Modelling, Verification, and Formal Analysis of Security Properties in a P2P System

Sam Sanjabi, Franck Pommereau

Modelling, Verification, and Formal Analysis of Security Properties in a P2P System

Sam B. Sanjabi and Franck Pommereau
Laboratoire d'Algorithmiques, Complexité, et Logique (LACL)
Université de Paris-Est
Créteil, France
sam.sanjabi@gmail.com
pommereau@univ-paris12.fr

ABSTRACT

We present a security analysis of the SPREADS[1] *system, a distributed storage service based on a centralized peer-to-peer architecture. We formally modelled the salient behavior of the actual system using ABCD, a high level specification language with a coloured Petri net semantics, which allowed the execution states of the system to be verified. We verified the behavior of the system in the presence of an external Dolev-Yao attacker, unearthing some replay attacks in the original system. Furthermore, since the implementation is also a formal model, we have been able to show that any execution of the model satisfies certain desirable security properties once these flaws are repaired.*

KEYWORDS: Privacy Protection for Collaborative Systems, Security for Speecific Collaboration Domains (P2P), Security in Collaborative Multi-Agent Systems, Secure Collaborative Agents, Middleware Security.

1. INTRODUCTION

The SPREADS system is designed to provide a distributed storage service to its users, and functions using a peer-to-peer (P2P) backbone in which each node can act as both a client which requests operations – reading data, writing data, or deleting data – and as a storer which provides these operations to other nodes. Like other P2P systems, this functionality can be provided in one of two ways:

1. A *centralized* architecture in which the nodes all speak to a distinguished server, which collects and verifies the meta data required for the nodes to co-ordinate their actions.
2. A *decentralized* architecture in which the needed meta information storage and verification services are themselves spread across the nodes.

The analysis in this paper assumes a centralized system, as that is currently the state of the SPREADS implementation. Despite the name, the "centralized" system is only so in the sense that there exists a single server which must mediate some handshaking between peers, the bulk of network communication in such a system takes place between the peers themselves. We discuss the implications of conducting similar research on a decentralized system, which fully distributes the central service, in section 6. In conjunction with the SPREADS programmers, we identified a set of desirable properties to which the system should conform:

- The **termination** of the operations, while a part of their definition, should be guaranteed or at least quantified by modelling.
- The **integrity** of the data stored on the system should be preserved in the presence of malicious nodes, i.e. attackers should not be able to modify data during an operation.
- The **confidentiality** of the data should be guaranteed: only the owner of a file should be able to read it.
- The **authentication** of peers should be checked, i.e. each communication should be accompanied by a proof of the identity of the sender.

[1] Safe P2P-based REliable Architecture for Data Storage, France, ANR Telecom 2007. The project is a collaboration between the UbiStorage company (Amiens), the LACL Lab. of Univ. Paris East, the INRIA/LIP6 REGAL project team (Paris), the INRIA/I3S MASCOTTE project team (Sophia Antipolis) and the EURECOM NS Team (Sophia Antipolis). See also `http://www.spreads.fr/`.

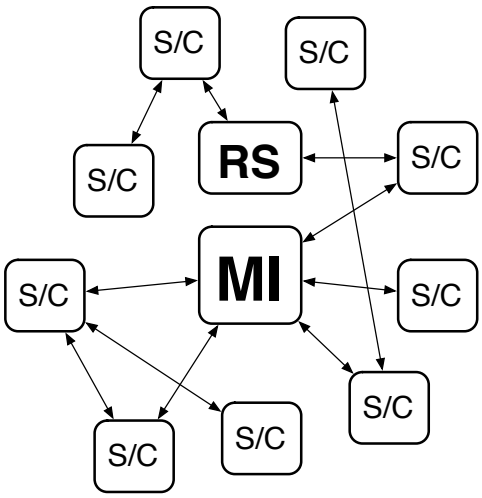

Figure 1. The SPREADS system

- The system's resistance to **denial of service** (DoS) attacks must be quantified.

In the current centralized implementation of SPREADS, some of these properties are more or less trivial to model and verify. Termination is guaranteed in all but the most extreme cases of failure, while the system is obviously susceptible to (DoS) attacks as flooding the central server would cause a halt in the entire network. In order to avoid similarly trivializing the pure security properties (integrity, confidentiality, and authentication), we began our modelling under the assumption that all communications between agents are encrypted and signed using a perfect asymmetric cryptosystem.

Our presentation begins with a description of the centralized SPREADS system in section 2, and its modelization in ABCD in section 3. The latter also describes the usage of the model checker, namely how a model is defined and how its execution states are examined. Section 4 presents two replay attacks found when verifying this initial model, and the modifications made to it in order to prevent them. We then proceed to conduct a general analysis of our model in section 5. This analysis amounts to a proof that no execution of the model allows an external attacker to learn any private data. While fairly tedious, this approach has the advantage of not being dependent on a static scenario to which an execution of the model is necessarily bound: the result applies to all such scenarios. Section 6 concludes and discusses future work.

