Guessing Attacks in the pi-calculus with a Computational Justification

Tom Chothia

tomc@lix.polytechnique.fr
Laboratoire d'Informatique (LIX)
École Polytechnique (CNRS)
91128 Palaiseau Cedex - France

Abstract. This paper presents an extension of the pi-calculus that can reason about brute force and guessing attacks. We relate new name declarations in the pi-calculus with random sampling in the computational model of security. The scope of a new name can then be expanded at a comparable cost as it would take to guess the randomly sampled value in the computational setting. We provide a function that calculates the cost of a given attack, taking into account the ease with which the attacker can confirm its guesses. We argue the correctness of this calculus by relating it to the computational model of security. We show that if the cost of an attack in the calculus is less than exponential in a security parameter, then there exists a polynomial time Turing machine that can defeat the process with non-negligible probability. On the other hand, if there is no sub-exponential cost attack, then the process is just as safe as its spi-calculus counterpart, and so the use of guessable names does not help the attacker.

1 Introduction

This paper is primarily concerned with guessing attacks on protocols. Randomly guessing values can be an effective way to break a protocol. However, guessing attacks are handled poorly by most formal security analysis techniques, including the spi-calculus [AG97] with its model based on free and bound names. Our aim in this paper is to provide a pi-calculus based model of guessing attacks with most of the simplicity of the formal model and some of the power of the computational model [GM84, BM84].

The spi-calculus based security analysis method use a mini-language with a construct to create new names. These new names can never be guessed, so an attacker must trick a process into giving away its secrets. The computational security model, on the other hand, is based on the random sampling of key values followed by a complexity analysis of the run time of any possible attacker.

In assembling our calculus we equate the new name declaration of the pi-calculus and random sampling in the computational model. In our calculus we write *new* $a : D_n$ to mean that a is a new name, randomly sampled from a domain of size n. To avoid the difficult complexity analysis required by the computational model we only allow the attacker to take advantage of the limited domain size through a guess operation: *guess* $x : D_n$. This operation allows an attacker to correctly guess a value from a domain, but at a cost. An attacking process can use this *guess* operation to try to break a protocol.

The attackers trace maps out a path to be followed later by another brute force attack. A real attacker would have to use trial and error to guess values, look for confirmations of their guesses and be prepared to go back and guess again. We can calculate the cost of this more complicated guessing attack from a single trace of a successful attack in the calculus using a simple function on traces. This function can find the cost of multiple guesses, taking into account exactly when the attacker can get their guesses confirmed. For a given protocol, the higher the value of the minimum cost attack the safer the protocol will be.

We show the correctness of the calculus and the cost function by relating our model back to the spi-calculus and to the computational model. First we add a security parameter for domain sizes. We define a Turing machine attacker that can interact with a process. In our calculus we translate new names as random bit strings chosen from a domain of size equal to the new name's domain size, whereas for the spi-calculus all names are mapped to bit strings with the same length as the security functions, hence they cannot be guessed in sub-exponential time. We also give a safety criterion for these Turing machine attackers. For our first theorem, we show that if there is a sub-exponential cost attack in our calculus then the process is unsafe in the computational model, i.e. there exists a polynomial time Turing machine that can defeat the safety criterion. By adding short guessable names we have changed the spi-calculus model of security and we must be sure that we have not allowed new attacks that fall outside of our model. Our second theorem proves that if there exists a correct mapping for the spi-calculus, then there is also a correct mapping for our calculus. This result shows that in shortening some of the new names in the guessing calculus, we do not allow for any new attacks that cannot be found in the calculus itself.

Work such as [AR00, BPW03, Lau04, MW04] has proved that there is a computationally correct mapping from the formal Dolev-Yao model [DY83] to the computational model. Mitchell, Scedrov et al. [MRST01] limit the pi-calculus to run in polynomial time. So, if a protocol can be broken in this calculus, we know it can also be broken by a polynomial computational attacker. The main difference between their work and the work we present here, is that they limit the non-determinism of a processes to ensure that it runs in polynomial time, whereas we do not limit the run time of the calculi process attacker instead, we ensure that the only action that is of use in a computational attack is the *guess* action and we cost that accordingly.

