Minimal Backups of Cryptographic Protocol Runs

Jay A. McCarthy*
Brigham Young University
Provo, UT
jay@cs.byu.edu

Shriram Krishnamurthi
Brown University
Providence, RI
sk@cs.brown.edu

ABSTRACT

As cryptographic protocols execute they accumulate information such as values and keys, and evidence of properties about this information. As execution proceeds, new information becomes relevant while some old information ceases to be of use. Identifying what information is necessary at each point in a protocol run is valuable for both analysis and deployment.

We formalize this necessary information as the *minimal backup* of a protocol. We present an analysis that determines the minimal backup at each point in a protocol run. We show that this minimal backup has many uses: it serves as a foundation for job-migration and other kinds of fault-tolerance, and also assists protocol designers understand the structure of protocols and identify potential flaws.

In a cryptographic context it is dangerous to reason informally. We have therefore formalized and verified this work using the Coq proof assistant. Additionally, Coq provides a certified implementation of our analysis. Concretely, our analysis and its implementation consume protocols written in a variant of the Cryptographic Protocol Programming Language, CPPL.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Formal methods; D.2.5 [**Testing and Debugging**]: Diagnostics; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Miscellaneous

General Terms

Languages, Reliability, Security, Theory, Verification

Keywords

Cryptographic protocols, CPPL, Strand Spaces, Coq

*Work completed at Brown University.

FMSE'08, October 27, 2008, Alexandria, Virginia, USA.

1. PROBLEM AND MOTIVATION

Each cryptographic protocol has a goal. This goal is conveniently expressed as data and *properties* about that data. For example, a two-party key agreement protocol's goal might be a key (the data) that is shared only between two parties (the property). This property expresses the relationship between three pieces of data: the key and the identities of the two parties.

Each step of a protocol can be seen as advancing towards the ultimate goal. Some steps provide necessary data. Some steps provide *evidence* that a property is true. And other steps serve to provide these two kinds of information to other parties.

The evidence of properties *may not be* explicitly in the protocol. Consider an authentication test [7]: protocol role P receives a nonce[1] that uniquely originated in a message sent by P encrypted with Q's public key. That P received the nonce back enables P to conclude that Q received and acted on the message (provided Q's private key is not compromised). The evidence of this property is not explicitly in the protocol. These kinds of properties are often checked in formal verification models, such as strand spaces [22].

The evidence of other properties *are* explicitly in the protocol, in the form of rely-guarantee *formulas* [8]. To use this technique, each message transmission is annotated with formulas that express properties about the transmitted values, and each message reception is annotated with formulas that express properties about the received values. Formulas on transmission *must be* guaranteed, or checked, by the transmitting party, while formulas on reception are assumed to be true. Such a protocol is sound if, in every execution, for every reception, the assumed formulas are guaranteed earlier in the execution (perhaps by another party).

This process of learning information and converging towards the final goal may be made up of smaller protocols that establish contingent, intermediate goals. For example, a protocol to secretly exchange some information may begin with a key agreement protocol to produce a session key. This process of sub-protocol invocation may be *implicit*, i.e., the main protocol embeds a sub-protocol pattern, rather than explicitly calling another protocol.

This structure is revealed when we consider what information (data and explicit evidence) is *necessary* at each step of a protocol run. For example, a key agreement protocol may use a nonce N_k to prevent replay attacks; when this protocol is used as a sub-protocol, after the session key is provided,

[1]Nonces are unique random values intended to be used only once; they are used, e.g., to protect against replay attacks.

N_k is no longer necessary, i.e., it never appears in message transmissions, receptions, or formula annotations. Data *and* properties can become unnecessary. In the case of the nonce, properties about its freshness may be unnecessary as well.

Thus far we have only discussed why the necessary information at a step of a protocol is valuable; we do not know if it can be known. It seems plausible that we should be able to know what information is *sufficient* for a protocol to continue, because this is essentially a backup of the protocol run. However, the *necessary* information is effectively a *minimal backup*: the smallest backup that allows the protocol to continue.

Our solution is to first define and compute backups of cryptographic protocol runs, then we will prove that they are minimal by a particular ordering. This solution strategy has the added benefit of giving us an "optimal" fault-tolerance strategy in the form of minimal backups.

Since our backups are minimal, and therefore capture the necessary information at a protocol step, they provide a rich diagnostic to protocol researchers. We show how studying the minimal backup can detect certain kinds of flaws in cryptographic protocols. We discuss how the structure of a protocol is revealed by how the minimal backup changes over time. We evaluate a test-suite of protocols [15] using this diagnostic method and discuss our findings. We find (Sec. 7) that almost all protocol roles discard over 60% of their available information at some point, thereby demonstrating their structure.

Our work is presented in the context of an adaptation of CPPL, the Cryptographic Protocol Programming Language [6]. We have built an actual tool and applied it to concrete representations of protocols, as we discuss in Section 7. All our work is formalized using the Coq proof assistant [23], and we make our formalization freely available[2]. Coq provides numerous advantages over paper-and-pencil formalizations. First, we use Coq to mechanically check our proofs, thereby bestowing much greater confidence on our formalization and on the correctness of our theorems. Second, because all proofs in Coq are constructive, our tool is actually a certified implementation that is extracted automatically in a standard way from our formalization, thereby giving us confidence in the tool also. Finally, being a mechanized representation means others can much more easily adapt this work to related projects and obtain high confidence in the results.

2. INFORMAL SOLUTION

In this section we provide the high-level overview of our solution to the minimal backup problem. Afterwards, we develop the formal framework and then present our findings.

Protocols.

Figure 1 presents the Andrew Secure RPC [19] protocol. (See Sec. 3 for details of the message syntax.) The essence of a cryptographic protocol is a list of messages to be sent or received on a network. A reasonable adversary has the ability to inject or redirect messages, so we cannot rely on where the informal descriptions indicate a message should be sent to or received from.

