n^{th}version of the file index f. The ACL database (Figure 2, middle) has records of the form (f, α) where α is the root of an IOMT capturing the ACL for file f. A record (u_i, a_i) in the nested IOMT with root α indicates that user u has access permission a (for file index f). For example, $a_i = 1$ for read access, $a_i = 2$ for read-write access and $a_i = 3$ for write-access to the ACL (users with access level 3 can even change the ACL α for f). In a record (c, s) (Figure 2, right) in the file-encryption-secret database the value s is an encrypted version of a file encryption secret K_f. Specifically, the secret is encrypted using the self secret of the module tracking the CDI database. The index computed as $c = h(K_f, f)$ is simultaneously a commitment to both the secret K_f and the *context* f. The implication of the context f is that secret K_f should be made privy only to users with access level 1 or higher in the ACL for file f.

Corresponding to the three different CDI database types, the MN employs members with $\rho = 3$ different member roles, say role F (for file version databases), role S (file encryption secrets) and role A (ACL). Any number of members may exist for each role, each maintaining data pertaining to different non overlapping ranges of file indexes.

C. UDIs

Modifications to the CDI databases are triggered by UDIs emanating from users of the service. Specifically, corresponding to creation of new files, new file indexes are added to the CDI databases. Corresponding to updates to a file with index f a new version record is added. File owners may also submit

ACLs for files (CDI α, the ACL root), delete files (remove record for file f), submit file encryption secrets, etc.

External entities (software running on end-user platforms) interact with S-role MN members to i) convey hashes and/or secrets corresponding to new file versions, changes to ACLs, and ii) query the MN for file hashes and secrets. The results of the query can then be compared with files provided by the service.

As in the CW model, requests from users, which are UDIs (as any user can send any request — even unauthorized ones), should be logged. As users have to share a secret with members with role S (as such members convey/accept encrypted file-encryption secrets to/from users), it is convenient to let members with role S to also maintain a log database. In this paper, in the interest of keeping the discussions simple, we shall ignore the log database.

D. MN Messages

Four types of MN messages can be defined: types ACL (to report ACL privilege), AU (to update ACL), VU (to add a new version), and FP (to report file parameters).

A message $ACL_{X,Y}(f, a, u)$ (from X to Y) of type ACL, can be created by a member X (with role A) and delivered to a member Y (with role F or S), indicating that user u has access permission a for file f. To generate such a message, X merely needs to confirm the existence of a record (f, α) in its CDI database, and the existence of record (u, a) in an IOMT with root α. An ACL message for a file f with $u = a = 0$ can be created only if the ACL IOMT root α for f is 0 (as we shall see later, such a message is used to trigger removal of (all versions of) file f).

A message $AU_{X,Y}(f, u, \alpha)$, can be created by S members and sent to A members, indicating a request from a user u to update the ACL for file f. While any user u can request an S member to create such a message, the AU message will be honored by the A-role member only if user u has access right $a = 3$ for file f. Such a user can also set the value α to zero (to request deletion of the file f).

A message $VU_{X,Y}(f, u, \lambda)$, represents a request to create a new version of file f. This can be created by S members and sent to A members to check if user u has the necessary access permission. After ensuring sufficient access rights, A members can then create another $VU()$ message addressed to an F member. Once again, while any user can trigger the S member to create a VU message the VU message will be honored by the A-member only if u has access right 2 or higher for file f. VU messages created by A members (in response to a VU message from a S member) trigger F members to appropriately modify their CDI database record for file f.

Message of type $FP_{X,Y}(\cdots)$ conveying parameters for version q of file f are created by F members and conveyed to S members who may then relay the contents of the message to users of the service.

E. Events and TPs

The MN rules database specifying pre-conditions and post-conditions for all TPs (corresponding to each of the 17 event types 01 to 17) is depicted in Table I. Each event type is associated with role type(s) of member(s) who may respond to the event (role type indicated in parentheses alongside the event number in column 1). Column 2 lists UDIs (as $\{\cdots\}$), and other inputs (OI) necessary to execute the TP. Only TPs corresponding external events accept UDIs. TPs for internal events are triggered by MN messages.

Columns 3 and 4 depict the pre-conditions and post-conditions, respectively. A MN message in pre-conditions (column 3) indicates receipt of the message. A message in post-conditions (column 4) implies the need to *create* such a message. As the events are listed from the perspective of a member X, all messages in pre-conditions indicate only the sender of the message (the receiver is always X); all messages in post-conditions indicate only the receiver (the sender is always X). As can be seen from the table, events 01, 02, 06, 11, and 12 are external events as they are *not* triggered by MN messages.

