CS 276 Project: Survey of Symbolic Techniques for Protocol Analysis

Adam Chlipala

May 19, 2004

1 Introduction

Traditional models of cryptography have been based on complexity theory and probability. Though an analysis of a system may proceed in layers, a proof of a system's security can in principle be reduced to one that assumes only the quality of some basic primitives, like RSA. However, often these details can obscure the essence of a problem. In such cases, symbolic models of cryptographic systems can make analysis much more tractable.

For instance, the design of cryptographic protocols is notoriously difficult. Subtle changes in a protocol can have drastic effects on its security. Precise complexity theoretic bounds on security are important, but often it is issues of trust between principals that lead to the most design errors. [3] introduced the BAN logic for reasoning about authentication protocols in a way that abstracts the details of cryptographic primitives. By breaking a protocol down into the assumptions of its principals at each stage, it is possible to use the BAN logic's formal proof system to show that a correct execution of the protocol implies the designer's conclusions, without relying on unwanted assumptions. Inability to prove this may reveal unexpected and undesired assumptions.

Another formal system with a similar purpose is the spi calculus [2]. It can be used to model a protocol as a system of interacting processes, standing for the principals. In contrast to the informal models of protocols common in the literature, spi calculus programs have a precise semantics and may even be executed. This facilitates formal analysis of protocols. Proofs based on formal notions of equivalences of processes can be used to prove properties like secrecy and authenticity of protocols. [1] makes automatizing the proof process even easier by introducing a type system that implies secrecy for well-typed programs.

I will introduce these formal models and show how they may be used to analyze specific protocols.

2 The BAN logic of authentication

The goal of an authentication protocol is to establish communication between principals in a way that satisfies each of the other's identity. It will generally be impossible to achieve this without some assumptions about knowledge, like "Bob is the only one who knows his private key;" and about trust, like "this server can be trusted to execute faithfully my request to forward data to a particular person."

The BAN logic models these beliefs explicitly, using a modal logic with one mode for each principal. For instance, $A \models \overset{K}{\mapsto} B$ will indicate that principal A believes that K is a valid public key to use for communication with B. An authentication protocol based on a key exchange server and public key cryptography may have a belief like this as its final goal. For a protocol based on symmetric key cryptography, a goal might be $A \models A \overset{K}{\leftrightarrow} B$, indicating that A believes K is a valid key for secure communication between A and B. The BAN logic contains a proof system that can be used to determine if every belief a principal declares follows logically from the results of the protocol.

2.1 Formulas

For simplicity, the BAN logic makes everything a formula. Messages sent between principals are changed from arbitrary data into formulas representing the beliefs of the sender that are implicit in the fact that he is sending the data. This somewhat counterintuitive idea should be made clearer in the example to follow shortly.

The basic logic can be extended to contain judgments dealing with all sorts of beliefs that are important for protocols that use particular cryptographic techniques. The following are the kinds of beliefs in the simple version of BAN presented in [3]. Since messages are translated into beliefs, this same informal grammar also describes the allowed kinds of messages.

$P \models X$:	P believes X
$P \triangleleft X$:	At some point, P has received X from another principal.
$P \prec X$:	At some point, P sent X.
$P \Rightarrow X$:	P is an authority on the truth of X.
$\sharp(X)$:	No message containing X was sent before the protocol began.
$P \overset{K}{\leftrightarrow} Q$:	K is a good symmetric key for communication between P and Q.
$\overset{K}{\mapsto} P$:	K is a good public key for P.
$\{X\}_K$:	This stands for the encryption of plaintext X using key K.

2.2 Proof rules

BAN's model of protocol execution is simplified. A binary notion of time is used, where a message is sent or a principal believes something either any time before the current protocol run or during it. The intuitive reason for this abstraction is that temporal reasoning is required mostly to deal with replay attacks. For that purpose, we only care whether, for example, another principal has sent a message during the current interaction or whether an old message has been resent, possibly indicating an attack.

Also, BAN assumes that cryptographic primitives are secure, without stating formally what this means. The proof rules will operate under the assumption that the only way to encrypt or decrypt a message is to know the appropriate key, with no consideration of computational resources or probability. Primitives secure in the usual complexity theoretic senses will have this property in practice.