These messages are decorated with formulas that express properties about values in the messages. In the Andrew protocol, these formulas would express the freshness of the nonces `na`, `nb`, and `nbn` and the key `kabn`. The evidence of these formulas would be the generation time of the value. Executions may compare this time with the current time at their use to ensure the assumptions about the capacity of the adversary are valid. (For example, an execution may assume that a nonce can be guessed after T seconds and ensure that this interval has not passed.)

```
a knows a:name b:name kab:symkey
  learns kabn:symkey
b knows a:name b:name kab:symkey
  learns kabn:symkey

1 a -> b : a, {|na:nonce|} kab
2 b -> a : {|na:nonce, nb:nonce|} kab
3 a -> b : {|nb:nonce|} kab]
4 b -> a : {|kabn:symkey, nbn:nonce|} kab
```

Figure 1: Andrew Secure RPC Protocol

Backups.

The intuitive picture of a protocol as a list of messages gives rise to a natural understanding of a protocol backup: a backup after a step of the protocol is (a) the data necessary to produce or understand the rest of the messages and (b) the evidence of formulas relevant to this data. We formalize this intuition in Section 4.

Consider the Andrew protocol from the perspective of the B role. Before the protocol starts, i.e., before step 1, the only necessary data is the initial knowledge of the role. However, between steps 1 and 2, it is necessary to remember `na`, so it can be sent back to role A. Similarly, between steps 2 and 3, it is necessary to remember `nb`, so that A's response can be checked, and that it was uniquely generated locally, so that its freshness can be checked as well.

Minimal Backups.

We must define an ordering on backups before we are able to meaningfully describe a backup that is minimal. Intuitively, a backup a is smaller than a backup b, when the messages are physically smaller and the formulas are stronger, i.e., any formula of b can be deduced from the formulas in a. (A formula could express a property such as "A is even", "A is prime", or "A is 2". The latter property is stronger than the first two, since they can both be derived from it.) For details, see Section 4.1.

Computing the Minimal Backup.

When we compute the minimal backup, we must ensure that (a) minimality is achieved and (b) that the content is available at the correct point in the protocol run. This part of work is highly dependent on our formalization of protocols, so we must simply refer the reader to Section 5.1.

After computing the content of the backup, we must calculate the relevant formulas, i.e., the formulas about the necessary data. The interesting part about this is that the formulas and backup content are defined inductively. We ensure that this set is finite and computable in Section 5.2.

[2]Sources are available at:
`http://www.cs.brown.edu/research/plt/dl/fmse2008/`.

$$
\begin{array}{rcl}
s & \rightarrow & . \quad | \quad sd \Rightarrow s \\
sd & \rightarrow & +m, \Phi \quad | \quad -m, \Psi \\
m & \rightarrow & \text{nil} \quad | \quad v \quad | \quad k \\
 & | & (m, m') \quad | \quad hash(m) \quad | \quad < v = m > \\
 & | & [m]v \quad | \quad [|m|]v \quad | \quad \{m\}v \quad | \quad \{|m|\}v \\
v, u & \rightarrow & x : t \\
t & \rightarrow & \text{text} \quad | \quad \text{msg} \quad | \quad \text{nonce} \\
 & | & \text{name} \quad | \quad \text{symkey} \quad | \quad \text{pubkey} \quad | \quad \text{channel}
\end{array}
$$

Figure 2: Strand Spaces Syntax

Outline.

We formalize our notion of cryptographic protocols in Sec. 3. We define the notion of protocol backups and provide an ordering on backups in Sec. 4. We show how to provably compute the minimal backup for a protocol at any given point in its execution in Sec. 5 using the Andrew protocol role as an example. Finally, we discuss our findings, related work, and our conclusions.

3. INTRODUCTION TO STRAND SPACES

The essence of a cryptographic protocol is a list of messages that are sent and received. These messages are decorated with the trust management formulas that they rely on or guarantee. This intuition precisely matches the extant *strand space* [21] model of cryptographic protocols extended with trust managment formulas [8]. Naturally, people will not want to specify a protocol at this level of abstraction. This model is, however, the semantic interpretation of the practical cryptographic protocol programming language, CPPL, discussed at the end of this section. There is, therefore, a convenient notation that corresponds to this semantics.

3.1 Syntax

Our syntax of strands is presented in Figure 2.[3] Figure 3 presents the strand encoding of role `B` of the Andrew Secure RPC [19] protocol (Fig 1). This strand only encodes half of the protocol; another strand would be necessary to fully encode the protocol.

A strand (s) is a list of strand descriptions (sd), where "." represents the empty strand and $sd \Rightarrow s$ represents combining one description (sd) with the rest of the strand (s). A strand description is either: $+m, \Phi$, which represents guaranteeing the formulas Φ and sending m; or, $-m, \Psi$, which represents receiving the message m and relying on the formulas Ψ.

Formulas in relies and guarantees may contain strand identifiers, in addition to logical variables and strand values. If bound in the environment at runtime, a strand identifier will be replaced in Φ (Ψ) by the value to which it is bound; if not yet bound, it serves as a query variable that will be bound as a consequence of a trust management call. Logical variables in a trust management formula, if they occur, are interpreted implicitly universally.

We use m for message patterns. They may be constructed by concatenation (,), hashing ($hash(m)$), variable binding and pattern matching ($< v = m >$), asymmetric signing ($[m]v$), symmetric signing ($[|m|]v$), asymmetric encryption ($\{m\}v$), and symmetric encryption ($\{|m|\}v$). In these last four cases, v is said to be in the *key-position*. For example, in the Andrew role `kab` is in key-position on line 2. Concatenation is right associative. Parentheses ("(" and ")") are informally used to control precedence.

We write $vars(\dots)$ to refer to the identifiers that appear in a syntactic object. We write $bound(\dots)$ for those identifiers that are bound by the object.

[3]The syntax and semantics of the formulas is a parameter of the language.

3.2 Semantic Interpretation

A strand merely specifies what messages are sent and received. In this model, a protocol designer regards the cryptographic primitives as black boxes, and concentrates on the structural aspects of the protocol. The representation does not specify to whom messages are sent or from whence they are received. This corresponds to the Dolev-Yao model [3] that allows an adversary to have maximal power to manipulate the protocol by modifying, redirecting, and generating messages *ex nihilo*. This ensures that proofs built on the semantics are secure in the face of a powerful adversary.

The basic abilities of adversary behavior that make up the Dolev-Yao model, including: transmitting a known value such as a name, a key, or whole message; transmitting an encrypted message after learning its plain text and key; and transmitting a plain text after learning a ciphertext and its decryption key. The adversary can also manipulate the plain-text structure of messages: concatenating and separating message components, adding and removing constants, etc. Since an adversary that encrypts or decrypts must learn the key from the network, any key used by the adversary—compromised keys—have always been transmitted by some participant.

