2010 Fourth IEEE International Conference on Secure Software Integration and Reliability Improvement Companion

Modeling Efficient Multi-chained Stream Signature Protocol using Communicating Sequential Processes

Ahmet Koltuksuz
Yasar University
Department of Computer Engineering
Izmir, Turkey
ahmet.koltuksuz@yasar.edu.tr

Murat Ozkan, Burcu Kulahcioglu
Izmir Institute of Technology
Department of Computer Engineering
Izmir, Turkey
{muratozkan, burcukulahcioglu}@iyte.edu.tr

***Abstract*—Communicating Sequential Processes (CSP) is a process algebra, designed for modeling and analyzing the behavior of concurrent systems. Several security protocols are modeled with CSP and verified using model-checking or theorem proving techniques successfully. Unlike other authentication protocols modeled using CSP, each of the Efficient Multi-chained Stream Signature (EMSS) protocol messages are linked to the previous messages, forming hash chains, which introduces difficulties for modeling and verification. In this paper; we model the EMSS stream authentication protocol using CSP and verify its authentication properties with model checking, by building an infinite state model of the protocol which is reduced into a finite state model.**

Keywords—communicating sequential processes, model checking, security protocol verification

I. INTRODUCTION & RELATED WORK

Concurrency theory aims to model parallel systems by means of transition, net, graph or algebraic formalisms. CSP is an example of algebraic formalisms, which is designed specifically for the description of communication patterns of concurrent system components that interact through message passing. It is introduced by Hoare in 1978 [1] and further developed to its modern form as process algebra in 1984 [2].

Since Lowe's analysis [3] of Needham – Schroeder Authentication protocol with CSP using the model checker Failures - Divergences Refinement (FDR), many different security protocols [4] with different security properties [5][6] have been modeled and verified, including their authentication and secrecy specifications [7].

In this paper, we model Efficient Multi-chained Stream Signature (EMSS) [8] stream authentication protocol using CSP, which has been proved to be a useful formalism for modeling, specifying and verifying security protocols.

Unlike standard authentication protocols that have been modeled and verified using CSP [9]; sender in EMSS protocol broadcasts continuous stream of data along with hashes of previous messages. The recipient can check for authenticity of received messages after the reception of a signed message. Thus the hash chaining mechanism forms an important part of the protocol.

The study in [10] verifies the EMSS protocol with Team Automata, using compositional proof rules. Another study is presented in [8], which expresses the hash chain with a graph and reduces authentication problem into a reachability problem on the graph. For other stream authentication protocols, the verification processes are generally based on theorem proving techniques. An outstanding counter-example is [11], in which Timed Efficient Stream Loss-tolerant Authentication Protocol (TESLA) is verified using CSP and model checking techniques along with data-independence techniques based on [12][13].

We use a similar approach with [11], by first building an infinite state model of EMSS protocol. Then, we justify why this model is not suitable for verification using model checking and reduce it to a finite state model, by observing several properties of hashing. Finally we validate and verify our model using FDR.

The rest of the paper is structured as follows: In section 2, we briefly introduce the basic concepts of CSP and the model checker, Failures-Divergences Refinement (FDR). In section 3, we describe the EMSS protocol along with the hash chaining mechanism involved. In section 4, we present our infinite state model of the protocol, including the network model, symbolic cryptographic operations, the honest agents and the intruder process. Section 5 describes modeling of hash chains for the infinite state model. In section 6, we discuss why it is not feasible to verify this model and the changes necessary to express the same model using finite number of states. Section 6 is devoted to conclusion and future work.

II. COMMUNICATING SEQUENTIAL PROCESSES (CSP)

In CSP, systems are described in terms of processes which are composed of instantaneous and atomic discrete events. The relations between processes and operations on processes are formalized with *operational semantics* of the algebra. Using the operational semantics, every CSP process can be converted to an equivalent labeled transition system (LTS).

For a thorough reference of CSP, see [1][2][14][15].

A. Notation

The processes P and Q can be defined as:

$$P, Q ::= STOP \mid a \rightarrow P \mid P \,\square\, Q \mid P \sqcap Q \mid a{:}A \rightarrow P(a) \mid P \setminus A \mid P \underset{A}{\|} Q \mid \|_{A_P} P \mid P \,|||\, Q \mid \mu X \cdot F(X)$$

978-0-7695-4087-0/10 $26.00 © 2010 IEEE
DOI 10.1109/SSIRI-C.2010.23

IEEE computer society

In addition to these operations, more operations on processes and their inter-relationships are defined within *algebraic semantics* of CSP.