A tuple (x, y) in pre-conditions indicates the presence of record (x, y). $(x, 0)$ represents absence of record for x (or presence of place-holder for x). $(x, y) \to (x, y')$ in post-conditions implies the need to update the IOMT root to account for the update to the value of record index x (from y to y'). $(x, (y, a))$ in pre-conditions indicates the presence of a nested record (y, a) for a record with index x. $(x, (y, a) \to (y, a'))$ in post-conditions indicates the need to update the nested record (and accordingly, update the IOMT root). $s \rightsquigarrow s'$ indicates that s' and s are related through symmetric encryption.

A user reserves a file index f by creating event 01, which generates a AU message, which becomes input to event 08, which outputs a AU message, that becomes input to event 04, which results in a confirmation message to the user. A user u can also trigger event 01 to modify the ACL for file f. In this case, the AU message from event 01 triggers event 09. Only if the user has access level 3, the ACL is updated, and a AU response is created, which triggers event 04, to send a message to the user, confirming successful ACL modification. If user u does not have sufficient access right, event 11 is triggered to create a ACL message, which triggers event 03, which creates a message informing the user of his/her access right. If the file does not exist, event 12 is invoked instead to create the ACL message. If the user does not have access, or if the file does not exist, the user receives a message conveying values $\{f, u, 0\}$. Thus, if the user does not have access to a file f, the user does not even get to know if file f exists. A user can request deletion of a file by updating the ACL to 0. Following this, event 12 can be invoked with $u = 0$ to create a ACL message, that triggers event 13, to delete all versions of file f.

A user u can convey a new file version, by invoking event 02. The VU message invokes event 10. Only if the user has write access, is the output VU message created. If the update is the first version of the file, the VU message triggers event 14. Else, it triggers event 15. Both output a FP message which triggers event 05, resulting in a acknowledgement to the user, that the update was successful. If the user did not have access, or has read-only access, as earlier, event 11 or 12 can be triggered to convey this fact to the user.

Any user can provide a secret to the MN by triggering

TABLE I. MN Rules for Cloud Storage Service MN with $\rho = 3$ types of member roles, $\nu = 17$ types of events (and TPs), and $\mu = 4$ types of MN messages. Pre/post-conditions for 17 events are listed for a member with identity X.

Events	UDI / OI	Preconditions	Post-conditions
01(S)	$\{f,\alpha\}, Y, u$		$AU_Y(f,u,\alpha)$
02(S)	$\{f,\lambda\}, Y, u$	Y type Λ	$VU_Y(f,u,\lambda)$
03(S)		$ACL_Y(f,u,a)$	$\{f,u,a\}_X$
04(S)		$AU_Y(f,u,\alpha)$	$\{f,u,\alpha\}_X$
05(S)		$FP_Y(f,q,\lambda,q')$	$\{f,q,\lambda,q'\}_X$
06(S)	$\{f,c,s'\}$	$s' \rightsquigarrow K, c = h(K,f) \neq 0$	$K \rightsquigarrow s, (c,0) \to (c,s)$
07(S)		$ACL_Y(f,u,a>0), (c,s), s \rightsquigarrow K, c = h(K,f)$	$K \rightsquigarrow s', \{f,c,s'\}$
08(A)		$AU_Y(f,u,\alpha), (f,0)$	$(f,0) \to (f,\alpha), AU_Y(f,u,\alpha)$
09(A)	α'	$AU_X(f,u,\alpha), (f,(u,a)), a > 2$	$(f,\alpha') \to (f,\alpha), AU_{X,Y}(f,u,\alpha)$
10(A)	Z	$VU_X(f,u,\lambda), (f,(u,a)), a > 1$	$VU_Z(f,u,\lambda)$
11(A)	$\{f,u\}, Z, a$	$(f,(u,a))$	$ACL_Z(f,u,a)$
12(A)	$\{f,u\}$	$(f,0)$	$ACL_Z(f,u,0)$
13(A)	θ	$ACL_Z(f,0,0), (f,\theta)$	$(f,\theta) \to (f,0)$
14(F)	Z	$VU_Y(f,u,\lambda), (f,0)$	$(f,0) \to (f,(1,\lambda)), FP_Z(f,1,u,\lambda,1)$
15(F)	Z, q, λ'	$VU_X(f,u,\lambda), (f,(q-1,\lambda')), (f,(q,0))$	$(f,(q,0)) \to (f,(q,\lambda)), FP_Z(f,q,u,\lambda,q)$
16(F)	Z, q, λ	$ACL_Y(f,u,a>0), (f,(q,\lambda)), (f,(q+1,0))$	$FP_Z(f,q,u,\lambda,q)$
17(F)	Z, q, λ	$ACL_X(f,u,a>0), (f,(q,\lambda))$	$FP_Z(f,q,u,\lambda,0)$

event 06. To send a secret to a user, event 11 can be invoked to create a ACL message confirming that the user has the requisite access right, to trigger event 07. If the user does not have any access to f, event 11 or 12 can be triggered to convey this fact to the user.