Zunino and Degano [ZD04] enhance the standard Dolev-Yao attacker so that it can guess a key. Using computational security methods they show that there is a negligible probability of these kind of guesses succeeding and so standard Dolev-Yao attacker is just as powerful as the enhanced one.

There is a great deal of work on the subject of attacks based on guessing poorly chosen passwords, for instance Gong et al. [GLNS93] on the computational side and Lowe [Low02], Delaune and Jacquemard [DJ04] and Corin et. al [CMAFE03] on the formal side. This work focuses on the idea that passwords are often chosen poorly and so can sometimes be guessed by dictionary attacks. Much of this work considers decryption functions to simply map bit strings to bit strings and so, in order to verify the guess of a key from an encrypted message you must have access to that encrypted message and know the message's contents.

In Section 2, we review some background work, including the pi and spi-calculi and the computational security model. Section 3 introduces the pi-calculus with guessing, and the cost function for attacks is giving in Subsection 3.4. We address the correctness of our system in Section 4 and finally we conclude and discuss further work in Section 5. We provide a summary of the calculus's technical details in an appendix. A longer vision of this paper, with extended descriptions and proofs is available at `www.lix.polytechnique.fr/~tomc`.

2 Background: Formal and Computational Analysis

This paper is aimed at combining some aspects of computational analysis methods with formal process analysis in the pi-calculus. So, this section first reviews protocol analysis in the pi and spi-calculi and then outlines how computational methods can be used to prove much stronger results.

2.1 The pi and spi-calculus

The pi-calculus is a miniature concurrent language. This language is simple enough to allow formal analysis while expressive enough to describe most interesting concurrent processes. The exact syntax given to the pi-calculus varies from paper to paper; here we use the following:

$$\begin{array}{rcl} \text{Process } P,Q & ::= & 0 \\ & | & \mathit{send}\ a\langle b\rangle \\ & | & \mathit{rec}\ a(x);P \\ & | & \mathit{new}\ a;P \\ & | & !P \\ & | & (\ P \mid Q\ ) \\ & | & [a=b];P \end{array}$$

The first piece of syntax 0, represents the stopped process. The *send* operation broadcasts the name b over channel a. The next operation receives a name over the channel a and substitutes it for x in the continuing process P. The *new* operation creates a new communication channel. The bang operator ! can perform recursion by spinning off an arbitrary number of copies of a process. The bar | represents two processes running in parallel and finally the match operation, $[a=b];P$ executes P if and only if a is equal to b.

The key reduction rule of the calculus allows two processes to communicate:

$$\mathit{send}\ a\langle b\rangle \mid \mathit{rec}\ a(x);P \longrightarrow P[b/x]$$

One of the most important aspects of the pi-calculus is that new names are both new and unguessable, for instance the process $\mathit{new}\ a;\mathit{rec}\ a(x);P$ can never receive a communication on the channel a, no matter what the surrounding processes might try. This also means that there is no way to write down an attacker that tries every possible value until it "finds" the right one.

In order to model a larger number of interesting protocols the spi-calculus extends the pi-calculus with primitives for encryption. A new term is added, of the form $\{M\}_N$, to mean the message M encrypted with the key N. These terms can be decrypted using an operation of the form: $case\ L\ of\ \{x\}_N\ in\ P$. If a process provides the correct key, the semantics rule for decryption substitutes the encrypted value for the variable:

$$case\ \{M\}_N\ of\ \{x\}_N\ in\ P \longrightarrow P[M/x]$$

Encryption of the message M with the key N is performed by simply writing the term $\{M\}_N$.

The small size and expressivity of the spi-calculus makes the detailed analysis of processes possible. This including the analysis of security properties, as shown by Abadi and Gordon [AG97]. We briefly summarize some of this work here.