- $STOP$: Represents a deadlocked process.
- $a \rightarrow P$ *(Prefixing)*: The process will communicate the event a and then behave as process P.
- $P \Box Q$ *(Deterministic Choice)*: This process can behave either as P or Q, but the environment decides on which process to run.
- $P \sqcap Q$ *(Nondeterministic Choice)*: Similar to deterministic choice. The main difference is, environment cannot decide on which process to run but the selection is performed nondeterministically.
- $a{:}A \rightarrow P(a)$ *(Prefix Choice)*: This represents a deterministic choice between the events of the set A which may be finite or infinite. This notation allows representing input and output from channels. The input $c?x : A \rightarrow P(x)$ can accept any input x of type A along channel c, following which it behaves as $P(x)$. Its first event will be any event of the form $\mathrm{c.a}$ where a ∈ A. The output $c!v \rightarrow P$ is initially able to perform only the output of v on channel c, then it behaves as P.
- $P \setminus A$ *(Hiding)*: This process is similar to process P but the environment will not see the members of the event set A . The hidden events will occur immediately, as the environment is not able to see these events and synchronize with them.
- $P \underset{A}{||} Q$ *(Parallel Composition)*: Let A be a set of events, then the process behaves as P and Q acting concurrently, with synchronizing on any event in the synchronization set A . Events not in A may be performed by either of the processes independent of the other.
- $||_{A_P} P$ (*Indexed Parallel*): Defines a process which is composed of P processes with a set of respective interfaces A_P.
- $P \,|||\, Q$ (*Interleaving*): Similar with parallel composition but the two components do not interact on any events. This is achieved by synchronizing on nothing, so $P \,|||\, Q = P \underset{\emptyset}{||} Q$.
- $\mu X \cdot F(X)$ (*Recursion*): Represents a process which behaves like $F(X)$ but with every free occurrence of X in P (recursively) replaced by $\mu X \cdot F(X)$; where the variable X here usually appears freely within $F(X)$.

B. Denotational Semantics and Traces Model

Denotational semantics of CSP provide models for capturing and comparing the behavior of processes. Unlike the operational semantics, which is more directly interested in the processes within the system, behavioral models are related with processes at a more abstract level.

Different behaviors of the system are captured with different models, so they represent the system in different levels of detail. What they have in common is that each of the models provides an abstract way of representing a process as a set of behaviors it can have in one or more categories.

There are three widely used models in CSP, that represent the most commonly used behaviors, namely Traces ($\mathcal{T}$), Stable Failures ($\mathcal{F}$) and Failures / Divergences ($\mathcal{N}$) models.

In Traces model [14], only the actions which are visible to the environment are recorded. The sequence of all events that a process has communicated by some point in its execution forms the *trace* of the process. This model is only involved in finite traces of processes.

The traces model is useful for building the safety specifications of processes which define the behavior that the system should perform. Other models can be used to capture further properties of systems (e.g. liveness) by recording more detailed behavior of systems.

In this paper, only traces model will be considered, as we are interested in the safety properties of the protocols.

C. Refinement

Let P and R be processes such that, $P \sqcap R = R$, which indicates $P \sqcap R$ can be used anywhere instead of R. In other words, every behavior of P should also be a behavior of R. Hence, for such processes, we can say that P *refines* R or $R \sqsubseteq P$ (i.e. R is a refinement of P).

The refinement relation is reflexive, anti-symmetric and transitive, thus it forms a partial ordering between processes. In traces model, $STOP$ is the most refined process, because for any P, $traces(P) \supseteq traces(STOP)$. The least refined process is RUN_Σ, for any process P with an alphabet of Σ, $traces(RUN_\Sigma) \supseteq traces(P)$.

Having defined the refinement relation, it is straightforward to apply it on $\mathcal{T}$, $\mathcal{F}$ and $\mathcal{N}$; however it is beyond the scope of this paper.

D. Tool: Failures-Divergences Refinement (FDR)

FDR [16] is a model checker, designed to establish verification results for systems, which are modeled and specified as CSP_M(Machine Readable CSP) scripts [17]. In addition to refinement checking, FDR can also perform determinism and deadlock checks on processes.

To verify a system for correctness, we construct a process representing the system (i.e. the implementation process) and a specification process. FDR checks whether every behavior of the specification process is also a behavior of the implementation process, (i.e. refinement check) by performing an exhaustive search on the generated state space and it can generate counter-example traces, like other model checking tools.

FDR translates processes into a corresponding finite LTS according to operational semantics through *compilation*. In compilation, a two-level approach is used for calculating operational semantics: The low level is fully general but relatively inefficient, whereas the high level is restricted (e.g. it cannot handle recursion) but much more efficient in space and time.

Like other model checking tools, FDR suffers from state space explosion problem. Compression techniques can be

applied on processes and scripts can explicitly specify which technique to apply on each process.

III. The Efficient Multi-chained Stream Signature Protocol

EMSS (Efficient Multi-chained Stream Signature) [8] is a stream authentication protocol, based on signing a small number of special packets in a data stream. Each packet is linked to a signed packet via multiple hash chains. This is achieved by appending the hash of each packet (including possible appended hashes of previous packets) to a number of subsequent packets. Hence the protocol claims to amortize the cost of signing each packet, and achieve one-way authentication, even in lossy channels.

In [8] several schemes are defined with probabilities of successful verification of each packet, however, $(1,2)$ scheme will be used as our protocol.

The protocol can formally be described as follows:

1. $A \rightarrow B : P_0,\ P_0 = d_0$
2. $A \rightarrow B : P_1,\ P_1 = d_1, h(P_0)$

i. $A \rightarrow B : P_i,\ P_i = d_i, h(P_{i-1}), h(P_{i-2})$

… … …

n. $A \rightarrow B : P_n,\ P_n = \{h(P_{n-1}), h(P_{n-2})\}_{sk(A)}$

where, h is the hash function and d represents the data.