BAN defines a series of proof rules used for deducing the truth of new formulas from old ones. A proof for a protocol works as follows: We have a set of assumptions true at the beginning of the protocol run. For each message M sent to a principal P, we add the formula $P \triangleleft M$. (Note that this judgment does not indicate the sender of the message, so the proof system is accurate in reflecting that attackers may have arbitrary control over message flow.) Now we must derive any desired conclusions starting from these base formulas, using the BAN proof rules.

Here is a selection of those rules, along with their "plain English" meanings:

$$\frac{P \models Q \overset{K}{\leftrightarrow} P \quad P \triangleleft \{X\}_K}{P \models Q \prec X} \; \textit{message}$$

P believes that K is a good key for symmetric key encrypted communication with Q. In other words, P believes that if anyone but P and Q knows K, those parties may be trusted not to use it or reveal it to others. P has received a message containing the encryption of plaintext X with K. By the previous assumption, the only party capable of performing this encryption is Q. Therefore, P believes that Q sent X at some point. That point may have been before the current execution of the protocol.

$$\frac{P \models \sharp(X) \quad P \models Q \prec X}{P \models Q \models X} \; \textit{nonce}$$

P believes that message X has never been uttered before this run of the protocol. P believes that Q has sent X at some point. Therefore, P believes that Q must have done this sending during the current run.

Since we identify sending a message with believing a formula, Q believes X now.

X can be a *nonce*, a random value included in a protocol to garble the encryption of a packet, making different encrypted versions of a packet valid in different contexts and invalid in others. *nonce* is complemented by a rule that allows the conclusion that a compound message is fresh if any of its components is fresh. Thus, *nonce* can be used to provide protection against replay attacks, since it allows the conclusion that a principal still believes a formula iff it involves a component unique to the protocol run.

$$\frac{P \models Q \Rightarrow X \quad Q \models X}{P \models X}\ \mathit{jurisdiction}$$

P trusts Q as an authority on the truth of X. Q believes X. Therefore, P believes X. *jurisdiction* is particularly useful in reasoning about servers trusted to relay information between principals, generate good keys, or perform other essential tasks.

2.3 Example: Wide-mouthed-frog

The simplest example in [3] is that of the wide-mouthed-frog protocol, an authentication protocol resting on a bad assumption that may be exploited. The protocol works with three principals. A and B wish to agree on a secret key K_{ab} to use for private symmetrically encrypted communication. The problem is that a principal doesn't want to need to obtain key data for each potential communication partner ahead of time. S is a trusted authentication server that helps others obtain such keys as needed. Both A and B have secret keys (K_{as} and K_{bs}, respectively) to use for encrypted communication with S. This allows them to maintain only one key each, instead of each needing one key per potential partner. All participants are presumed to have synchronized clocks.

Only two messages are required to accomplish the exchange:

Message 1	$A \rightarrow S$	$A, \{T_a, B, K_{ab}\}_{K_{as}}$
Message 2	$S \rightarrow B$	$\{T_s, A, K_{ab}\}_{K_{bs}}$

In English: A decides that he wants to set up communication with B. A sends to S his name and a packet encrypted with the key he shares with S. The packet contains the current timestamp, A's desired communication partner, and a randomly generated key. S decrypts the packet to obtain K_{ab}. He then forwards it to B in an encrypted packet that also contains the current timestamp and A's name. Any principal receiving a message with an out-of-date timestamp during this protocol discards it.

To analyze wide-mouthed-frog with BAN, we need to translate its messages into assertions.

The occurrences of A and B in Message 1 only serve to direct the flow of traffic. They have no impact on the security of the protocol. Therefore, we will ignore them in the translation. According to the "contract" of the protocol, when an honest A sends his first message, he is declaring that T_a is a valid timestamp and K_{ab} is a new, randomly generated key. Timestamps are useful here only because it is easy to tell if a timestamp is fresh. Assuming the domain of timestamps is reasonably large, we can model them as values guaranteed to be fresh. With this view, A simply declares that T_a is a fresh value of some sort, and we can avoid reasoning about clock synchronization between principals.