2. THE SPREADS SYSTEM

The centralized system comprises a multitude of identical peers which act as both a client which requests data storage or retrieval operations from the network, and also as a storer which provides these services to the other peers. In order to distribute a file across the nodes and to minimize errors, a data fragmentation and redundancy server using Reed-Solomon codes [8] takes a given file and returns the fragments to be stored across the nodes. Finally, the necessary meta information required to co-ordinate the network operations is kept on a central server with which all the nodes must communicate. While each node must communicate with the two distinguished servers during certain operations, they must also conduct exchanges with one another to actually retrieve and store data fragments.

This section presents the primary communication exchanges undertaken by the agents during the three client operations. For clarity, we eschew many of the details of the information passed in the actual implementation of the system and focus solely on those relevant to the abstractions we used when modelling it, as well as details that are relevant to the security properties discussed in later sections. Throughout the description, we use the naming conventions in Figure 1 to refer to the agents: the client process C, the storer processes S, the meta information server MI, and the Reed-Solomon server RS.

We begin with the data storage operation `PUT`. This takes as input the data D which a client wishes to store. A successful execution of the protocol, initiated by the client wishing to store the data on the network, proceeds as follows:

PUT(D)

$$
\begin{array}{rcccl}
1: & C & \longrightarrow & RS & : D \\
2: & RS & \longrightarrow & C & : [frag_1(D), \ldots, frag_n(D)] \\
3: & C & \longrightarrow & MI & : \texttt{GET\_NEW\_KEY} \\
4: & MI & \longrightarrow & C & : K \\
5: & C & \longrightarrow & MI & : \texttt{GET\_STORERS}, len(D) \\
6: & MI & \longrightarrow & C & : [S_1, \ldots, S_n] \\
7: & C & \longrightarrow & S_i & : K, frag_i(D) \\
8: & S_i & \longrightarrow & C & : \texttt{FRAG\_STORED} \\
9: & C & \longrightarrow & MI & : K, [S_1, \ldots, S_n] \\
10: & MI & \longrightarrow & C & : \texttt{META\_STORED}
\end{array}
$$

The client first sends its data to the Reed-Solomon server, which returns n fragments to be stored. The nature of this fragmentation includes some redundancy so that some subset of them is all that is required to again reconstruct the file, but the exact details of how this is done and the parameterization of just how many fragments are returned is irrelevant to the discussion at hand. For simplicity, our model assumes $n = len(D)$. Notice also that the RS server is in

practice duplicated onto each node in order to avoid network communication. Once it obtains the fragmented data, the client requests a key from the central server. This is not a cryptographic key but merely an identifier associated to the data block that is to be stored (essentially a file identifier). In step 5 the client requests a list of storers from the MI server, quantified by the length of the data block D. The server therefore returns the names of enough storers in which the client may store the fragments it obtained in step 2.

At this point, the parallel behavior of the protocol begins in earnest: the client must send each fragment to its corresponding server along with the key K to which that fragment is associated, and await an acknowledgement. So, steps 7 and 8 are executed for $1 \leq i \leq n$ in parallel. Upon successfully receiving the fragment, each server associates it to the provided key in its local database and sends the client a confirmation that the operation was carried out successfully. The entire `PUT` operation is only successful if "enough" such acknowledgements are received, with the desired threshold determined by the redundancy parameters required by the system operator. In our model, we simply assume that the operation is successful if all of the storage requests were successful, one branch in the state graph is created for each possibility.

Finally, if sufficiently many fragments are stored, the client proceeds (in step 9) to send the MI server the list of storers in which the fragments associated with K now reside. The latter stores this information, acknowledges having done so, and the operation terminates.

The data retrieval operation, `GET`, is considerably simpler. Being a read-only operation, it does not require any upkeep of meta information. A successful run of the protocol is described by the following:

GET(K)

$1:$	C	$\longrightarrow$	MI	: `GET_STORERS`, K
$2:$	MI	$\longrightarrow$	C	: $[S_1, \ldots, S_n]$
$3:$	C	$\longrightarrow$	S_i	: K
$4:$	S_i	$\longrightarrow$	C	: $frag_i(D)$

The requesting client asks for the list of storers associated with the file identifier K that the client would like to retrieve, the client then sends this identifier to each of the storers in the returned list, each of which in turn returns the fragment they've stored for that particular key. So, steps 3 and 4 are executed for $1 \leq i \leq n$ in parallel. As long as sufficiently many fragments are returned to the client, the original datablock D can be reconstructed. In principle, the deletion operation `FREE` could be implemented in a similar fashion. However, the following protocol holds some advantages over this design:

FREE(K)

$1:$	C	$\longrightarrow$	MI	: `FREE`, K
$2:$	MI	$\longrightarrow$	C	: `KEY_MARKED`

$1:$	S_i	$\longrightarrow$	MI	: `FREE_POLL`
$2:$	MI	$\longrightarrow$	S_i	: $[K_1, \ldots, K_m]$

When a client wishes to delete a file K, he informs the central server of this. At this point the server marks the file as deleted *without actually deleting the data*. In the mean time, each storer periodically polls the central server asking if there have been any deleted keys associated to them. If so, the server sends a list of such keys and the fragments associated to them are physically deleted by the storer. This design implements a delay between the deletion request and the actual deletion of the data, allowing the network operator to potentially undo a deletion in the interim if the client requires it.