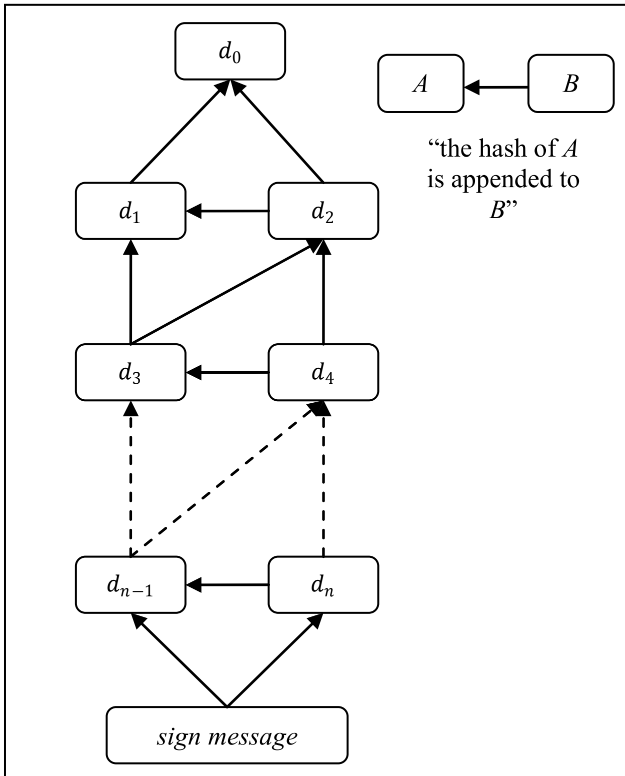

Figure **1**. The relation between packets in EMSS protocol

In first message, only d_0 is sent to the receiver. In the second message, d_1 is sent to the receiver along with the hash of d_0. Next message consists of d_2 and hashes of d_1 and d_0. Thus, data d_0 may be authenticated even if the message P_1 is lost or unauthenticated. Likewise, the message P_i contains the hashes of messages P_{i-1} and P_{i-2} where $2 < i < n$. The last message consists of the hashes of P_{n-1} and P_{n-2}, encrypted with $sk(s)$ - the private key of the sender (i.e. the signature message). This hash chaning mechanism of EMSS is illustrated in Fig. 1.

The signature message is repeated periodically for every n messages. Clearly, the receiver must buffer messages until a signature message arrives, for authenticity verification.

Receiver can only check for the authenticity of the previous messages; if and only if the signature in the last message is correct and there is a chain of hashes that reaches to one of the signed hashes.

The EMSS protocol aims to authenticate sender to multiple receivers; providing one-way authentication between agents. Additionally the sender cannot claim that the authenticated messages are not sent by it, so the protocol claims to provide nonrepudiation properties. However, the protocol does not make any secrecy claims; contents of the messages are readable by everyone.

In this paper, we assume that sender already knows the identities of receivers which run the protocol. We further assume that the receivers know the public key of the sender.

IV. Modelling EMSS Protocol Using CSP

In this section we present a finite state model of EMSS protocol and the concepts and assumptions used in the modeling process.

A. Modeling Assumptions

We assume that the agents show no behavior other than the behavior described in the protocol. Also, agents know their own secret keys, and the public key of the sender agent is already known by the receiver. We also assume that the intruder has the capabilities of the Dolev - Yao intruder model [18] which can eavesdrop, replay, modify or inject messages.

Additionally, we assume to have a perfect cryptosystem in order to focus our analysis on the protocol, which means that nobody can decrypt the messages unless they know the secret keys, a ciphertext $\{m\}_K$ can be generated by principal possessing m and K. For hashing, we assume that the contents of hashed messages cannot be retrieved by any agent and hashes of different messages are always different.

B. Symbolic Data Types

We represent protocol messages and its contents (e.g. agent identities, cryptographic items, etc...) in a structured way. Messages and their compound subcomponents are constructed from simpler data items by concatenation. This means that a communication of a compound message is equivalent to the communication of all of its atomic subcomponents.

Hence we define a data type $fact$ to represent atomic data items such as agent identities, keys, encrypted and hashed messages and compound data items that are built using atomic data items:

$$fact ::= Sq.\langle fact \rangle \mid Garbage \mid$$
$$pk.Agent \mid sk.Agent \mid$$

$key.(Agent, Agent) \mid$
$Pk.(PKey, \langle fact \rangle) \mid$
$Encrypt.(SKey, \langle fact \rangle) \mid$
$Hash.(\langle fact \rangle)$

The set $Agent$ is the set of agent labels. The identity of an agent is represented by its label A, where $A \in Agent$.

Sequencing construct is defined as $Sq.\langle fact_0, fact_1, \ldots, fact_n \rangle$, which represent a sequence of facts; $fact_0$ to $fact_n$, where n is an arbitrary number. Using sequencing allows us to create and use compound messages.

Invalid or lost messages are symbolized as $Garbage$. It can be viewed as a placeholder for such messages.

Public and secret keys of agents are represented respectively as $pk.Agent$ and $sk.Agent$. Hence, a key pair for an agent A becomes $pk.A$ and $sk.A$; which are the *dual* of each other. Similarly, symmetric keys are represented with $key.(Agent, Agent)$. A session key between A and B, k_{AB}, is represented as $key.(A, B)$. We also define sets $PKey$ and $SKey$, which are the sets of all private and public keys and all symmetric keys, respectively.

Encrypted messages using public key and symmetric cryptosystems share the same representation; which is $tag.(key, \langle fact \rangle)$, where tag represents the encryption type and key and $\langle fact \rangle$ symbolizes the key and the clear text message respectively.