A useful concept when discussing the adversary is a *uniquely originating value*. This is a value that only originates (enters the network) at one unique location. Nonces and other randomly generated data are perfect examples of unique origination. By definition, the adversary cannot know these values until they have sent in an unprotected context.

A strand is *local* in the sense that it describes what one principal P does. This involves deciding what values to bind to identifiers; what messages to send; how to process a message that is received; and how to select a procedure to call as a sub-protocol. A strand says nothing about how messages are routed on a network; nothing about what another principal P' does with messages received from P; nothing about how another principal P'' created the messages that P receives; etc. Likewise, it describes only the execution of one protocol role. In essence, the execution semantics describes only a single principal executing a single run of a single role.

To reason about a protocol's execution as a whole, we need to combine strands. We do this using the *global* bundle semantics as provided by earlier work [6]. But, the details of this are not important for our purposes, because our work naturally focuses on the local semantics, i.e., the execution at a single server, rather than the global semantics. However, it is incredibly important when actually using strand spaces to verify protocols to include this step in the analysis.

3.3 The Runtime Environment

To explain how strands execute, we must first introduce the notion of a runtime environment. In our view of protocol behavior, as a principal executes a single local run of a pro-

```
0  − ("call", pr:name, "b", ai:nonce, a:name, b:name, kab:symkey), _ ⇒
1  + ("accept"), accept([chana:channel]) ⇒
2  − ("msg", chana, a, {| na:nonce |} kab), _ ⇒
3  + ("new_nonce"), new_nonce([nb:nonce]) ⇒
4  + ("msg", chana, {| na, nb |} kab), _ ⇒
5  − ("msg", chana, {| nb |} kab), _ ⇒
6  + ("new_nonce"), new_nonce([nbn:nonce]) ⇒
7  + ("new_symkey"), new_symkey([kabn:symkey]) ⇒
8  + ("msg", chana, {| kabn, nbn |} kab), _ ⇒
9  + ("ret", ai, kabn), _ ⇒ .
```

Figure 3: Andrew Secure RPC Role B Strand

tocol, it builds up an *environment* that binds identifiers to values encountered. As in logic programming, these bindings are commitments, never to be updated; once a value has been bound to an identifier, future occurrences of that identifier must match the value or else execution of the run aborts. For example, on line 2 of the Andrew role, the value for `a` must be the same as was received on line 0; any other value will prevent execution of the run from continuing.

We also have an auxiliary notion of *guaranteeing* formulas Φ in a runtime environment. This means asking the runtime trust management system to attempt to prove the formulas Φ. This occurs on line 1 of the Andrew role (Fig. 3), where the formula `accept([chana:channel])` is guaranteed. Identifiers in Φ already bound in the runtime environment are instantiated to the associated values. Identifiers not yet bound in the runtime environment are instantiated by the trust management system, if possible, to values that make the formulas Φ true. Thus, on line 1 of the Andrew role, a new channel is instantiated to accept a connection. The runtime environment extended with these new bindings is the result of successfully guaranteeing Φ. Thus, on line 1 of the Andrew role, the identifier `chana` is bound. If the runtime trust management system fails to establish an instance of Φ the guarantee fails. This notion of guaranteeing is employed whenever a message is sent.

Finally, there is the notion of *relying* on formulas Ψ in a runtime environment. This means adding assumptions (or "facts" in Datalog terminology) to the runtime trust management system. As with guarantees, identifiers in Ψ must have the same values as those in the runtime environment. These facts are available for future instances of guaranteeing. The Andrew role (Fig. 3) does not contain any relies. However, if on line 2 there was a formula, it would be relied upon. In general, relying is employed whenever a message is received.

These trust management notions coalesce into the crucial definition of protocol *soundness*: a protocol is sound if whenever a principal relies on a formula, another principal has previously guaranteed it. This dimension of protocol analysis and verification—commitment and trust—greatly enhances the utility and expressiveness of strands.

3.4 Well-formedness

Compilers use BNF context-free grammars to syntactically parse programs. They must also use context-sensitive rules

$$\textsc{Nil}\ \frac{}{\sigma \vdash_s nil} \qquad \textsc{Var}\ \frac{v \in \sigma}{\sigma \vdash_s v} \qquad \textsc{Const}\ \frac{}{\sigma \vdash_s k} \qquad \textsc{Join}\ \frac{\sigma \vdash_s m \qquad \sigma \vdash_s m'}{\sigma \vdash_s (m, m')}$$

$$\textsc{Hash}\ \frac{\sigma \vdash_s m}{\sigma \vdash_s hash(m)} \qquad \textsc{Bind}\ \frac{\sigma \vdash_s m}{\sigma \vdash_s < (i : msg) = m >}$$

$$\textsc{SymEnc}\ \frac{i \in \sigma \qquad \sigma \vdash_s m}{\sigma \vdash_s \{|m|\}(i : symkey)} \qquad \textsc{SymSign}\ \frac{i \in \sigma \qquad \sigma \vdash_s m}{\sigma \vdash_s [|m|](i : symkey)}$$

$$\textsc{PubEnc}\ \frac{i \in \sigma \qquad \sigma \vdash_s m}{\sigma \vdash_s \{m\}(i : pubkey)} \qquad \textsc{PubSign}\ \frac{i \in \sigma \qquad \sigma \vdash_s m}{\sigma \vdash_s [m](i : pubkey)}$$

Figure 4: Message well-formedness (sending)

to determine which syntactic expressions are legal programs: e.g., when the syntax refers only to bound identifiers. Similarly, not all strands describe realizable protocols. For example, $(+x, \_ \Rightarrow .)$ cannot be realized, because it does not account for where the value for x comes from. Like a programming language, every identifier in a strand must be bound for it to be well-formed.

The only surprising aspect of the well-formedness condition on strands is that, due to cryptographic primitives, the conditions are different on message patterns used for sending and receiving.

Intuitively, to send a message we must be able to construct it, and to construct it, every identifier must be bound. Therefore, a pattern m is well-formed for sending in an environment σ (written $\sigma \vdash_s m$ herein; see Figure 4) if all identifiers that appear in it are bound. For example, the message on line 4 of the Andrew role is well-formed, but if we removed line 3 it would not be, because `nb` would not be bound.