Similarly, the presence of A in Message 2 is irrelevant to security, so it need have no counterpart in the translation. We handle S's timestamp in the same way as A's. However, since S doesn't generate K_{ab} itself, by including it in Message 2, S is *not* asserting that it is a good key. Rather, it is asserting that A, who chose the key, thinks it is good.

Thus, we have the following idealized protocol:

Message 1	$A \rightarrow S$	$\{T_a, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{as}}$
Message 2	$S \rightarrow B$	$\{T_s, A \models A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}$

We can formalize the initial assumptions discussed so far in BAN:

A and B believe they have good keys for communication with S.	$A \models A \overset{K_{as}}{\leftrightarrow} S$, $B \models B \overset{K_{bs}}{\leftrightarrow} S$
S believes it has good keys for communication with A and B.	$S \models A \overset{K_{as}}{\leftrightarrow} S$, $S \models B \overset{K_{bs}}{\leftrightarrow} S$
A believes it has generated a good key.	$A \models A \overset{K_{ab}}{\leftrightarrow} B$
B trusts S to relay A's declaration that the key is good.	$\forall K. B \models S \Rightarrow A \models A \overset{K}{\leftrightarrow} B$
If the protocol finishes, the timestamps are known to be fresh.	$S \models \sharp(T_a)$, $B \models \sharp(T_s)$

The goal of the protocol is to distribute a new key for A and B. In other words, we want A and B to agree that a key is good for later communication, or $A \models A \overset{K_{ab}}{\leftrightarrow} B$ and $B \models A \overset{K_{ab}}{\leftrightarrow} B$.

Now that we've fixed what we're willing to assume and what we want in an ideal protocol, we can try to use the proof rules to derive the conclusions from the premises. Besides the assumptions above, we get to reason using $S \triangleleft \{T_a, A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{as}}$ and $B \triangleleft \{T_s, A \models A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}$.

Since the first conclusion $A \models A \overset{K_{ab}}{\leftrightarrow} B$ is an assumption, we may prove it trivially.

As for the second conclusion $B \models A \overset{K_{ab}}{\leftrightarrow} B$, we begin our attempt by using the *message* rule: Since B believes K_{bs} is a good key to use with S, and B receives a message encrypted with that key, B can deduce that the message came from S. Formally, since $B \triangleleft \{T_s, A \models A \overset{K_{ab}}{\leftrightarrow} B\}_{K_{bs}}$ and $B \models B \overset{K_{bs}}{\leftrightarrow} S$, we may deduce $B \models S \prec (T_s, A \models A \overset{K_{ab}}{\leftrightarrow} B)$.

Since $B \models \sharp(T_s)$, $B \models \sharp(T_s, A \models A \overset{K_{ab}}{\leftrightarrow} B)$. Thus, by *nonce*, $B \models S \models (T_s, A \models A \overset{K_{ab}}{\leftrightarrow} B)$. In other words, since S at some point sent a message containing T_s, and T_s is fresh, S must still believe the contents of the message. If we hadn't included the nonces in the protocol, at this point we would get stuck in the proof effort. This would indicate a genuine hole in the protocol: Even if A learns that a key has been compromised, without the timestamps, an attacker can replay S's old "assurance" to B that the compromised key is good. B may then send a message encrypted with the compromised key, allowing the attacker to intercept and decrypt it.

An "and-elimination" rule allows the conclusions $B \models S \models T_s$ and $B \models S \models A \models A \overset{K_{ab}}{\leftrightarrow} B$ from the previous deduction. B trusts S as an authority on the truth of $A \models A \overset{K_{ab}}{\leftrightarrow} B$. Therefore, B can believe this formula as well. Formally, the *jurisdiction* rule allows us to conclude $B \models A \models A \overset{K_{ab}}{\leftrightarrow} B$.

Unfortunately, we are stuck at this point! There is no way to prove the desired conclusion. This is because wide-mouthed-frog makes one other (non-standard) assumption that is fairly apparent upon inspection: B must trust A to generate good keys. If a protocol designer hadn't realized that this assumption existed, however, attempting this proof would reveal the flaw.