A user u merely requesting file parameters for the latest version of a file can be satisfied by invoking event 11 to create a ACL message that triggers event 16. The preconditions for event 16 ensure that q is the latest version through non-existence of version $q+1$ (pre-condition $(f,(q+1,0))$). If the user requests a specific (older) version, event 17 is triggered instead. Note that the FP message created this time sets the field corresponding to the highest version number to 0 to indicate that it did not “bother to check” the highest version number. The FP message created by event 16 or 17 triggers event 05. Once again, if the user does not have any access privilege, or the file does not exist, event 11 or 12 can be triggered to convey this fact to the user.

At first sight, it may appear that replay attacks (for example, an old request for ACL update may be replayed), are ignored. This omission is deliberate, as strategies to prevent replays can be addressed by generic protocols for creation and verification of MN messages.

IV. Discussions and Conclusions

A novel mirror network model for securing ISes was outlined, driven by the need to reduce complexity of both the assurance software for any IS, and the platform on which the assurance software is executed. The complexity of the platform was kept low by deliberately constraining MN modules to perform only logical and hash operations. The complexity of the assurance software was minimized by constraining it to be a list of simple pre-conditions and post-conditions.

The only assumptions behind the MN approach are i) the correctness of the MN specification for the IS to be secured, and ii) the integrity of MN modules. No hardware/software of the IS itself need to be trusted to realize the desired assurances. As the MN specification is open, anyone with IS domain knowledge can verify its correctness. As the MN modules are deliberately constrained to possess simple and identical functionality, an infrastructure for mass production (possibly as chips), verification, and certification of MN modules can be realized at a reasonably low cost.

A. Comparison With Trinc

Another approach in the literature which leverages a simple trustworthy module specification to bootstrap system assurances is Trinc [2]. Specifically, a *trinket* is a module following the *Trinc* specification, whose sole purpose is the attestation of monotonic counters stored inside the trinket.

Similar to MN modules, every trinket has a unique identity. Every trinket also has an asymmetric key pair certified against its identity. A primary counter in the trinket is leveraged to create a plurality of secondary counters as follows. Whenever a new secondary counter is created, it is identified by the current value of the primary counter — which is incremented on creation of the new counter. Built-in functions of a trinket can be used to a) request a trinket to create a new counter with identity n, or b) bind (by computing a digital signature) some arbitrary value x and an incremented counter value $c_n' \geq c_n$ (where c_n is the current value of counter with identity n).

As an example, consider a scenario where a dynamic constrained data item (for example, a file hash) F, is bound to a counter with identity n, and value c_n, in a trinket with identity G. More specifically, let a value x bound to the counter (n, c_n, G) (through a signature of trinket G) represent a one way function of the signature of the provider/owner of file F. Whenever F is updated, the owner ensures that a fresh signature x' is bound to $(F, n, c_n' > c_n, G)$ — by requesting the trinket G to update the counter n to c_n', and issue a certificate binding x' to the updated counter. Anyone receiving the file F (even from an untrusted repository) can verify its freshness by obtaining the attestation by G (binding x' to its current counter (n, c_n', G)). Specifically, as the counter n is no longer c_n, the old value of F (along with signature x) can *not* be replayed by the repository.

To reduce the overhead associated with digital signatures, the Trinc specification also includes an alternate mechanism to attest/verify certificates, using shared symmetric secrets. In this case, however, a system-specific trusted third party is required to set-up secrets bound to specific counters of different trinkets — which are then shared between all entities that are required to verify the attestation.

Compared to MN modules, the main disadvantages of Trinc are as follows. Firstly, the Trinc specification limits the number of counters that can be "remembered" (and hence, the number of CDIs that can be reliably tracked by a trinket) to a small queue length (10 to 15). One way to overcome this limitation is by addition of built-in Merkle-tree functionality in Trinc for tracking any number of counters using a single monotonic counter [13].

Even with this addition, strategies to secure any practical system using Trinc will, unfortunately, require components *other* than the Trinc modules to be trusted. The reason for this is that Trinc by itself does not offer an explicit mechanism for binding a Trinc identity G to a specific subsystem/database that includes data F, or binding a specific piece of data F associated with the subsystem to a specific counter n in a specific trinket G. In the specific example above, the owner of F is trusted to do so. Unlike the MN model, where such system-specific bindings (to enforce system-specific rules) can be taken into account by the rich MN specification, the Trinc model does *not* have the ability to enforce IS specific rules. The shortcomings of Trinc are addressed by addition of simple (in terms of resource requirements), yet rich functionality to MNs modules — IOMT and mutual authentication capabilities — which require demand simple sequences of hash operations.