A similar intuition holds for using a message pattern to receive messages. To check whether a message matches a pattern, the identifiers that confirm its shape—namely, those that are used as keys or under a **hash**—must be known to the principal. Thus, we define that a pattern m is well-

$$\textsc{Nil}\ \frac{}{\sigma \vdash_r nil} \qquad \textsc{Var}\ \frac{}{\sigma \vdash_r v} \qquad \textsc{Const}\ \frac{}{\sigma \vdash_r k} \qquad \textsc{Join}\ \frac{\sigma \vdash_r m \qquad \sigma \vdash_r m'}{\sigma \vdash_r (m, m')}$$

$$\textsc{Hash}\ \frac{\sigma \vdash_s m}{\sigma \vdash_r hash(m)} \qquad \textsc{Bind}\ \frac{\sigma \vdash_r m}{\sigma \vdash_r < (i : msg) = m >}$$

$$\textsc{SymEnc}\ \frac{i \in \sigma \qquad \sigma \vdash_r m}{\sigma \vdash_r \{|m|\}(i : symkey)} \qquad \textsc{SymSign}\ \frac{i \in \sigma \qquad \sigma \vdash_r m}{\sigma \vdash_r [|m|](i : symkey)}$$

$$\textsc{PubEnc}\ \frac{i \in \sigma \qquad \sigma \vdash_r m}{\sigma \vdash_r \{m\}(i : pubkey)} \qquad \textsc{PubSign}\ \frac{i \in \sigma \qquad \sigma \vdash_r m}{\sigma \vdash_r [m](i : pubkey)}$$

Figure 5: Message well-formedness (receiving)

formed for receiving in an environment σ (written $\sigma \vdash_r m$; see Figure 5) if all identifiers that appear in key-positions or **hash**es are bound. For example, the message pattern on line 2 of the Andrew role is well-formed because `kab` is bound, but if it were not, then the pattern would not be well-formed.

We write $\sigma \vdash fs$ to mean that the formulas fs are well-formed in the environment σ. This holds exactly when the identifiers mentioned in fs are a subset of σ.

We write $\sigma \vdash sd$ to mean that the strand description sd is well-formed in the environment σ.

> $\sigma \vdash +m, \Phi$ holds if and only if $\sigma \vdash \Phi$ and $\sigma \cup bound(\Phi) \vdash_s m$; this corresponds to the intuition that Φ is guaranteed and *then* the message m is sent.
>
> $\sigma \vdash -m, \Psi$ holds if and only if $\sigma \vdash_r m$ and $\sigma \cup bound(m) \vdash \Psi$; this corresponds to the intuition that m is received and *then* Ψ is relied upon.

We write $\sigma \vdash s$ to mean that the strand s is well-formed in the environment σ.

> $\sigma \vdash .$ holds for all σ.
>
> $\sigma \vdash sd \Rightarrow s$ holds if and only if $\sigma \vdash sd$ and $\sigma \cup bound(sd) \vdash s$, corresponding to the intuition that strands describe an order of message reception and transmission.

3.5 Pragmatics

Naturally, no one would actually write down the strand semantics of a protocol initially. Therefore, we provide a domain-specific language, an extension of CPPL [6], to specify protocols in a more natural style. The semantic interpretation of this language is a set of strands, representing every possible path through a protocol. The language extends the original work with assertions, message let-values, empty messages, and failure continuations. Additionally, we provide a definition of well-formedness that ensures that every strand interpretation of a well-formed CPPL program is a well-formed strand. Figure 6 shows the CPPL encoding of the Andrew B role from above.

```
 1 proc b (a:name b:name kab:symkey) _
 2  let chana = accept in
 3  recv chana (a, {| na:nonce |} kab) -> _ then
 4   let nb = new nonce in
 5   send _ -> chana {| na, nb |} kab then
 6   recv chana {| nb |} kab -> _ then
 7    let nbn = new nonce in
 8    let kabn = new symkey in
 9    send _ -> chana {| kabn, nbn |} kab then
10    return _ (kabn)
11    else fail
12   else fail else fail
13  else fail
14 end
```

Figure 6: Andrew Secure RPC Role `B` in cppl

4. DEFINITION OF A BACKUP

Intuitively, the backup at point p is a message that can be generated at point p and if received later can be used to completely reconstruct the state at p sufficiently to execute the rest of the protocol run. For example, at step 0 of our example protocol (Fig. 3), the backup only contains the initial knowledge of the roles, because the protocol has not done any work. Before step 9, when `ai` and `kabn` are to be sent, it is clearly necessary to know them. The backup should also contain the evidence of any formulas related to `ai` or `kabn`. In this case, the only formula is `new_symkey(kabn:symkey)` from step 7.

This satisfies the second part of our intuition: if we received `ai`, `kabn`, and evidence of `new_symkey(kabn:symkey)`, then we could complete this strand from step 9. Does this backup meet the first part of the intuition? Can it be generated at step 9? It certainly can: `ai` is bound at step 0, `kabn` is bound at step 7, and `new_symkey(kabn:symkey)` is guaranteed (evidence is provided) at step 7.

With this intuition, we will now formally define a backup.

Definition A pair of a message m and the evidence of the properties associated with the formulas fs is the backup of the well-formed strand `sl` $\Rightarrow$ `sr` after `sl` if and only if:

1. The message m can be generated and the evidence of the formulas in fs can be provided after `sl`: $\emptyset \vdash$ `sl` $\Rightarrow +m, fs \Rightarrow .$
2. `sr` can be executed after receiving message m and the evidence of the formulas fs: $\emptyset \vdash -m, fs \Rightarrow$ `sr`
3. fs entails all *relevant* formulas: If x is an identifier in `sr`, m, or fs and f is a formula in `sl` that mentions x, then f must be *entailed* by fs.

The first condition corresponds to the intuition that a backup must be generated at the given point in the protocol. The second condition formalizes the intuition that the rest of the protocol must be executable by consulting the backup. The third and final condition corresponds to the intuition that all of the *relevant* properties must be deducible from the backup, where relevance is defined as mentioning any identifier in the backup (m or fs) or the rest of the protocol (`sr`).

Preservation.