This needn't be a very serious flaw, since in many situations this belief may be reasonable. For instance, the protocol may always be initiated by centrally maintained servers trusted to implement good key generation. Nonetheless, even a well-meaning A may unwittingly generate predictable keys and open up B to interceptions of messages it wants kept private.

Whether or not the assumption is a good one, by adding it we can now finish the proof with a final application of *jurisdiction*.

3 The spi calculus

One major weakness of BAN is that, to use it, one must perform a manual translation of a protocol into an abstract form. Also, even the original protocol can be considered abstract: Such protocols are generally described by their sequences of a messages, a form that is easy for humans to understand, but which has no precise semantics. Real implementations will be written in standard programming languages, relying on informal justifications that the implementations match the protocols. This leaves the author of a piece of

cryptographic software in a quandary, if he is concerned about security. He must reason from the nuts and bolts of his implementation to the very abstract assertion model of BAN, with no automatic translation available.

[2] suggests an alternative to get around this problem. They propose the spi calculus, a version of the well-known pi calculus extended with cryptographic primitives. We can represent a protocol with a system of spi calculus *processes* interacting through messages exchanged over a set of *channels*. Each process is controlled by a program written in a simple language. The language is simple enough to make reasoning tractable, but precise enough to be executed. In fact, the pi calculus model of concurrency is very close to that implemented in languages like Erlang and Concurrent ML. Therefore, it is conceivable that a beefed-up version of the spi calculus could even be used to reason about security properties of real protocol implementations directly.

3.1 The language

The spi calculus includes a few forms of *terms*, or data, indicated by meta-variables L, M, and N:

x	*Variables*, used in the usual way for binding constructs
n	*Names*, unforgeable capabilities that can be channel identifiers or cryptographic keys
(M, N)	Pairs of terms
$\overline{i}$	Natural numbers
$\{M\}_N$	The encryption of M with shared key N

Next, we need to define the programming language for processes. A *process*, with meta-variables P, Q, and R, is one of:

$\mathbf{0}$	Do nothing. (the null process)
$\overline{M}\langle N\rangle.P$	Send message N over channel M, then proceed with process P.
$M(x).P$	Run P with x bound to the contents of a message received from channel M.
$P \mid Q$	Run P and Q in parallel.
$(\nu n)P$	Generate a fresh name n and run P with n bound to that name.
$!P$	Run an infinite number of copies of P in parallel.
$[M\ is\ N]P$	If $M \neq N$, loop forever. Otherwise, run P.
$let\ (x, y) = M\ in\ P$	Execute P with x and y bound to the components of pair M.
$case\ L\ of\ \{x\}_N\ in\ P$	Execute P with x bound to the decryption of L using key N.

3.2 Informal discussion of execution semantics

The spi calculus has the usual style of interleaved concurrency semantics. When multiple processes are running concurrently and several can make execution steps, any one of them may be the first to make that step.

To avoid reasoning about buffered communication, the spi calculus uses synchronous (or rendezvous) message passing. The participants in a message transmission must "rendezvous" and hand off the message between them before either can proceed. If a process wants to send a message to a channel but no process is waiting to receive from that channel, the sender blocks until a receiver appears. The same is true for receivers blocking until the appearance of a sender.

The spi calculus uses somewhat unusual idioms to achieve control flows common in popular programming languages. This is done to avoid adding new constructs to the language. For example, a loop in which a process repeatedly receives from a channel and processes each packet is modelled by using the ! construct to spawn infinitely many processes that each perform a single read and processing.

The generation of both communication channels and keys is handled by the $(\nu n)P$ process form. We assume that this construct always generates a completely new, unguessable value. The unguessability of it is modeled by the strategic omission of programming constructs. There are only two ways for a program to gain access to a name, both determined by lexical scoping:

- The program may lexically be inside the ν construct that generated the name. In the process systems we will build to model protocols, this will generally reflect one of two situations: We will indicate

public channels and keys by wrapping the whole composition of processes with a series of ν constructs, one for each channel or key that should be known to all processes. The other common idiom is to make the body of a ν process consist of a composition of processes representing the actions of a single principal. This corresponds to having that principal generate a key at random. The rules of lexical scoping prevent the key from "leaking" into other processes, unless the next method we will discuss is used.