This then was the starting point of our modelization. The next section introduces our modelling language, the abstractions and assumptions taken in our work, and the attacker model.

3. THE ABCD MODEL

ABCD (Asynchronous Box Calculus with Data, [13]) is a specification language that allows its users to express the behavior of concurrent systems at a high level. A specification is translated into colored Petri nets, which can then be model checked. An ABCD compiler is packaged with the distribution of SNAKES [12, 11] so we directly used SNAKES as a model checker also (although it is more dedicated to quick prototyping and provides low performance on state space computations).

In particular, the ABCD meta syntax allows its users to define complex processes in an algebra that allows:

- Sequential composition $P; Q$
- Non-deterministic choice $P + Q$
- Iteration $P * Q \equiv Q + (P; Q) + (P; P; Q) + \ldots$
- Parallel composition $P|Q$

Processes are built on top of atoms comprising either named sub-processes, or (atomic) *actions*, i.e. conditional accesses to typed buffers. Actions may produce to a buffer, consume from a buffer, or test for the presence of a value in a buffer, and are only executed if the given condition is met. The semantics of an action is a single transition in a

Petri net which appropriately produces or consumes tokens to or from places implementing buffers.

For a description of the syntax and semantics of ABCD, as well as an illustrative example, please consult [13, sec. 3.3]. As a very basic example, consider the following producer/consumer example:

```
buffer shared : int = ()
buffer count : int = (1)

[count-(n),count+(n+1),shared+(n)]
                  * [False] |
[shared-(n) if n % 2 == 0]
                  *[False]
```

The two first lines define two buffers, allowing to store integer values; the first one is initially empty while the second one holds the single value 1. Then, a composition of four actions (each enclosed in square brackets) defines the process part of the specification. The `[False]` action is one which can never be executed. Others include accesses to buffers and an optional guard. The '−' operation on a buffer attempts to consume a value from it and bind it to the given variable, scoped to the current action. The language also supplies a read-only version '?', thus `count?(n)` will read a value from `count` into variable `n` without removing it from the buffer. Similarly, the '+' operation attempts to write a value to the buffer, and there are also flush (>>) and fill (<<) operations which perform writes into and reads from sets respectively. The first component of the parallel composition above therefore continuously populates the buffer named `shared` with increasing integers. The second sub-process just pulls the even ones out of the `shared` buffer. The language also allows its users to name valid processes into a `net` declaration and instantiate them repeatedly.

In order to model check a system specified in ABCD, a directed graph is constructed in which each node represents a marking of the Petri net generated by the specification, and each edge represents the execution of a single transition in the Petri net. This graph can therefore be searched for the presence or absence of a bad marking, i.e. one that invalidates a desired property of the system. Note that for the example shown above this process would not terminate because the state graph is infinite, therefore such explicit state graph can be used only for system with finitely many markings. Otherwise, solutions exist to handle symbolically infinite state graphs, usually through approximations or abstractions (e.g., [10, 7, 14]).

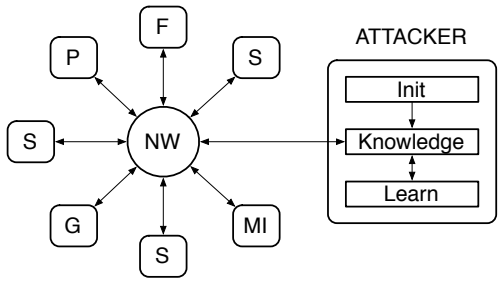

Figure 2. Our ABCD model of SPREADS. The central server (MI), storers (S), clients performing PUT (P), GET (G), or FREE (F) operations, and the attacker are modeled as separate agents communicating through a distinguished global buffer place NW.

3.1. Structure of the Model

Our model of SPREADS consist of approximately 450 lines of ABCD code and just over 600 lines of code in Python which defines various types and support functions used in the model. (Indeed, SNAKES' implementation of ABCD actually works with Python-coloured Petri nets.) The ABCD specification defines a number of agents (i.e. `nets`) which emulate the functional behavior of the system:

- Each of the operations `PUT`, `GET`, and `FREE` is implemented as its own `net` which plays the client side of the protocol.
- The storer process is implemented as it's own `net` which can respond to the client read and write requests, but can also at any time poll the central server.
- The central meta-information server is implemented as a `net` which responds to requests for keys, lists of storers, polls from storer processes, or meta data updates from `PUT` and `FREE` agents
- An attacker process which observes all communications, stores any information it has learned into a local knowledge buffer, and attempts to replay any valid packets it learns. This agent will be further detailed below.