B. Conclusions

Current approaches to secure ISes predominantly rely on

1) ever changing *reactionary* measures (software updates, IDSes, firewalls) to improve the integrity of different subsystems and
2) cryptographic strategies for securing interactions between subsystems.

The former strategies are plagued by the possibility of new bugs in updates, and/or bugs in the very design of complex IDSes. Breaches in the latter (cryptographic) strategies [12] often result from the lack of integrity in the environment in which cryptographic protocols are executed (after all, a cryptographic algorithm is at most only as reliable as the platform in which cryptographic keys are stored and the cryptographic algorithm is executed). The novel and holistic MN approach to assure information systems is motivated by the often repeated (and unfortunately just as often ignored) maxim that "complexity is the enemy of security" [1]. The main novelty of the proposed approach stems from applying system integrity models to the assurance protocol of an IS (instead of the IS itself, as in the Clark-Wilson model).

Not withstanding complexity of ISes, rules that govern *how* data should be manipulated by ISes tend to be simple. Security breaches in systems rarely result from incorrect rules. Rather, they result from issues in the process of *implementing* the rules into a working system. This process includes numerous tasks performed during design, deployment and maintenance of the system, possibly by numerous personnel. It is far from practical to be able to assure the integrity of every component and personnel of such a complex process. The crux of the MN model is that it permits us to short-circuit this process to observe "if an IS is indeed abiding by design rules."

It is important to note that the MN approach does *not* obviate the need for measures necessary to root out malicious functionality in IS components, for if such functionality results in illegal modifications to the IS databases, the IS can no longer demonstrate its integrity to its users. In other words, all that the MN approach guarantees is that ISes will not be able to *hide* security violations from users (and other stake-holders) of the IS.

Our ongoing work involves developing MN rules for a wide range of information systems, with the longer term goal of developing a succinct language for expressing pre-conditions and post-conditions as instructions that can be easily interpreted by resource limited MN modules.

REFERENCES

[1] B. Schneier, "A plea for simplicity: you can't secure what you dont' understand," Information Security, November 1999.

[2] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc: Small Trusted Hardware for Large Distributed Systems," 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09), Boston, MA, April 2009.

[3] D .D. Clark, and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," in Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy (SP'87), May 1987, Oakland, CA; IEEE Press, pp. 184–193.

[4] M. Ramkumar, Symmetric Cryptographic Protocols, Springer, 2014.

[5] V. Thotakura, and M. Ramkumar, "Minimal TCB For MANET Nodes," 6th IEEE International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob 2010), Niagara Falls, ON, Canada, September 2010.

[6] S. D. Mohanty, and M. Ramkumar, "Securing File Storage in an Untrusted Server Using a Minimal Trusted Computing Base," First International Conference on Cloud Computing and Services Science, Noordwijkerhout, The Netherlands, May 2011.

[7] A. Velagapalli, S. Mohanty, and M. Ramkumar,"An Efficient TCB for a Generic Data Dissemination System," International Conference on Communications in China: Communications Theory and Security (ICCC-CTS), 2012.

[8] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," Advances in Cryptology, CRYPTO '87. Lecture Notes in Computer Science 293. 1987.

[9] M. Ramkumar, "Trustworthy Computing Under Resource Constraints With the DOWN Policy," IEEE Transactions on Secure and Dependable Computing, pp 49-61, Vol 5, No 1, Jan-Mar 2008.

[10] M. Ramkumar, "The Subset Keys and Identity Tickets (SKIT) Key Distribution Scheme," IEEE Transactions on Information Forensics and Security (TIFS), pp 39-51, Vol 5, No 1, Mar 2010.

[11] M. Ramkumar, "On the Scalability of a "Nonscalable" Key Distribution Scheme," IEEE SPAWN 2008, Newport Beach, CA, June 2008.

[12] Z. Durumeric et. al., "The Matter of Heartbleed," IMC 2014, Vancouver, Canada, Nov 2014.

[13] Sarmenta, L. F. G. Dijk, M. V. ODonnell, C. W. Rhodes, and S. Devadas, "Virtual Monotonic Counters and Count-Limited Objects using a TPM Without a Trusted OS," Proceedings of the 1st ACM CCS Workshop on Scalable Trusted Computing (STC06), pages 27–42, 2006.