- A name n may be sent over a communication channel. If the name is part of a message encrypted with a key known only to one possible recipient, then only that recipient has a chance of learning n. Formally, the recipient will have access to the key, which means that it is a process within the scope of the ν that first generated the key, or that it is within the scope of a message receive construct that has received that key over some channel. The recipient uses the *case* x *of* $\{y\}_m$ *in* P construct, where m is a name for the key, x is a variable bound to a message that we hope is $\{n\}_m$, and P is the process to be run subsequently with y bound to n.

Thus, we have neatly traded complexity theoretic reasoning about the probability of key guessing for machine-checkable rules based on scoping. If key spaces are large enough, it is quite reasonable to assume that attackers will be unable to guess any individual key. Also, for cryptographic primitives secure in the usual senses, it makes sense to assume that no attacker can decrypt a message unless he has received the key through a chain of communication that was in some sense initiated by the original generator of the key.

3.3 Example: Andrew secure RPC handshake *("something new" part 1)*

[3] suggests the following simplified and improved version of the Andrew secure RPC handshake protocol. The protocol involves two principals, a client A and a server B. They know a secret key K_{ab}, and A has decided that he would like a new, different secret key to use to communicate with B. B generates a key K'_{ab} and informs A of it.

Message 1	$A \rightarrow B$	N_a
Message 2	$B \rightarrow A$	$\{N_a, K'_{ab}\}_{K_{ab}}$
Message 3	$A \rightarrow B$	$\{N_a\}_{K'_{ab}}$
Message 4	$B \rightarrow A$	$\{M\}_{K'_{ab}}$

N_a is a fresh nonce generated by A to prevent replay attacks. The final message is not really part of the protocol. It is included to provide an opportunity for secrecy to be broken if the protocol has a flaw. M can be any message.

We can model this protocol with the following system of spi calculus processes. c_{ab} and c_{ba} are public channels meant for communication from A to B and vice versa.

A's behavior can be modeled by this process:

$$P_A(K_{ab}) \equiv (\nu n_a)\overline{c_{ab}}\langle n_a\rangle .c_{ba}(x).\mathit{case}\ x\ \mathit{of}\ \{y\}_{K_{ab}}\ \mathit{in}\ \mathit{let}\ (y_n, y_k) = y\ \mathit{in}\ [n_a\ \mathit{is}\ y_n]\overline{c_{ab}}\langle\{n_a\}_{y_k}\rangle .c_{ba}(z).\mathbf{0}$$

The definition of the process P_A depends on the current key K_{ab}. Considering the code in order from left to right, first ν is used to generate the nonce n_a. Then, Message 1 is sent over channel c_{ab}. A waits to receive Message 2 over c_{ba} and bind x to the result. x is decrypted into y using K_{ab}. At this point, the process will get stuck if x is not a valid encryption. If it is valid, y is decomposed into its two components, y_n (which should be N_a) and y_k (which should be K'_{ab}). An equality check is performed on n_a and y_n, with the process getting stuck if it fails. Finally, A sends Message 3 over c_{ab} and waits to receive Message 4 over c_{ba}.

Next, we can define a process for B:

$$P_B(K_{ab}, M) \equiv c_{ab}(x_n).(\nu n_k)\overline{c_{ba}}\langle\{(x_n, n_k)\}_{K_{ab}}\rangle .c_{ab}(y).\mathit{case}\ y\ \mathit{of}\ \{y_n\}_{n_k}\ \mathit{in}\ [x_n\ \mathit{is}\ y_n]\overline{c_{ba}}\langle\{M\}_{n_k}\rangle .\mathbf{0}$$

P_B begins by receiving x_n (which should be A's nonce N_a) over channel c_{ab}. Then it generates the new private key n_k. It sends x_n and n_k over c_{ba}, encrypting with K_{ab}. It receives A's reply in y and decrypts y using n_k, getting stuck if y isn't a valid encryption. It checks that the result of decryption equals the nonce A sent earlier, getting stuck otherwise. Finally, it sends M encrypted with the new key.