Communication between agents is handled through a distinguished buffer place `nw` in the Petri net representing the network (Figure 2), and is assumed to be signed and encrypted using a perfect asymmetric cryptosystem. The encryption is symbolic, and it is assumed that every agent A knows its own private key $Priv(A)$, and knows the public keys $Pub(-)$ of all agents. Therefore, when agent A

wants to send a message D to agent B, he simply adds the following tuple to the `nw` buffer:

$$(A, B, \{\{D\}_{Priv(A)}\}_{Pub(B)})$$

This is "received" by B, who is listening for packets with his name as a target and encrypted with his own public key. The attacker may only learn the contents of an encrypted packet if it first learns the component of the key required to decrypt it.

Additionally, the following elements of the actual system are abstracted in the model in order to simplify the implementation and to stay within the constraints of what ABCD allows:

- The Reed-Solomon fragmentation is not modelled. Instead, when a client attempts to write a data block D, the MI server just returns a list of $len(D)$ storers.
- The parallelism inherent in steps 7 and 8 of the `PUT` protocol, and steps 3 and 4 of the `GET` protocol is modelled iteratively. ABCD does not directly support the dynamic creation of parallel processes, therefore clever agent identifiers and a loop are used to model communications between clients and storers. Specifically, the clients iterate through the list of storers and place their outgoing packets onto the `nw` buffer one by one before again retrieving the responses from the same buffer iteratively.

Our design then allows the user to specify a static scenario to be model checked. For example, the user may use our pre-defined nets to construct (following the naming conventions in Figure 2) a process such as:

$$\text{P("DAT")} \mid \text{S}_1 \mid \text{S}_2 \mid \text{S}_3 \mid \text{MI} \mid \text{Attacker}$$

Which will model the scenario in which a client attempts to do a `PUT` of the string "DAT" into three storers. The user only needs to make sure that a few coherency conditions are met, for example that at least three storers are available in his scenario if a client attempts to write data of length 3. The process is then compiled into a Petri net by the ABCD compiler, and the corresponding state graph can be searched using the SNAKES library for a bad marking – say for example one in which one of the agent's private keys appear in the attacker's knowledge buffer.

3.2. The Attacker

The final agent definable in our model is the attacker, which continually observes the communications on the network. As shown in Figure 2, the attacker has three components: a buffer named `knowledge` which is essentially a set of the information that the attacker currently "knows", a set of initial knowledge, and a learning engine with which it uses to glean new knowledge from what it observes on the network. Intuitively, the attacker performs the following operations:

1. It intercepts each message that appears on `nw` and adds it to its knowledge
2. It passes each message along with its current knowledge to the learning engine and adds any new knowledge learned to it's current knowledge
3. It then may either do nothing, or take any message that is a *valid message in the protocol* that is contained in its knowledge and put it back on `nw`

In ABCD, these actions are expressed by the following term:

```
[nw-(m), knowledge>>(k),
         knowledge<<(learn(m,k))];
([True] + [knowledge?(x), nw+(x)
                        if message(x)])
```

The two first lines implement steps 1 and 2: a message `m` is removed from the network, and this message is passed to a method `learn()` along with the contents of the current knowledge. The return value of this method is filled back into the `knowledge` buffer. The next lines implements step 3: the process can either choose to do the empty action `[True]`, or to replay any element of its knowledge that satisfies the `message()` predicate – which checks if `x` is a valid protocol message – back on the network. Note that a branch is created in the state graph for each message that can be intercepted in the first line, another for the choice in the second line, and another for each valid message in the knowledge. This is why the attacker is the most computationally intensive component of our modelling.

It remains to discuss the actions of the `learn()` operation. This function takes as input a message m and a set k. From this input it increases the size of the set k by applying the following operations:

1. it adds m to k if it did not already exist
2. if m is an encrypted datum $\{m'\}_K$ and the public/private counterpart to K allowing it's decryption is in k, and m' is not in k, then it adds `learn`(m', k) to k
3. if m is a tuple $(m_1, \ldots, m_n)$ and m_i is not in k, then it adds `learn`(m_i, k) to k for each such m_i
4. it augments k with all tuple combinations constructible from the contents of k, as well as the encryption of these with any keys k. It then adds `learn`(m', k) to k for each message m' so added.