It may not be immediately clear how or why this definition of a backup preserves the semantic meaning of protocol. The backup assumes that the strand `sl` has executed (per condition one.) The backup message allows the strand `sr` to be executed (per condition two.) Since the protocol run is entirely described by the run of $\texttt{sl} \Rightarrow \texttt{sr}$, we must only explain why this is equivalent to $\texttt{sl} \Rightarrow +m, fs \Rightarrow -m, fs \Rightarrow \texttt{sr}$. Clearly the beginning and the end are equivalent. The actual recording and reading of the backup, however, is not so clear. This is where we must assume that the backup is completely and entirely secure and invisible to the outside world: if it were not, then these actions would be potentially dangerous, and therefore, not equivalent. Since we assume that they *are* safe, they can be ignored for the purposes of the semantics of the protocol. Therefore, using backups preserves meaning precisely: there are no more or less attacks the adversary is capable of.

It is conceivable to not make this assumption and still maintain security. For example, we could encrypt every backup message with a key known only to the principal before recording it. We do not need to be specific about what mechanism is used for this purpose.

4.1 Minimality

Since our goal is to compute the *minimal* backup of a protocol at a certain point, it is necessary to define an ordering on backups. We have defined a backup as a pair of a message m and a set of formulas fs, so we must provide an ordering on these, then combine those orderings to construct one ordering on backups.

Intuitively, a message m is smaller than a message m' when it is physically smaller, i.e., uses less bits. However, backup messages are just sequences of values, not general messages, i.e., they do not contain encryption, hashing, etc. Therefore, we abstract the backup *message* to the set of identifiers that appear in the message. We approximate the physical ordering by the subset relation. For example, `(a)` is smaller than `(a,b)`, which is smaller than `(a,b,c)`.

The meaningful partial order on sets of formulas is in terms of entailment. Φ entails Ψ, when for every $\psi \in \Psi$, $\Phi \models \psi$. The formula "A is 2" entails the two formulas "A is even" and "A is prime".

Given these definitions of ordering, is it feasible that a "smallest" object exists? In the case of the values, clearly the subset relation has this property. However, for formulas the situation is more murky. For example, two formulas may entail each other, and therefore are interchangeable in any backup, so backups are not unique. Therefore, we will not be able to compute the "minimal" backup (since this implies a uniqueness we cannot attain.) Instead, we will guarantee one member of the equivalence class of backups under this notion of formula equivalence.[4]

5. COMPUTING THE MINIMAL BACKUP

Using the ordering on backups given above, our goal is to compute the minimal backup. There are two parts of this task: (1) calculating the message content and (2) calculating the relevant formulas. Because the relevant formulas are dependent on the message content—any formulas relevant to identifiers in the message are relevant—we calculate the message content first, then show how to find the relevant formulas.

We will provide a running example by considering the backup at point 8 in the Andrew protocol (Fig. 3).

5.1 Backup Content

Our general strategy for computing the necessary data to include in the backup of $\texttt{sl} \Rightarrow \texttt{sr}$ is:

1. Consider the smallest set σ such that $\sigma \vdash \texttt{sr}$.
2. Construct a message, m, that binds σ.
3. Prove that m is well-formed for sending after the execution of `sl`, i.e., condition (1) of a backup.
4. Prove that `sr` is well-formed after receiving m, i.e., condition (2) of a backup.

Computing σ.

σ can be computed through a number of proofs and simple manipulations of sets, combined with a lot of tedious work at the bottom-most point. We will walk through the high-level process and leave out the tedium. As a notational convenience, we will write $sset(x)$ for the smallest set σ such that $\sigma \vdash x$, where x may be a strand, strand description, message, or formula. We write $sset_r(x)$ and $sset_s(x)$ for the smallest set such that the message x is well-formed for receiving and sending, respectively.

We first prove the trivial theorem that $sset(.) = \emptyset$ (recall that . is the empty strand). We then prove that for strand description sd and strand s,

$$sset(sd \Rightarrow s) = sset(sd) \cup (sset(s) - bound(sd)).$$

The rationale of this theorem is very clear: strand descriptions introduce bindings, which could not possibly be in the backup, and they require bindings, which must be in the backup.

Next, we do a similar case analysis on strand descriptions. For the receiving strand description $-m, fs$, we show that

$$sset(-m, fs) = sset_r(m) \cup (sset(fs) - bound(m)).$$

Intuitively, a message m has data that are necessary to deconstruct it, such as keys or hash contents, $(sset_r(m))$, after which it binds identifiers $(bound(m))$ that are referred to $(sset(fs))$ by the relied-upon formulas, fs. For the sending strand description $+m, fs$, we have a dual formulation where

$$sset(+m, fs) = sset(fs) \cup (sset_s(m) - bound(fs)).$$

Intuitively, a principal guarantees a number of formulas fs, then constructs a message m, which refers to some identifiers $(sset_s(m))$, that may have been bound by the formula derivations $(bound(fs))$.

At the bottom, it remains to define the smallest sets for formulas, sent messages, and received messages. Each of these is very straight-forward and tedious: The identifiers mentioned by a formula are the smallest set for a set of formulas; The identifiers mentioned by a sent message are the smallest set; and, the keys and hash contents are the smallest set for a received message.

Example. In our example, the necessary set is $\{\texttt{chana}, \texttt{kabn}, \texttt{nbn}, \texttt{kab}, \texttt{ai}\}$, because all these are referred to

[4]Astute observers will recall Church's Theorem, which implies that in reasonable logics "A iff B" is undecidable.

in the strand starting at point 8. In this case, there are no identifiers bound, so nothing is subtracted from the set at any point.

Constructing the message m.

It is trivial to construct a message that binds σ: simply include each member of σ in the message, i.e., $m = x_1, x_2, \ldots, x_n$, where x_i is the ith element of σ. We additionally prefix the message with the constant `"backup"` and an integer indicating the point in the protocol. This is a nod to the practical concern that it is necessary to ensure that backups at different points cannot be confused.

Example. In the example, the backup message is: `("backup", "8", chana, kabn, nbn, kab, ai)`. In this message, both `nbn` and `ai` are nonces that can be used to distinguish backups of different sessions.

Proving well-formedness of m.