The tag Pk symbolizes that the public key encryption is used. Thus, an encrypted protocol message using public keys has the form $k.(PKey, \langle fact \rangle$. A protocol message $\{A\}_{pk(A)}$ is represented as $Pk.(pk.A, \langle A \rangle)$.

Similarly, the tag $Encrypt$ symbolizes that symmetric key encryption is used. Such a message has the form $Encrypt.(Skey, \langle fact \rangle)$. A protocol message $\{A\}_{kAB}$ is represented by $Encrypt.(key.(A, B), \langle A \rangle)$.

Additionally, the data type $fact$ is defined recursively to allow the use of nested encryption, nested hashing and sequencing inside an encryption.

C. Cryptographic Operations

In the analysis, the main focus of interest is finding the attacks which are mounted on the behavior of protocol participants, not on the cryptosystems used in the protocol. Furthermore, CSP is not a suitable formalism to represent and verify cryptosystems. Hence, we use symbolic cryptographic operations; which are abstracted away from the underlying cryptographic mechanism.

Symbolic operations build data from atomic data items, by marking these data items with suitable tags. Encryption, decryption and hashing can be represented symbolically in this manner.

1) Symbolic Encryption & Decryption

Symbolic encryption with a key and a plaintext involves tagging these data types as encrypted.

The use of symbolic decryption can be explicit or implicit. An agent process might synchronize with an encrypted message communicated through a channel. Since synchronization will mean that all fields within the tag should be the same; then that process would implicitly decrypt the message; without using additional mechanisms.

In intruder process, decryption and encryption operations are defined as deductions. Intruder is able to encrypt a message if *any* key and the message are in its knowledge base. However, it may not know the contents of the message until it possesses the right key (i.e. the decryption operation).

2) Symbolic Hashing

Symbolic hashing involves tagging the data to indicate hashing. This representation adheres to the properties of hashing operation and our modeling assumptions because of several reasons:

- Symbolic hashing does not explicitly prevent other processes from reading the contents of the hashed messages. However, our processes (including intruder process) don't access the contents of the hashed messages.
- Two hashed messages cannot have two equivalent values with symbolic hashing. Clearly, $Hash.(A) \neq Hash.(B)$ when $A \neq B$.

Being simple and elegant for most cases, this standard representation causes problems in hash chaining, which is covered next section.

D. Honest Agents

The honest agents are the agents that are known to be running the process as in protocol specification. In our model, $Send$ and $Recv$ processes are honest agents where the parameter id symbolizes the identity of an agent.

1) Sender Agent

We define the sender process in three sub processes: First process is responsible for sending 1st and 2nd protocol messages; which have a different format than i^{th} message. Second process sends i^{th} message and the last process sends the signature message.

The first part chooses a receiver to communicate with, receives the data from $DataStream$ process and forms and transmits protocol messages using the $send$ channel.

After sending initial protocol messages, the identity of the receiver agent is passed to the following process along with the hashes of the previous messages.

Note that, $P_0 = d_0$ and $P_1 = \langle d_1, hash(d_0) \rangle$.

$Send_0(id) = \sqcap\ b: Agent \bullet$
$\quad getData.id?d_0 \to send.id.b.d_0 \to$
$\quad getData.id?d_1 \to$
$\quad send.id.b.Sq.\langle d_1, hash(d_0) \rangle \to$
$\quad\quad Send_i(id, b, hash(P_0), hash(P_1))$

The second part requests for another data item, hashes the old data items and sends the ith protocol message. Alternatively; it may choose to send the signature message; which is resolved nondeterministically:

$Send_i(id, b, h_1, h_2) = Send_n(id, b, h_1, h_2) \sqcap$
$\quad (getData.id?d_i \to$
$\quad send.id.b.Sq.\langle d_i, h_1, h_2 \rangle \to$
$\quad\quad Send_{i+1}(id, b, h_2, hash(P_i))$

The final part of the sender agent involves sending the signature message. After sending the signature message; we might start over with a new run of the protocol:

$Send_n(id, b, h_1, h_2) =$

$send.id.b.Pk.(sk.id, Sq.\langle h_1, h_2\rangle) \rightarrow$
$Send_0(id)$

Note that in a new run; old hash and data values are assumed to be forgotten by the sender; hence the processed data and the hash items are not sent as parameters to the new protocol run.

2) Receiver Agent

Similar to the sender, the receiver is modeled using three sub processes, each having similar roles with their $Sender$ counter parts.

However, unlike the sender process, the receiver should also check the authenticity of the received protocol messages after the reception of a signature message. We just buffer them until a signature message arrives in the set $Recvd$:

$Recv_0(id) = \square\, a{:}\,Agent, dr_0{:}\,AllData\ \bullet$
$recv.a.id.dr_0 \rightarrow$
$\square\, hr_0{:}\,AllHashes, dr_1{:}\,AllData\ \bullet$
$recv.a.id.Sq.\langle dr_1, hr_0\rangle \rightarrow$
$Resp_i(id, a, \{\langle dr_0\rangle, \langle dr_1, hr_0\rangle\})$

The second process is responsible for collecting any i^{th} message sent by the sender. The sender might choose to send the signature message after 2nd message; so this part should be prepared to receive a signature message.