Once each of these steps are completed, the `learn()` method returns the final k. This description is essentially an inductive definition of the Dolev-Yao spy process defined in [6]. However, note that step 4 is infinite: for instance any element can simply be repeated in a tuple of arbitrary size. For this reason, in our implementation, we add the additional restriction that any elements added to k must be valid fragments of messages in the SPREADS protocol model. We formally define a slight abstraction of this method in section 5.

One of the advantages of this design is that the attacker model is partially parameterized by the knowledge it is initially given. By giving it only public information (agent identifiers, public keys, etc.), it behaves as a malicious agent external to the system. We can also model the attacker as one of the nodes themselves by simply giving it the private information it requires to identify itself as one, or more specifically everything it needs to "play" as a legitimate agent in some execution trace in the state graph. We similarly model compromised keys, file identifiers, or any other private information.

4. SECURITY FLAWS DETECTED

Despite the relatively simple structure of the model, and the fact that all the communications are encrypted and signed, it turns out that an external attacker can still exploit certain replay attacks on the system. This section discusses three such attacks that we found during the development and verification of our model. We also discuss the changes (if any) that needed to be made to the model in order to prevent these attacks.

4.1. Intra-Session Attacks

The first type of error occurs when a client confuses one type of data with another within a single operational session. For instance, consider the first two steps of the `PUT` protocol (omitting the interaction with the Reed-Solomon server because it is not modelled):

$$\begin{array}{llcll} 1: & C & \longrightarrow & MI & : \texttt{GET\_NEW\_KEY} \\ 2: & MI & \longrightarrow & C & : K \end{array}$$

This second message can be intercepted and replayed by the attacker posing as the meta information server. It can do so after the client request for a list of storers:

$$\begin{array}{llcll} 3: & C & \longrightarrow & MI & : \texttt{GET\_STORERS}, len(D) \\ 4: & A(MI) & \longrightarrow & C & : K \end{array}$$

This message has the exact same structure as the returned list of storers, and since the modelization is symbolic, the client has no way to tell that the data it received was in fact a list. In the real system, this situation can occur since the message is just a string of bits interpreted by the client. Since the string may not actually represent a list, it may lead to a failure of a legitimate `PUT` session due to a replay by an external attacker. This is actually how we discovered this attack: during the computation of the state space of a simple scenario, a run-time error was raised by SNAKES because it attempted to iterate over a key instead of a list.

This exploit can be prevented by giving every message sent in the steps of the three protocols a distinguished type: in the particular example above, simply an identifier stating that K is indeed a file ID in step 2, and that the returned message in step 4 is a list of storers. This will allow the client to check if the data he is receiving is indeed of the expected type. Some of the messages in the protocols already have these types (e.g. `GET_NEW_KEY`), it suffices simply to augment each "arrow" in the protocol with a unique type. To illustrate, the `GET` protocol becomes:

$$\begin{array}{llcll} \textbf{GET}(K) & & & & \\ 1: & C & \longrightarrow & MI & : \texttt{GET\_GET\_STORERS}, K \\ 2: & MI & \longrightarrow & C & : \texttt{GET\_RET\_STORERS}, [S_1, .., S_n] \\ 3: & C & \longrightarrow & S_i & : \texttt{READ\_FRAG}, K \\ 4: & S_i & \longrightarrow & C & : \texttt{RET\_FRAG}, frag_i(D) \end{array}$$

The prefix `GET_` in the types of the first two messages serve to distinguish them from the corresponding messages in the `PUT` protocol, preventing a similar error from occurring between multiple sessions.

4.2. Inter-Session Attacks

Once the protocol is fully typed, the attacker can still replay packets from earlier sessions in order to create inconsistencies. For example consider the typed key return message in the `PUT` protocol:

$$\begin{array}{llcll} 1: & C & \longrightarrow & MI & : \texttt{PUT\_GET\_NEW\_KEY} \\ 2: & MI & \longrightarrow & C & : \texttt{PUT\_RET\_NEW\_KEY}, K \end{array}$$

The second message can be replayed in a later `PUT` session, leading the client to mistakenly re-use an old file identifier and therefore overwrite existing data. This problem has been discovered by analysing a scenario with two `PUT` operations that did not yield a consistent distribution of the initially sent data onto the storers. One way to solve this particular problem is to simply disallow overwrites on the storers (SPREADS in fact does this). While this resolves the serious integrity issue displayed above, it does nothing about the general problem that messages from earlier sessions replayed by the attacker will be accepted by legitimate agents. For example, replaying storer lists or storer acknowledgements can easily lead to externally caused errors.

The correct solution is to require each protocol initiator – thus clients attempting to do a PUT, GET, or FREE, as well as a storer attempting to poll the central server – to generate a fresh session identifier at the beginning of each operation. This identifier must then be passed with all the messages in that session in order to avoid confusing them with messages from earlier runs.