We must prove that m is well-formed. If it is not, then an implementation will not be able to generate the backup. Recall that a backup of $\texttt{sl} \Rightarrow \texttt{sr}$ is only defined if the strand is well-formed, i.e., $\emptyset \vdash \texttt{sl} \Rightarrow \texttt{sr}$. We will use this to prove that the message m is well-formed after `sl`, i.e., it can be generated after `sl`. We first prove the following theorem:

THEOREM 1. *(`type_strand_cut`) For strands s and s', $\Sigma \vdash s \Rightarrow s'$ if and only if $\Sigma \vdash s$ and $\Sigma \cup bound(s) \vdash s'$.*

PROOF SUMMARY. Induction on the length of the strand s combined with set theory identities. □

Based on this general result, it is simple to prove that m is well-formed. First, we can read off that $\emptyset \vdash \texttt{sl} \Rightarrow \texttt{sr}$ implies $bound(\texttt{sl}) \vdash \texttt{sr}$. Since we know σ is the smallest set such that $\sigma \vdash \texttt{sr}$, we can conclude that $\sigma \subseteq bound(\texttt{sl})$, i.e., the necessary data is present. Then, we prove that $\theta \vdash m$ if and only if $vars(m) \subseteq \theta$, when m is a message that has the form $x_1, x_2, \ldots, x_n$, as our backup message does. Therefore, $bound(\texttt{sl}) \vdash m$, because $vars(m) = \sigma$ and $\sigma \subseteq bound(\texttt{sl})$.

Example. In our example, the message is well-formed because each identifier is bound somewhere before point 8: `chana` is bound at point 1; `kabn` at point 7; `nbn` at point 6; `kab` and `ai` at point 0.

Proving well-formedness of `sr`.

Our construction of m allows for a concise proof of $\emptyset \vdash -m, \_ \Rightarrow \texttt{sr}$, because we constructed m such that the identifiers bound are exactly $sset(\texttt{sr})$. This is the smallest set σ such that $\sigma \vdash \texttt{sr}$, so clearly $\emptyset \vdash -m, \_ \Rightarrow \texttt{sr}$.

Example. In our example, the rest of strand is well-formed because m binds all the identifiers necessary: $\{\texttt{chana}, \texttt{kabn}, \texttt{nbn}, \texttt{kab}, \texttt{ai}\}$.

5.2 Computing the formulas

A backup must also entail all formulas relevant to the data. A minimal backup includes that smallest such set. Therefore, we must (a) identify the relevant formulas and (b) compute the minimal set that entails them all.

Relevant formulas.

A formula is considered relevant to a value if it contains a reference to its binding. For example, "A is prime" is relevant to the value A. Some formulas mention values not otherwise contained in the backup data. For example, suppose that the backup data is only A, but a formula relevant to A is "$A + B = C$". Since this formula references B and C, they must be included in the backup set as well. Furthermore, any formulas relevant to B and C must also be included.

Thus, the relevant formulas of a backup are an inductively defined set. We must prove that this set is finite and computable. If it is not finite, then it cannot be realistically saved. If it is not computable, then it cannot be constructed, used, or studied.

THEOREM 2. *The set of relevant formulas is finite and computable for all backups of all protocols.*

PROOF SUMMARY. We rely on the finite length of strands and the finite number of formulas attached to any point in the strand. Since the set of all formulas of a strand is finite and computable, and because identifier reference is computable, we can easily construct the set of relevant formulas. □

Strongest formulas.

Once we have the set of relevant formulas, we must select the minimal, or strongest, subset, i.e., the smallest set such that all relevant formulas are entailed.

In our protocol environment, CPPL, the logic of formulas is not specified. Instead, it is left as a parameter, so that whatever logic is necessary for a particular protocol may be used. Therefore, we cannot define the minimality calculation in general, since some logics do not have a computable notion of entailment, and push this burden on the provider of the logic.

However, a very common logic is a simple predicate language combined with a database. This is the logic used by the Andrew Secure RPC example. In this logic, entailment is mapped to inclusion in the database. Therefore, no formula entails anything but itself, and the minimality calculation is trivial: each set is minimal for itself.

5.3 Putting Together the Pieces

At this point, we can give our minimum backup of the strand $\texttt{sl} \Rightarrow \texttt{sr}$: It is the message m that binds $sset(\texttt{sr})$ as well as all identifiers mentioned in Φ, where Φ is the relevant subset of the formulas that appear in `sl`.

It is necessary to prove that this backup—m and Φ—satisfy the three conditions on backups: (1) $\emptyset \vdash \texttt{sl} \Rightarrow +m, \Phi \Rightarrow .$; (2) $\emptyset \vdash -m, \Phi \Rightarrow \texttt{sr}$; and (3) If x is an identifier mentioned in `sr`, m, or Φ and f is a formula in `sl` that mentions x, then f must be deducible from Φ.

Each of these properties is verified easily by the construction of the various parts. The backup values, m, were constructed such that (1) and (2) hold, and modified in a sound way during the the relevant formula calculation. Property (3) holds by definition of our computation of the relevant formula set.

We present the minimal backup at each point of role B of the Andrew Secure RPC protocol in Table 1.

6. INSIGHTS

As mentioned earlier, the minimal backup of a protocol provides some subtle insights into its structure. This capability comes from a principled understanding of *what* the minimal backup is: an extensional description of what is

Step	Values	Formulas
0	$\{\}$	$\{\}$
1	$\{kab, ai\}$	$\{\}$
2	$\{kab, ai, chana\}$	$\{accept(chana)\}$
3	$\{kab, ai, chana, na\}$	$\{accept(chana)\}$
4	$\{kab, ai, chana, na, nb\}$	$\{accept(chana), new\_nonce(nb)\}$
5	$\{kab, ai, chana, nb\}$	$\{accept(chana), new\_nonce(nb)\}$
6	$\{kab, ai, chana\}$	$\{accept(chana)\}$
7	$\{kab, ai, chana, nbn\}$	$\{accept(chana), new\_nonce(nbn)\}$
8	$\{kab, ai, chana, nbn, kabn\}$	$\{accept(chana), new\_nonce(nbn), new\_symkey(kabn)\}$
9	$\{ai, kabn\}$	$\{new\_symkey(kabn)\}$

Table 1: Andrew Secure RPC Role B Backups

relevant at each point in the protocol. This extensionality allows us to easily compare and understand protocols.