$Recv_i(id, a, Recvd) = Recv_n(id, a, Recvd) \square ($
$\square\, hr_1, hr_2 : AllHashes, d_r : AllData\ \bullet$
$recv.a.id.Sq.\langle d_r, hr_1, hr_2\rangle \rightarrow$
$Recv_{i+1}(id, a, Recvd \cup \{\langle d_r, hr_1, hr_2\rangle\}))$

The final process is responsible for the reception of the signature message.

$Recv_n(id, a, Recvd) = \square\, hr_0, hr_1 : AllHashes\ \bullet$
$recv.a.id.Pk.(sk.a, Sq.\langle hr_0, hr_1\rangle) \rightarrow$
$Check(Recvd, \{\langle hr_0, hr_1\rangle\}, n)$

3) Check Process

EMSS protocol does not explicitly specify a method for checking the authenticity of received messages, allowing us to make assumptions about the behavior of an agent. This makes the model of EMSS protocol more interesting than other protocol models, in which *all* behavior of agents are specified in the protocol.

After the arrival of the signature message, we should check the buffer for (possibly) authentic messages.

We pass the buffer of received packets (i.e. $Recvd$ set), set of validated messages (i.e. initially the hashes in the signature message) and the total number of received packets n as parameters to checker process.

Basically, $Check$ process begins from the $(n-1)^{th}$ message and for each message P_i, it checks that, whether any of the messages P_{i+1} and P_{i+2} are in validated set. We cannot verify a message, if there is not a link to at least one validated message. We also check if the hash values in the validated messages are equal to this message's hash.

If a message meets these conditions, it is considered as an authentic and sent via $putData$ channel. Moreover, that message is added to the verified set V and removed from the message buffer set R.

$Check(R, V, i) =$
$if\ i < 0\ then$
$Recv_0(id)$
$else$
$if\ \big((P_{i+1} \in V)\ and\ hash(P_i) = hp_1(P_{i+1})\big)$
$or\ \big((P_{i+2} \in V) and\ hash(P_i) = hp_2(P_{i+2})\big)$
$then$
$putData.data(P_i) \rightarrow$
$Check(R - P_i, V \cup P_i, i-1)$
$else$
$Check(R - P_i, V, i-1)$

On the other hand, if a message fails these conditions, it is considered as non-authentic, and the message is removed from set R.

The helper functions $hp_1(P_i)$ and $hp2(P_i)$, which appear in $Check$ process, return first or second hash value in P_i, respectively. That is, $hp_1(P_i) = h_{i-1}$ and $hp_2(P_i) = h_{i-2}$ if $P_i = Sq.\langle d_i, h_{i-1}, h_{i-2}\rangle$. The function $data(P_i)$ returns the data part of the message P_i.

Note that, if more than two consecutive messages cannot be verified, then the second if condition will never be satisfied by subsequent recursions, which matches with our expectation.

E. The Intruder

The intruder process obeys the Dolev - Yao model with perfect cryptography assumption. It can intercept or overhear messages, add them to its knowledge base and forge fake messages using its knowledge base.

To generate messages from a set of known messages; intruder uses a $Deductions$ set. Deductions are composed of pairs (m, S); where S is a set of facts and m is the message that can be deduced using all of the facts in S.

Intruder initially knows all public facts and the private facts about an agent that it can impersonate.

The knowledge base of the intruder is not stored as a set; rather it is modeled as a network of two state processes, for performance issues stated in [19]. For each fact, reachable by intruder, $ignorantof(f)$ represents an unknown fact f and $known(f)$ represents a known fact f. These processes can carry out $infer$ actions, which uses deductions to figure out unknown information from known facts.

$SayKnown$ process does not make any inferences, but generates already known messages to be sent to the receiving agent. The $chase()$ function is a compression function to be used by FDR, which instructs the model checker to apply partial-order reduction on the selected process.

Overall intruder process is (without irrelevant details):

$Intruder =$
$chase(||_{f:LearnableFacts}\ ignorantof(f)) \backslash \{infer\}$
$|||\ SayKnown$

For details about the intruder, see [19], as we use it with slight changes and it is a general intruder model which is applicable for most protocol models.

F. The Network

We use a similar network model with [11], which is depicted in Fig. 2. This model is also appropriate as our model with simple changes because we also use Dolev-Yao intruder model. We define $Send = Send_0(Alice)$, $Recv = Recv_0(Bob)$, and $Alice,\ Bob,\ Mallory \in Agent$.

$Send$ and $Recv$ processes can only communicate with $Intruder$, thus, the $Intruder$ process models the medium.

$$System = (Send \,|||\, Recv\,) \,||\, Intruder$$

As obvious from protocol definition; $Send$ process only sends but does not receive any data and $Recv$ process does not send any data. However the $recv$ channel between $Send$ and $Intruder$ and the $send$ channel between $Recv$ and $Intruder$ are included in the model for the sake of generality.

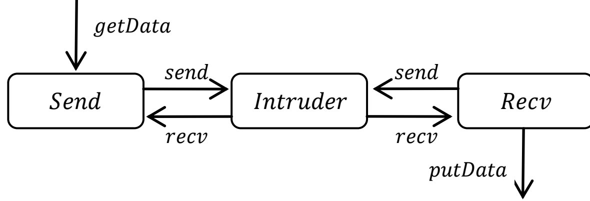

Figure 2. Overview of the network model showing processes and channels

Different from other authentication protocols, we have to send data within the messages in EMSS protocol. We assume that the data is retrieved using $getData$ channel from some other process. The receiver buffers messages until the signature message arrives and $Check$ process uses $putData$ channel to send validated data items on this channel.