4.3. Internal Attacks

The final attack type we discovered is one that applies even after session identifiers and types are added to the protocol. Namely, by having the attacker play as a client in the system, it becomes evident that there isn't anything that actually connects a file identifier to a user. For instance, suppose Alice does a PUT of her private information, which gets stored in the network under key K. If K becomes compromised, any other client can retrieve this data by doing an ordinary GET. This is fairly obvious by simply examining the GET protocol, but was actually discovered by providing the attacker enough initial knowledge to play the exchange himself in addition to K.

The solution adopted by SPREADS to counter this problem is to use file identifiers that are not easy to generate randomly (to prevent them from being guessed by other peers). This therefore requires no modification to our modelling, since we handle all data symbolically.

However, if a malicious node stores data, it knows all the file identifiers it has learnt during storage requests. So it may retrieve all the associated data. In the current centralized implementation, this problem can be solved by associating on the server each file identifier to the identifier of the node that owns the data. Since every communication is authenticated, the server can detect an invalid request in the first message of a GET.

5. THEORETICAL ANALYSIS

We now detail a more general analysis of the model. In particular, we show that no scenario executed in our model will lead to an unencrypted data fragment appearing in the attacker's knowledge. Note that this result is about the model only, and not about the system itself – i.e. it comes with all the usual caveats about abstractions and modelling errors that come with usual model checking results. In fact it is only possible because of the severe restrictions placed on communications in SPREADS, and the fact that our ABCD specification is not only the front-end of a model-checker, but also a formal model.

Recall from section 3 that each communication by an agent in our model is done by placing a tuple of the form $(A, B, \{D\})$ on the central network place, where where A and B are agent identifiers and $\{D\}$ is a signed and encrypted data block of the form $\{\{X\}_{Priv(A)}\}_{Pub(B)}$. This formally means that the Petri net semantics of our agents only have output arcs to the network buffer that contain such tuples, and this fact can simply be established by examining the code. The only other arc that enters the network place is that from the attacker's knowledge buffer, but that arc is also guarded by a conditional requiring that the only packets that are valid protocol messages (and thus tuples of this form) may fire the transition leading to it. Therefore we can combine these properties to deduce the following:

Fact 1. In any execution of our SPREADS model, the network place nw will only contain tuples of the above form in any reachable marking.

In order to prove a result about the contents of the knowledge buffer, we in fact need to know a bit more about the structure of valid protocol fragments. Luckily, by simply examining all of the arcs entering nw from valid agents in the model's code, we can also assert the following:

Fact 2. For any valid SPREADS model communication tuple $(A, B, \{D\})$, the unencrypted contents of the payload $\{D\}$:

(a) does not contain any agent identifiers

(b) does not itself contain any further encrypted data

We now define a coherency condition on the contents of the knowledge buffer: showing that this condition is an invariant through any execution of the learning engine is the central idea of our argument.

Definition 1. A knowledge set K is called *SPREADS safe* if it contains only valid SPREADS protocol packets, agent identifiers, public keys of agent identifiers, and encrypted elements $\{D\}$ of the form defined by facts 1 and 2 above. In addition, K must satisfy the following closure properties:

1. if an agent identifier A is in K, then so is $Pub(A)$
2. if agent identifiers $A_1, \ldots, A_n$, and payload $\{D\}$ are in K, then so are all tuples of the form $(A_i, A_j, \{D\})$

We are now ready to discuss the main analysis. We know that the only inputs into the attacker's knowledge come from nw, the initial knowledge, and the input from learn. The input from nw contains only valid protocol packets by fact 1, the input from the initial knowledge is static and will contain only the identities of all the agents in the scenario and their public keys (as we are modelling an external

learn(msg,K)
1: $K \oplus \{msg\}$
2: if $(can\_decrypt(msg, K))$ then
3: $K \oplus \mathbf{learn}(decrypt(msg), K)$
4: if $(is\_tuple(msg))$ then
5: foreach m in msg do
6: $K \oplus \mathbf{learn}(m, K)$
7: $K \oplus fragments(K)$
8: foreach m in $fragments(K) \setminus K$ do
9: $K \oplus \mathbf{learn}(m, K)$
10: return K

Figure 3. Pseudo-code for our implementation of the the learning engine. The $S_1 \oplus S_2$ operation signifies updating the contents of set S_1 with the elements of set S_2 (i.e. a union and an assignment)

attacker). We must therefore reason about the learning algorithm, which is presented in Figure 3.

The function takes a message and a knowledge set and produces an updated knowledge set. Step 1 simply adds the message to the set. Steps 2-3 determines if the message can be decrypted given the current knowledge, the $can\_decrypt()$ predicate is true if msg is an encrypted datablock $\{m\}_k$ and the key required to decrypt it is in K. If true the algorithm recurses on the decrypted contents m and updates the knowledge with the results. Steps 4-6 similarly determines if msg is a tuple, and if so recurses on each component of the tuple, updating K with each one. Steps 7-9 are the most complex: $fragments(K)$ is the set of all combinations and encryptions of elements of K which are valid fragments of protocol messages. The encryptions are only allowed using keys that are contained in the current knowledge, which is then updated with the contents of this set. The algorithm then recurses on each *new* element found using this procedure. The final K is then returned in the last step.