For example, when the minimal backup at points i and j of a protocol are the same, then the protocol *has the same requirements* at each of those points. This is extremely dangerous, because it means that in principle there is nothing preventing a protocol from skipping directly to point j after point $i-1$. The example protocol, Andrew Secure RPC [19], has this property. Notice in the minimal backup table (Table 1) that the backup contents are identical at steps 2 and 6. This means that role B can skip the execution of steps 3 through 5. This directly corresponds to a known attack on the protocol [1].

We learn interesting things about the structure of a protocol even when the minimal backups at two points in the protocol are not identical. Suppose that the only difference between the minimal backups at points 2 and 5 of a protocol is a single identifier, `nb`. This means that the points in between must have been for the very purpose of acquiring `nb`. If this is not actually the case, then something must be wrong with the protocol. An alternative take on this problem is to check when some value is learned *without* the generation of a new nonce to provide an authentication challenge; this pattern reveals the kinds of errors discussed by Lowe [13]. For example, the Denning-Sacco protocol [2] contains only one nonce, rather than two as Lowe suggests. We intend on using this intuition to automatically construct attacks on protocols, but do not pursue that avenue in this work.

This is an example of a general principle: when the size of the minimal backup decreases, then some sub-task has been completed. This reveals the structure of the protocol. In the extreme case, when the size of minimal backup decreases to zero, then this means that the protocol's past is independent of its future. Another way of thinking of this case is that what is presented as *one* protocol is in fact *two* independent protocols. Naturally, a good cryptographic protocol will not have this property, but many other protocols do: for example, after a request-response sequence in HTTP, the protocol essentially starts afresh. Our analysis would show this as the minimal backup becoming empty.

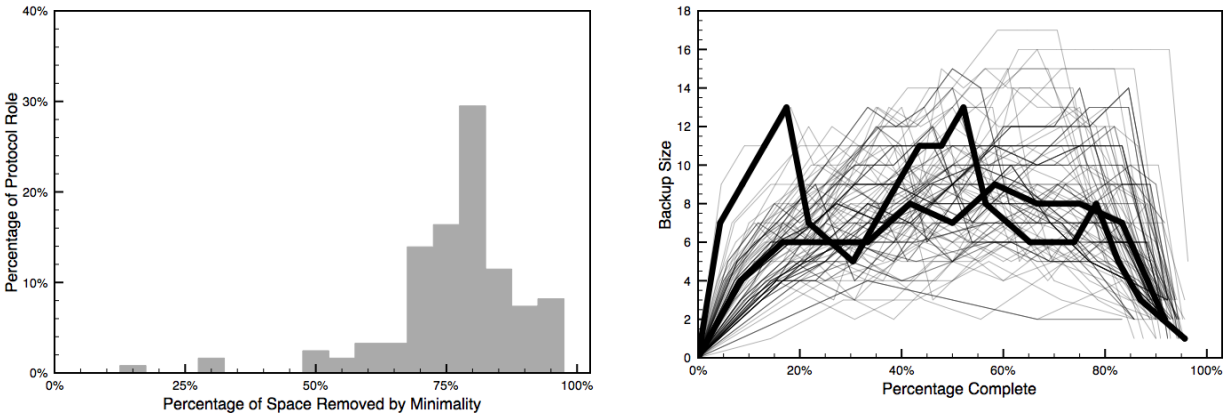
7. APPLICATIONS

Although we present minimal backups primarily as a diagnostic tool, they are useful in practical applications. The most obvious application of minimal backups is, of course, as a backup. If the minimal backup is stored on secure, non-volatile storage, then it can be used to recover a running protocol session from a failure. Job migration is a natural out-cropping of this application, as migration can be understood as causing a "failure" and "recovering" from it at another location. Of course, in a deployment where the protocol is part of some larger service, it is also necessary to handle TCP state (e.g., [14, 20]), persistent service data [12], et cetera [4], but the minimal backup of the cryptographic protocol provides the foundation of the entire fault-tolerance milieu. Both of these applications provide core components of a scalability strategy designed to handle high customer load: backups deal with inevitable failure and job migration enables load-balancing.

8. EVALUATION

We have studied the minimal backups of 121 protocol roles found in the SPORE Protocol Repository [15]. We have used our analysis to determine the minimal backups at every point in each of these protocols. Our implementation, extracted from Coq, was able to calculate these in approximately 1 second. We were interested in the applicability of the minimal backup-based analysis and the pragmatic benefits of minimality.

First, we found that no protocol role had an empty minimal backup after the first step of the protocol. Second, we found that every protocol exhibited some decrease in the size of the minimal backup at some point in the protocol. We calculated the largest percentage decrease in backup size for each protocol and determined the minimum (16%), mean (65%), median (69%), and maximum (93%). This shows that most protocols discard over half of their knowledge at some point in the run. Figure 7a shows the space savings due to minimality as a percentage of total backup space. Notice that the graph is heavily weighted to the right: this demonstrates the practical utility of minimality. Figure 7b presents a graph of the size of a minimal backup of a protocol at each stage of completion, where the average of all protocols has been emphasized. This graph indicates that backups have an "n" shape, which matches the intuition that they start from a small amount of knowledge, build towards some goal, but no longer require the intermediate knowledge by the end of the run. This evidence of our intuition justifies the diagnostic application of the minimal backup graph.



(a) (b)

Figure 7: Minimal Backup Diagnostic Graphs

9. RELATED WORK

Analysis.

Our analysis is similar to live expression analysis [10]. Liveness analysis determines whether a value is necessary in the rest of a computation, similar to how our analysis determines what information is necessary to complete a protocol run. We could use liveness analysis on the implementation of a cryptographic protocol to get some of the results of our analysis, e.g., sound backups. However, liveness analysis is typically not complete for implementation languages. Our analysis is complete, modulo our approximation of entailment[5], in the form of our proof of minimality of backups.

If we used live-expression analysis at the implementation level, then we would approximate the checkpointing approach of Li et al.'s [11]. This approach instruments a program with points where full system checkpoints can be taken. Due to the difficulty in determining what data is live in C, they must snapshot the entire system memory. They pursue minimality via a heuristic training process that determines where to place checkpoint locations. While their work is inspiring, we do not need to resort to these techniques, because we can calculate provably minimal backups. It would be interesting to compare the empirical results of each approach to ascertain how close they come to minimality.

Diagnostic.