V. Modeling the Hash Chain In EMSS Protocol

This section discusses the problems that we encountered during the modeling of EMSS protocol and proposes a method to overcome these problems. First we explain the straightforward approach and state the problems. Then, we present our fixed size hashed message approach for hash chain modeling.

A. The Straightforward Approach

For protocol models that do not use hash chaining, tagging data with $Hash.(\,)$ is enough. However; with the tagged representation, the size of the hash increases with the size of the message. The use of hash chains can create arbitrarily long hash representations; as in the EMSS protocol. Using this method, i^{th} message would be represented as:

$$\begin{aligned} &send.Alice.Bob.Sq.\langle d_i, \\ &\quad Hash.(Sq.\langle d_{i-1}, Hash.(Sq.\langle d_{i-2}\rangle \ldots)\rangle), \\ &\quad Hash.(Sq.\langle d_{i-2}, Hash.(Sq.\langle d_{i-3}\rangle \ldots)\rangle), \\ &\rangle \end{aligned}$$

where $Alice$ and Bob represents the identities of $Sender$ and $Responder$ processes, respectively.

Another issue is the types of $send$ and $receive$ channels. In CSP; it is possible to define channels with arbitrary type; however for model-checking purposes; the types of channels should be defined explicitly. Clearly, definition of channels that accepts messages with arbitrary number of nested hashes would be a problem while expressing the protocol for model-checking tools.

Because of these reasons, we cannot model hash chaining in the EMSS protocol using a tagged representation.

B. Proposed Approach

In our approach, the size of a hashed message is aimed to be fixed, whatever the message length is. So, we want to represent i^{th} protocol message with:

$$send.Alice.Bob.Sq.\langle d_i, h_{i-1}, h_{i-2}\rangle$$

where h_{i-1} and h_{i-2} represent the hash values of previous two messages.

Representing the protocol messages in such a manner requires a relationship between the hash values and the protocol messages to be hashed. We provide this relationship by using a mapping set$DataHash$, which contains all the possible messages assigned to a unique hash value. Afterwards, we can use this set to obtain the hash value of a protocol message.

We define $AllData$ set, which represent all possible data values in the protocol:

$$AllData = \{d_i \mid i \in \{0..m\}\,\}$$

where m represents the upper bound on data values.

Likewise, the $AllHashes$ set represents all hashes on the system, with an upper bound of n:

$$AllHashes = \{h_i \mid i \in \{0..n\}\,\}$$

Now, we create a map of hash values to corresponding contents of hashed messages. The map is a set of tuples which associate each hash value with a respective data value. We need to define different tuples for each of the possible protocol messages:

- $i = 1$: $(h_i, \langle d_x\rangle)$
- $i = 2$: $(h_i, \langle d_x, h_y\rangle)$
- $i < n$: $(h_i, \langle d_x, h_y, h_z\rangle)$

where h_i represents unique hash value of protocol messages $\langle d_x\rangle$, $\langle d_x, h_y\rangle$ and $\langle d_x, h_y, h_z\rangle$; where $h_i, h_y, h_z \in AllHashes$ and $d_x \in AllData$.

For 1st and 2nd messages, this set is defined as:

$$DataHash_1 = \{\,(h_i, \langle d_i\rangle) \mid i \in \{0..m\}\,\}$$

$$\begin{aligned} DataHash_2 = \{\, & \big(h_i, \langle d_x, h_y\rangle\big) \mid 0 \le x < m, \\ & 0 \le y < m, i = |DataHash_1|(x+1) + y\,\} \end{aligned}$$

and for i^{th} message:

$$\begin{aligned} DataHash_i = \{\, & \big(h_i\,, \langle d_x, h_y, h_z\rangle\big) \mid \\ & 0 \le x < m, \\ & 0 \le y, z < k, \\ & i = k(x+1)(y+1) + z\,\} \end{aligned}$$

where $k \;= \sum_{c=1}^{i-1} |DataHash_c|$

The $DataHash$ set with a maximum length of n is composed of union of all sets and represents all possible hash values of protocol messages:

$$DataHash = \bigcup_{i=1}^{n} DataHash_i$$

The $hash(\,ds\,)$ function is defined to return the unique h_i value assigned to a particular ds:

$$hash(ds) = \{h' \mid (h', s) \in DataHash,\; s = ds\}$$

This hashing scheme obeys our modeling assumptions about hashes; we cannot retrieve the contents of the message by using only the hash value and all hashes in the system are distinct.

VI. Verification of the Model

In previous sections, we described the EMSS protocol using CSP and modeled the hash chaining mechanism, by

assigning each possible protocol message a unique hash value. While this approach can model the protocol, problems arise when we want to perform verification using model checking.

The first problem is the infamous state-space explosion problem. When a model checking tool tries to construct the state space of the system, a new state is generated for each distinct item in the $DataHash$ set. However, the cardinality of $DataHash$ set can be quite high, even if a small $AllData$ set is used. For example, if we assume that $|AllData| = 2$, there will be around 150 million different ways of constructing 4th protocol message, as given in Table 1.

TABLE I. THE SIZE OF DATA HASH SET

i	0	1	2	3	4
$\|DataHash_i\|$	2	4	72	12168	$\sim 150\ M$

This exponential rise of the $DataHash$ set makes model checking infeasible beyond 3 messages, even if we limit ourselves with 2 distinct data values.