The keystone to our result is showing that, given a valid SPREADS message msg, and a SPREADS safe K, this algorithm returns a SPREADS safe set:

Theorem 1. *If msg is a SPREADS communication tuple $(A, B, \{D\})$ with $\{D\}$ defined as above, and K is SPREADS safe and contains all agent identifiers, then $\mathbf{learn}(msg, K)$ is also SPREADS safe.*

Consider a packet $(A, B, \{D\})$, and suppose in the worst case that it is not already contained in K. This packet is added to K in the first step.Furthermore, the packet is a tuple therefore the algorithm simply decomposes it and recurses on each of its components in turn. The first such recursion is on A, which by assumption is already contained in K, can't be decrypted, and isn't a tuple. Also, no new elements are added through recomposition, this is because:

- No elements can be encrypted to obtain fragments because by definition and fact 2(b) only public key encryptions of private key encryptions are valid fragments. Since K is SPREADS safe, it contains no private keys nor elements who's outermost encryption is done with a private key, therefore no valid fragments can be constructed by encryption.
- Since A was already in K at call time, all fragments of the form $(A_i, A_j, \{D\})$ containing A as source or target in K are already present by SPREADS safety.
- By facts 2(a) and (b), no other tuples composed of agent identifiers or encryptions can form valid fragments as these cannot appear inside the encrypted portion of a valid packet.

An identical argument occurs for the recursion on B. This leaves only the recursion on $\{D\}$. At worst case this is added to K in the first step, but cannot be decrypted because K cannot contain private keys by safety, nor is it a tuple and thus can't be decomposed. Now, since $\{D\}$ was not in K, we know that all tuples of the form $(A_i, A_j, \{D\})$ are valid fragments in $fragments(K) \setminus K$, and thus are added to K (note that this now makes K once again SPREADS safe). The recursion on each of these elements does not learn any data because we just established that all $(A_i, A_j, \{D\})$, and $\{D\}$ itself, have already been added to K and thus we can simply repeat the argument to conclude that none of the recursions in step 9 of the subrecursion on $\{D\}$ add anything to K. Finally, the argument in this last step can be repeated to conclude that the recomposition step in the outermost call to **learn** also learns nothing, and therefore the SPREADS safe knowledge computed in step 6 is the one that is returned by the function, completing the proof.

Recall that the initial knowledge when modelling an external attacker contains only the identifiers of all the agents in the scenario, as well as their public keys. Therefore the initial knowledge is SPREADS safe. Furthermore, the **learn** function is only called by the attacker on a message taken from the network (`nw`) and thus is a valid SPREADS packet by fact 1. These two observations lead to the following direct corollary to the above theorem:

Corollary 1. *During the execution of any SPREADS scenario modelled with an external attacker, there exists no marking in which the attacker's knowledge contains an unencrypted data fragment or a private key.*

This follows immediately from the definition of SPREADS safety and the implementation of the model.

6. RELATED AND FUTURE WORK

We've described the modeling of a P2P system in ABCD, and have used this model in two very different ways to both find security flaws in, and establish security properties of the system. In the first case, we actually generated a state graph using the Petri net semantics of our specification and discovered a number replay and internal attacks against the system. In the second case, we used the restrictions placed on communications in the system to show that no execution of our model on a valid scenario modelling the system will result in a state violating certain security properties.

While the types of analysis we've been able to conduct vary starkly in style, they both have pros and cons, and both bring different value to the system developers. Model checking by execution is able to discover attacks exploited by replaying and by internal attackers, both of which the general approach would have considerably more difficulties with as the interactions between the system agents become far more complex and the corresponding proofs immeasurably more difficult. On the other hand, the general analysis yields results that are independent of a particular scenario and therefore have considerable value in giving assurances to SPREADS' users that the system satisfies certain information security properties that they would expect.

At the heart of both of these approaches is the actual ABCD specification, which was used as both an executable program and a formal object. In fact, we envision using these specifications in later iterations of SPREADS' design to play more roles:

1. **Simulation.** Our ABCD specifications can be themselves provided as *input* to a simulation engine (rather than simulating the actual system). The idea is that executing Petri net traces would allow us to monitor the behavior of SPREADS with very large numbers of peers, allowing some quantitative analysis of the system's behavior with respect to variables like high rates of churn.

2. **Design.** While ABCD is useful in establishing the desired properties of the live centralized SPREADS system, it can also be used to model the behavior of the fully distributed system based on the current design ideas of the developers. Such modelization work could potentially influence the future system's design and implementation, and we plan very much to perform a similar analysis on the fully distributed system.