The general idea when using the spi calculus is that the processes involved in the protocol can be trusted to behave as desired. The potential attacks will involve external processes listening and broadcasting on public channels. Therefore, we model the protocol participants as a single process:

$$Sys(M) \equiv (\nu z_{ab}) P_A(z_{ab}) \mid P_B(z_{ab}, M)$$

We use ν to generate a fresh private key z_{ab} known only to the component processes, and we compose the instantiations of P_A and P_B to run in parallel.

3.4 Modeling secrecy in the face of arbitrary attackers

Now that we know how to model systems of principals following a protocol, we need to account for the ways attackers can interfere. We want to allow for attackers trying as wide a range of techniques as possible. The solution used in [2] is to let attackers be arbitrary spi calculus processes. The attacker and protocol system will be run in parallel. The attacker's job is to learn something about secret data, through what he receives over public channels. He may also send to public channels to trick the honest principals into revealing information.

The attacker is trying to determine facts about the secret parameters of the protocol. In the above example, M is a secret parameter. We'll make the attacker's job as easy as possible by only asking him to tell the difference between protocol systems using different values of the secret parameters. A special channel will be reserved for allowing the attacker to signal his decision on whether the protocol execution satisfies some predicate. If the attacker sends to this channel when run with the system with one set of parameters and doesn't send to it when run with another set of parameters, then he has discovered the difference in the secrets. A protocol system protects secrecy only if no attacker can succeed against it in this way.

3.5 Verifying secrecy through type checking

[1] presents a simple type system for the spi calculus that guarantees secrecy for well-typed systems. The main idea is to give each variable and name a type that indicates its level of security, *Secret*, *Public*, or *Any*. The secret parameters for a system will be treated as variables of type *Any*. This means that nothing is known about their secrecy, so the system cannot be permitted to leak information about them.

A few simple changes to the spi calculus are made to support this method. Pair terms are generalized to allow terms for tuples of arbitrary size. We will require that any encryption is performed with a key of level *Secret* or *Public*. (This makes intuitive sense, since using arbitrary inputs as keys is of dubious value.) When a message is encrypted with a *Secret* key, it must be a triple of a *Secret* value, an *Any* value, and a *Public* value. [1] recommends this because it makes clear what the secrecy levels of message components are, while standard protocols generally use implicit context to determine which received data must be kept secret.

Also, since the typed spi calculus will treat encryption functions as deterministic, it will require the presence of a fresh *confounder* in every message encrypted with a *Secret* key. This requirement serves only to avoid the standard problems with deterministic cryptography. The type system will enforce that every such encryption uses a new confounder in the last position of a 4-tuple plaintext.

The formalization begins with typing environments E of the kind familiar from standard programming language semantics. An environment is a mapping from variables to types and from names to types and terms. The term associated with a name expresses the one encryption for which the name may be used as a confounder, and this expression may refer to the variables of the environment. A name not meant to be used as a confounder can be associated with any term.

Now we can use typing rules to express the secrecy levels of terms and processes. For example:

$$\frac{E \vdash M_1 : Secret \quad E \vdash M_2 : Any \quad E \vdash M_3 : Public \quad E \vdash N : Secret \quad n : T :: \{M_1, M_2, M_3, n\}_N \in E}{E \vdash \{M_1, M_2, M_3, n\}_N : Public}$$

This rule expresses the fact that secret data is safe to transmit over public channels after encryption with a secret key. In English, it means: M_1, M_2, and M_3 are three message components of levels *Secret*, *Any*, and *Public*. N is a secret key. The environment E lists n as a name that may be used as a confounder to encrypt M_1, M_2, and M_3 with N. This means that n has never been used as a confounder in any other message, and so it is fresh and valid to use here. Therefore, this encryption has secrecy level *Public*.