Another problem is the time complexity of the $hash$ function. As the $DataHash$ set grows bigger, it would take more time to select the correct hash value from the set, assuming we could hold all the required states in memory.

We need to refine the model, by abstracting unnecessary details away from the proposed model. To begin with, we analyze and improve our hash chain model and then we carry our optimizations to the processes in the network.

1) Revising the Data Model

As we discussed in previous sections, the data values are assumed to be taken from the $AllData$ set. This set contains m different data values, which symbolizes all the data values in the system.

However, we are also not interested in the actual contents of the data transferred but we're interested in the source of the data, for specification purposes. Thus we drop $AllData$ set and represent the data sent by agents as $Data.Agent$, which belongs to the data type $fact$.

Note that, this revision also means that we do not need another external process inputting data on the $getData$ channel. Hence, this revision reduces the state space by removing a parallel composition from the system.

2) Bounding the Size of Hash Sets

In our hashing mechanism, we explicitly calculated the correct hash value of a given protocol message using the $hash$ function, which maps each protocol message to a single *correct* hash value. All other hash values would be *incorrect*, as there is one correct hash value.

This observation means that, while we are building the state space of the system, we use one state for representing the correct hash and all the remaining states for representing the incorrect hash value. We reduce the state space greatly, since we represent all incorrect hash values using a single symbolic value.

Thus, we declare two symbolic facts; h_C, representing a correct hash value and h_I, representing an incorrect hash value, with respect to the corresponding previous message. In this case, message $P_i = \langle data.Alice, h_C, h_I \rangle$ represents a message, carrying data from agent $Alice$ and correct hash value for P_{i-1} and an incorrect hash value for P_{i-2}. As we use one symbolic value for all of the incorrect hash values (and we are not interested in exactly which incorrect value has arrived) this approach greatly reduces the state space of the system and makes model checking possible, without losing any attacks.

Next, we need to redefine $AllHashes$ and $DataHash$ sets. The new symbolic hash values h_C and h_I will become members of $AllHashes$ such that $AllHashes = \{h_C, h_I\}$. Also, the $DataHash$ set no longer needs to contain all possible mappings of hashes and messages. We redefine $DataHash$ such that it contains only the messages whose hashes are correct:

$$DataHash = \{\langle data.A \rangle, \langle data.A, h_C \rangle, \langle data.A, h_C, h_C \rangle\}$$

The receiver needs to calculate the hashes of incoming messages, so we use a simplified hash function:

$$hash(ds) = if\ (h = \emptyset)\ then\ h_I\ else\ h_C$$

where $h = \{\ h \mid ds \in DataHash, ds = s\ \}$.

Having defined a new hashing mechanism, we need to show that this mechanism is compatible with our hashing assumptions:

- Model satisfies our first assumption, because we have no mechanism to deduce the input from hash values h_I and h_C.
- As we map all incorrect hash values to a single symbolic value h_I, different input messages can have the same hash value. However, hash h_C and h_I are only symbolic values denoting the correct or incorrect hash of a message. We are not interested in the exact hash value but rather we are interested whether the value is right or wrong. In this sense, we can say that the model satisfies this condition

3) Revisions on the Agent Processes

We remove $getData$ channel from the $Send$ process and also, there's no need to send hash values of previous messages to processes, we assume that sender always calculates and sends the correct hash value. Additionally we define the process $Last$ to be used for validation and verification, which defines the behavior of the system after the protocol run.

$$\begin{aligned}
&Send_0(id) = \sqcap\ b{:}\ Agent\ \bullet \\
&\qquad send.id.b.data.id \rightarrow \\
&\qquad send.id.b.Sq.\langle data.id, h_C \rangle \rightarrow \\
&\qquad\quad Send_i(id, b) \\
&Send_i(id, b)\ =\ Send_n(id, b)\ \sqcap \\
&\qquad (send.id.b.Sq.\langle data.id, h_C, h_C \rangle \rightarrow \\
&\qquad\quad Send_{i+1}(id, b)) \\
&Send_n(id, b) = \\
&\qquad send.id.b.Pk.(sk.id, Sq.\langle h_C, h_C \rangle) \rightarrow \\
&\qquad\quad Last
\end{aligned}$$

Similarly, the $Recv$ process has slight changes, such as removing $AllData$, so we do not rewrite $Recv$ process here.

In the $Check$ process, we do not need to compare the hash of current message with the hash values in previous messages. We assume that, if any of previous messages have been verified, then they contain the right hash value for the current message. Thus, we only need to check whether any

of the previous messages are in checked set. Also, we do not want to start a new process run after the check is finished, so we replace $Recv_0$ with $Last$ process.

$$
\begin{aligned}
&Check(R, V, i) = \\
&\quad if\ i < 0\ then \\
&\quad\quad Last \\
&\quad else \\
&\quad if\ hash(P_i) = h_C\ and \\
&\quad\quad \big((P_{i+1} \in V)\ or\ (P_{i+2} \in V)\big) \\
&\quad then \\
&\quad\quad putData.data(P_i) \rightarrow \\
&\quad\quad\quad Check(R - P_i, V \cup P_i, i - 1) \\
&\quad else \\
&\quad\quad Check(R - P_i, V, i - 1)
\end{aligned}
$$

4) Validation of the CSP Model

We want to make sure that we built the right model i.e. validating the model, by building a test specification. For validation, we define $Last$ process such that $Last = test.ok \rightarrow STOP$. We assume that the model is correct, if a state where two agents can synchronize on $test.ok$ is reachable. That is actually hiding all other events within the system and checking if any $test.ok$ event occurs in traces:

$$STOP \sqsubseteq_T System \setminus \{\Sigma - \{test\}\}$$

This property is checked by FDR and it is not satisfied. FDR returns traces in which the $test.ok$ event occurs, meaning that such a state is reachable and our model is valid.