If successful, we will have a system that was designed, model checked, simulated, and formally analyzed around the exact same specification, strengthening the ties between many aspects of the software development process that are often at odds.

6.1. Related Work

Our approach can in some ways be seen as straddling between more temporal logical approaches such as CTLK [5] and pure process algebras such as the asynchronous π-calculus [9]. ABCD offered the "best of both worlds" for our particular problem: its process algebra syntax allows models to be easily defined (compared to temporal logics) and their properties to be checked directly via marking graphs. On the other hand, the structured nature of its Petri net semantics allowed us to verify systems with nondeterministic choice and iteration – both essential to constructing an accurate model of SPREADS – which often lead to intractable (or worse) model checking problems in pure process algebras.

Similar works include the specification and analysis of simpler security protocols using a Petri net semantics of the *Security Protocol Language* (SPL [4]): [3, 2]. SPL is not suitable to model SPREADS' protocol because it lacks features that turned out to be crucial for our modelling: in particular, the ability to model loops and choices in conjunction with the annotation of the model with complex data types and associated functions. (Remember that our model comprises more than 600 lines of Python code.) A similar comparison could be drawn with approaches like AVISPA [1]: on the one hand, the symbolic approach allows for more efficient verification than our explicit approach; but on the other hand, the control flow in these tools is generally limited to parallel and sequential compositions. Furthermore, it is often not possible to introduce user-defined data types and the functions into such tools.

We believe that rich control flow and an extensible data domain are two crucial features to model complex protocols. In particular:

- protocols or scenarios that involve complex agent behaviours require more than sequential and parallel compositions. For instance, branching protocols (like optimistic fair contract signing schemes) are awkward if not impossible to model without a choice; similarly, a form of repetition (loop or recursion) is required to model behaviours like communicating with a dynamically created list of agents;

- by finding the right mixture of modelling and implementation, one can generally achieve a satisfactory abstraction that leads to a reduced state space, overcoming the inherent limitations of explicit model-checking.

REFERENCES

[1] "Automated Validation of Internet Security Protocols and Applications", Available: http://avispa-project.org.

[2] R. Bouroulet, R. Devillers, H. Klaudel, E. Pelz, and F. Pommereau, "Modeling and analysis of security protocols using role based specifications and Petri nets", International Conference on Applications and Theory of Petri Nets, Xi'an, China, Vol. 5062 of LNCS, pp. 72–91, 2008.

[3] R. Bouroulet, H. Klaudel, and E. Pelz, "Modelling and verification of authentication using enhanced net semantics of SPL (Security Protocol Language)", International Conference on Application of Concurrency to System Design, Turku, Finland, pp. 179–188, 2006.

[4] F. Crazzolara and G. Winskel, "Events in security protocols", ACM Conference on Computer and Communication Security, Philadelphia, PA, pp. 96–105, 2001.

[5] C. Dima, "Revisiting satisfiability and model-checking for CTLK with synchrony and perfect recall", International Workshop on Computational Logic in Multi-Agent Systems, Dresden, Germany, Vol. 5405 of LNCS, pp. 117–131, 2008.

[6] D. Dolev and A. Yao, "On the security of public key protocols", *IEEE Transactions on Information Theory*, Vol. 29, No. 2, pp. 198–208, 1983.

[7] H. Klaudel, M. Koutny, E. Pelz, and F. Pommereau, "An approach to state space reduction for systems with dynamic process creation", International Symposium on Computer and Information Sciences, Cyprus, pp. 1–6, 2009 (to appear).

[8] R. Kotter, "Fast generalized minimum distance decoding of algebraic-geometry and reed-solomon codes", *IEEE Transaction in Information Theory*, Vol. 42, No. 3, 1996.

[9] U. Montanari and M. Pistore, "Finite state verification for the asynchronous π-calculus", International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Amsterdam, The Netherlands pp. 255–269, Springer-Verlag, 1999.

[10] L. Petrucci, "Combining Finkel's and Jensen's reduction techniques to build covering trees for coloured nets", *Petri Nets Newsletter*, number 36, 1990.

[11] F. Pommereau, "Snakes is the net algebra kit for editors and simulators", Available: http://pommereau.blogspot.com.

[12] F. Pommereau, "Quickly protoyping Petri nets tools with SNAKES", International Conference on Simulation Tools and Techniques for Communications, Networks and Systems, Brussels, Belgium pp, 1–10, 2008.

[13] F. Pommereau, "Algebras of Petri nets", habilitation thesis, University Paris-East, Créteil, France, 2009.

[14] F. Pommereau, R. Devillers, and H. Klaudel, "Efficient reachability graph representation of Petri nets with unbounded counters", International Worshop on Verifications of Infinite-State Stystems, Lisbon, Portugal, Vol. 239 of ENTCS, pp. 1–10, 2007.