The diagnostic aspect of minimal backups can be seen as a generalization of the idea of authentication tests [7]. These are protocol patterns that represent the attainment of a subgoal. In a sense, analyzing the minimal backup is a means of finding other similar patterns. However, minimal backup patterns do not provide the guarantees of an authentication test, in general.

The diagnostic aspect of our analysis can also be considered a way of determining the degree to which a protocol exhibits soft-state [16, 9, 12]. This state, recognized as desirable by systems designers, is what is not *necessary* for a protocol to execute properly, but may be useful in improving performance. For example, by definition a cache protocol does not need a cached version of all of memory, so it can remove pages from the cache under press. But when it has some particular page, there are performance benefits. In a fuzzy way, the difference between the entire state and the minimal backup represents the soft-state. Our literature review has not unveiled any analyses that automatically determine the degree to which a protocol uses soft-state.

[5]In our implementation, the logic's entailment relation *is* decidable. This caveat only applies when an undecidable logic is used.

Fault-tolerance.

In the realm of cryptographic protocols, Williams [24] discusses how to produce a fault-tolerant version of a trusted third-party authentication service, with applications to Kerberos and the Needham-Schroeder protocol. At the time of this writing, we were not able to obtain a copy of this paper, despite email requests. Based on the abstract, we assume that (a) their work is tied to details about the protocols in question, rather than a general technique; and, (b) that they do not provide verified guarantees about backup minimality.

Gong [5] developed a distribution authentication protocol designed to increase fault-tolerance, while providing reasonable security guarantees. Inspired by this approach, Reiter [17, 18] developed a methodology of constructing replications of general services and applied this work to an authentication protocol.

These two strategies work by requiring multiple servers to partially serve, or authenticate, a client that has many such servers available. The strategy increases fault-tolerance, because only a fraction of those servers must be available at any given time. Our work contrasts with this approach by providing an analysis that allows fail-over of *any* protocol without modification through a uniform strategy of storing the minimal backup securely.

10. CONCLUSION

We have formally defined a protocol backup for a particular point in a cryptographic protocol execution. We have shown how to provably compute the minimal backup, given a reasonable ordering on backups. We have described the utility of the minimal backup as a diagnostic to protocol designers. We have also explained how the minimal backup can be used to solve the problems of fault-tolerance and job-migration of cryptographic protocol sessions. Finally,

we have provided a summary of our study of the minimal backups of 121 protocol roles from the SPORE repository.

This work demonstrates that the internal structure of a protocol is revealed by investigating its minimal backup. This is made possible by the formal proof that the backups are truly minimal and that their construction is sound with respect to the protocol semantics. This analysis is also useful for enabling a scalability property: fault-tolerance.

We are interested in investigating how to automatically analyze minimal backups to identify common, or important, protocol structures. For example, we might be able to automatically identify authentication tests or replay attacks. We are also interested in pursuing an optimization of cryptographic protocols that decreases the size of the minimal backup, while preserving the security guarantees of each protocol role.

Acknowledgments.

This work is partially supported by the NSF (CCF-0447509, CNS-0627310, and a Graduate Research Fellowship), Cisco, and Google. We are grateful for the advice and encouragement of Joshua Guttman and John Ramsdell.

11. REFERENCES

[1] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *Proceedings of the Royal Society*, Series A, 426(1871):233–271, December 1989.

[2] D. Denning and G. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8), Aug. 1981.

[3] D. Dolev and A. Yao. On the security of public-key protocols. *IEEE Transactions on Information Theory*, 29:198–208, 1983.

[4] F. Douglis. Transparent process migration: Design alternatives and the Sprite implementation. *Software–Practice and Experience*, 21(8):757–785, Aug. 1991.

[5] L. Gong. Increasing availability and security of an authentication service. *IEEE Journal on Selected Areas in Communications*, 11(5):657–662, 1993.

[6] J. D. Guttman, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Programming cryptographic protocols. In *Trust in Global Computing*, 2005.

[7] J. D. Guttman and F. J. Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, June 2002.

[8] J. D. Guttman, F. J. Thayer, J. A. Carlson, J. C. Herzog, J. D. Ramsdell, and B. T. Sniffen. Trust management in strand spaces: A rely-guarantee method. In *European Symposium on Programming*, 2004.

[9] P. Ji, Z. Ge, J. Kurose, and D. Towsley. A comparison of hard-state and soft-state signaling protocols. In *SIGCOMM Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2003.

[10] G. A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1973.

[11] C.-C. J. Li, E. M. Stewart, and W. K. Fuchs. Compiler-assisted full checkpointing. *Software–Practice and Experience*, 24(10):871–886, 1994.

[12] B. C. Ling, E. Kiciman, and A. Fox. Session state: Beyond soft state. In *Networked Systems Design and Implementation*, 2004.

[13] G. Lowe. A family of attacks upon authentication protocols. Department of Mathematics and Computer Science 5, University of Leicester, 1997.

[14] M. Marwah, S. Mishra, and C. Fetzer. Tcp server fault tolerance using connection migration to a backup server. In *Dependable Systems and Networks*, 2003.

[15] Project EVA. Security protocols open repository. http://www.lsv.ens-cachan.fr/spore/, 2007.

[16] S. Raman and S. McCanne. A model, analysis, and protocol framework for soft state-based communication. In *SIGCOMM Applications, Technologies, Architectures, and Protocols for Computer Communications*, 1999.

[17] M. K. Reiter and K. P. Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems*, 16(3):986–1009, 1994.

[18] M. K. Reiter, K. P. Birman, and R. van Renesse. A security architecture for fault-tolerant systems. *ACM Transactions on Computer Systems*, 12(4):340–371, 1994.

[19] M. Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems*, 7(3):247–280, 1989.

[20] F. Sultan, K. Srinivasan, D. Iyer, and L. Iftode. Migratory tcp: connection migration for service continuity in the internet. *Distributed Computing Systems*, 2002.

[21] F. J. Thayer, J. C. Herzog, and J. D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3):191–230, 1999.

[22] F. J. THAYER Fábrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *1998 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, May 1998.

[23] The Coq development team. *The Coq proof assistant reference manual*, 8.1 edition, 2007.

[24] D. Williams and H. Lutfiyya. Fault-tolerant authentication services. *International Journal of Computers and Applications*, 2007.