We want to make sure no non-public data is sent over a public channel. This rule expresses that, using the judgment $E \vdash P$, which means that P is a program obeying the secrecy rules when its environment is E.

$$\frac{E \vdash M : Public \quad E \vdash M_1 : Public \quad \ldots \quad E \vdash M_k : Public \quad E \vdash P}{E \vdash \overline{M} \langle M_1, ..., M_k \rangle .P}$$

These and closely related rules ensure that processes don't directly convey their secrets to attackers. However, we are also interested in partial information leakage. One potential scenario is that a protocol participant compares a secret value that he has encrypted against a number of known constants and sends a public message if he finds a match. Clearly this is a breach of secrecy. Luckily, the typed spi calculus prevents it by forcing all equality checks to type check with this rule:

$$\frac{E \vdash M : T \quad E \vdash N : R \quad E \vdash P \quad T, R \neq Any}{E \vdash [M \ is \ N]P}$$

Remember that secret parameters to a system will be treated as variables of type *Any*. Since this is the type indicating no information on secrecy status, the type system will not allow a proof that such a variable has any type but *Any*. Thus, no direct comparison between parameters can occur. Inductively, due to this and other rules, it is also impossible that any variable not of type *Any* has a value conditionally dependent on a parameter.

The remainder of the type system is a straightforward extension of these rules to handle the rest of the language. [1] proves the following theorem to formalize the intuition behind the rules:

Theorem 1 *If E is an environment that assigns every variable the type Any and every name the type Public, then, for every process P such that $E \vdash P$, it follows that P is secure when the variables in E are taken as its secret parameters.*

3.6 The example proved secure *("something new" part 2)*

A process can be proved secure by annotating each binding construct with a type for each variable it introduces and a type and encryption term for each name it introduces. The encryption term can be omitted for names not meant to be used as confounders, in which case the term is taken to be some dummy term. These annotations can be used to guide proofs, since the rules are entirely syntax-directed. The only guidance needed to construct proofs automatically is the types and terms to choose for variables and names for application of binding construct proof rules, and annotations provide these.

For the RPC handshake example, we'll consider our environment E to consist of an *Any* variable x_M and *Public* names c_{ab} and c_{ba} associated with dummy encryption terms. The system will be modified slightly to fit the typed spi calculus' syntactic restrictions on encryption with secret keys. $*$ will be used to stand for an arbitrary expression in unused message positions.

$$
\begin{aligned}
P_A(K_{ab}) \quad \equiv \quad & (\nu n_a : Public)\overline{c_{ab}}\,\langle n_a \rangle\,.c_{ba}(x : Public). \\
& case\ x\ of\ \{y_k : Secret, y_{dummy} : Any, y_n : Public, n_{dummy} : Any\}_{K_{ab}}\ in \\
& [n_a\ is\ y_n]\overline{c_{ab}}(\nu n_{conf} : Any :: \{*, *, n_a, n_{conf}\}_{y_k})\,\langle \{*, *, n_a, n_{conf}\}_{y_k} \rangle\,.c_{ba}(z : Public).\mathbf{0}
\end{aligned}
$$

$$
\begin{aligned}
P_B(K_{ab}) \quad \equiv \quad & c_{ab}(x_n : Public).(\nu n_k : Secret)(\nu x_{conf} : Any :: \{(n_k, *, x_n, n_{conf})\}_{K_{ab}}) \\
& \overline{c_{ba}}\,\langle \{(n_k, *, x_n, n_{conf})\}_{K_{ab}} \rangle\,.c_{ab}(y : Public). \\
& case\ y\ of\ \{x_{dummy1} : Secret, x_{dummy2} : Any, y_n : Public, n_{dummy} : Any\}_{n_k}\ in \\
& [x_n\ is\ y_n](\nu n_{conf2} : Any :: \{*, x_M, *, n_{conf2}\}_{n_k})\overline{c_{ba}}\,\langle \{*, x_M, *, n_{conf2}\}_{n_k} \rangle\,.\mathbf{0}
\end{aligned}
$$

$$
Sys \quad \equiv \quad (\nu z_{ab} : Secret)P_A(z_{ab}) \mid P_B(z_{ab})
$$

ν bindings of n_{conf} and n_{conf2} are used to generate confounders to use with encryption.

Though tedious, it is easy to step through proving $E \vdash Sys$, using the complete set of typing rules. Then, using Theorem 1, we may finally conclude that the system maintains secrecy: no outside attacker can learn anything about the value of x_M when run in parallel with Sys.

References

[1] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[2] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[3] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication, from proceedings of the royal society, volume 426, number 1871, 1989. In *William Stallings, Practical Cryptography for Data Internetworks, IEEE Computer Society Press, 1996.* 1996.