5) Verification of CSP Model

EMSS protocol claims to authenticate the sender to receiver agent, which means that the receiver should only accept data items that it believes to be sent by the correct sender. To verify this property, we must check whether a state exits in which receiver agent accepts a data that has not been sent by the stated sender:

$$STOP \sqsubseteq_T System \setminus \{\Sigma - \{putData.Alice\}\}$$

This property is checked by FDR and satisfied, indicating that the EMSS protocol fulfils its authentication claims.

VII. CONCLUSIONS & FUTURE WORK

In this paper, we discussed the challenges that were encountered during the modeling of the EMSS protocol using CSP. The main challenge was the modeling of the hash chain mechanism; which was not possible using the straightforward approach. We overcame this problem by using a fixed sized hashed message approach.

Then, we tried to convert the infinite state model of the protocol into a finite state model, using fixed sized hashed messages. We used symbolic hash values to represent the correct and incorrect hashes of messages, which enable us to validate and verify our model using the model checker FDR, without losing any attacks.

Although [11] uses data independence techniques to build the finite state model, we do not think that they are necessary to apply to our model of the EMSS protocol. This is mainly because of the protocol uses data items in its definition and we are not interested in the values of the data items. Additionally, the $hash$ function is available to agents and the intruder so the recycling of hash and data items is unnecessary.

From the definition of the EMSS protocol given in [8], we cannot determine how the protocol behaves using multiple parallel sessions of protocol runs, so we assumed that each run is performed involving one sender and one receiver.

This work may be extended to include the checking of both authentication and nonrepudiation properties involving unbounded number of receivers, which is not considered in this paper.

REFERENCES

[1] C. A. R. Hoare, “Communicating Sequential Processes”, ACM, 1978, Communications of the ACM, Vol. 21, pp. 666-677.

[2] S. D. Brookes, C. A. R.Hoare, and A.W. Roscoe, “A theory of Communicating Sequential Processes”, 3, New York, NY, USA : ACM, 1984, Journal of the ACM, Vol. 31, pp. 560-599.

[3] G. Lowe, “Breaking and fixing the Needham-Schroeder public-key protocol using FDR”, Software-Concepts and Tools, 17(3), 1996. pp. 93–102.

[4] S. Schneider, and R. Delicata, “Verifying security protocols: An application of CSP”, In Communicating Sequential Processes, 2005, pp. 243–263.

[5] L. H. Nguyen and A.W. Roscoe, “Authentication protocols based on low-bandwith unspoofable channels: a comparative survey”, Unpublished, 2009.

[6] C. Dilloway and G. Lowe, “On the specification of secure channels”, Proceedings of the Workshop on Issues in the Theory of Security (WITS '07), 2007.

[7] S. A. Shaikh, V. J. Bush, and S. A. Schneider, “Specifying authentication using signal events in CSP”, Computers & Security, 28, 2009, pp. 310–324.

[8] A. Perrig, R. Canetti, J. D. Tygar and D. Song, “Efficient authentication and signing of multicast streams over lossy channels”, 2000, IEEE Symposium on Security and Privacy, pp. 56-75.

[9] B. Donovan, P. Norris and G. Lowe, “Analyzing a library of security protocols using Casper and FDR”, In Proceedings of the Workshop on Formal Methods and Security Protocols, 1999.

[10] F. Martinelli, M. Petrocchi and A. Vaccarelli, “Analysing EMSS with compositional proof rules for non-Interference”, Workshop on Issues in the Theory of Security (WITS’03), 2003, pp. 52–53.

[11] P. J.Broadfoot and G. Lowe, “Analysing a stream authentication protocol using model checking”, In ESORICS ’02: Proceedings of the 7th European Symposium on Research in Computer Security. 2002, London, UK: Springer-Verlag. pp. 146–161.

[12] R. S. Lazic, “A semantic study of data-independence with applications to the mechanical verification of concurrent systems”, 1998, Oxford University D. Phil thesis.

[13] A. W. Roscoe and P. J. Broadfoot, “Proving security protocols with model checkers by data independence techniques”, Journal of Computer Security, 1999.

[14] A.W. Roscoe, The theory and practice of concurrency. s.l. : Prentice Hall, 1998.

[15] S.A. Schneider, Concurrent and real-time systems: the CSP approach. s.l. : John Wiley., 1999.

[16] M. Goldsmith, “FDR2 User’s Manual”, version 2.82. 2005

[17] B. Scattergood, “The semantics and implementation of machine-readable CSP”, 1997. Ph.D. thesis, University of Oxford, .

[18] D. Dolev and A. C. Yao, “On the security of public key protocols”, IEEE Trans. on Information Theory, 29(2), 1983, pp. 198-208.

[19] A. W. Roscoe and M. H. Goldsmith, “The perfect spy for model-checking crypto-protocols”, In Proceedings of DIMACS workshop on the design and formal verification of crypto-protocols. 1997.