An attacker is modelled as a *context*, written $C[\_]$ which is any possible surrounding term, into which the process being attacked may be plugged. This means that the attacker can access any of the process's free names (those not bound by a new or input operator) and use these to communicate with the process in anyway it chooses. The attacker may also perform any computations it wishes using the spi-calculus.

$$\begin{aligned} System(M) &\equiv new\ pwd; (\ Client \mid Server(M)\ ) \\ Client &\equiv new\ reply; send\ ser\langle pwd, reply\rangle \\ &\qquad\qquad \mid rec\ reply\ x; P(x) \\ Server(M) &\equiv\ !rec\ ser\ (x,y); [x = pwd]; send\ y\ \langle\ M\ \rangle \end{aligned}$$

This system is comprised of a client and a server. The server listens for a connection on a public channel, this connection is made up of a password and a reply channel. The server then checks the password and, if it is correct, replies with the message M. The password is private between the client and the server, therefore no attacker may know, or guess, it and hence the message M is safe.

We could prove this formally by showing that the for any M and M', if $P(M)$ has the same visible actions as $P(M')$ then the system $System(M)$ has the same visible actions as $System(M')$ which would in turn imply that for all hostile attackers A, the process $A[Systems(M)]$ has the same visible actions as $A[System(M')]$ and so P cannot leak the message M to the attacker A.

2.2 The Computational Security Model

Implementations of formal processes are susceptible to attacks that are "outside their model". For example, a new name that is implemented as a bit can be correctly guessed with a one in two chance. Whereas, the spi-calculus model of the process might say that this value is a bound name and hence is guaranteed to be unguessable (alternatively the name could be free, then the attacker would always know it).

The computational model of security avoids some of these problems by modelling attackers as polynomial Turing machines and values as bit strings [GM84, BM84]. This means that the attacker can carry out any computational feasible operation. Secret values in this model are randomly chosen from a probability distribution. Given a probability distribution D_n of size n, we write: $x \overset{r}{\leftarrow} D_n$ to mean that x is a value that has been randomly sampled from that distribution.

A security criterion is used to judge the safety of a given process. The choice of which criterion to use will depend on the exact nature of the required security property. However, a typical criterion defines the *attackers advantage* to equal the probability that, for some sampled value, the attacker can correctly identify that value, minus the probability that the attacker incorrectly identifies the value.

Attackers are probabilistic, polynomial time Turing machines (PRTMs), this is a Turing machine that runs in polynomial time in some security parameter and has the ability to make a random choice. It is enough for these attackers to return 0 or 1 depending on what they believe to be the result of their attack.

The chance of the attacker defeating the criterion must become very small, very quickly as the size of the security parameter grows. More formally, we say that the attackers advantage be *negligible*. Where a function f is *negligible* if for all c there exists N such that for all $x > N$ we have that $f(x) < x^{-c}$.

As an example of a simple criterion, one could say that an encryption scheme E is safe if

$$\begin{aligned} Adv(n) \;=\; & Pr[x \overset{r}{\leftarrow} D_n, k \overset{r}{\leftarrow} Key_n : A(n, E_k(x), x) = 1] \\ & - Pr[x, y \overset{r}{\leftarrow} D_n, k \overset{r}{\leftarrow} Key_n : A(n, E_k(y), x) = 1] \end{aligned}$$

is negligible. In this criterion, the attacker is given the length of the security parameter, a random element and an encrypted value. The attacker must answer 1 if it believes that the value it has been giving is the same as the encrypted value and 0 if it believes the values are different.

A stronger, and more realistic criterion might give the attacker the ability to use the encryption algorithm, this is done in the form of an *oracle* and we write $A^{f(\text{-})}$ to mean the attacker has access to an oracle to perform the function f with its chosen input. So, for instance, a criterion to ensure that the attacker does not know when the same value has been encrypted a number of times could be written as

$$Adv(n) \;=\; Pr[k \overset{r}{\leftarrow} Key_n : A^{E_k(-)}(n) = 1] \;-\; Pr[k \overset{r}{\leftarrow} Key_n, x \overset{r}{\leftarrow} D : A^{E_k(x)}(n) = 1]$$

where the false oracle $E_k(x)$ ignores the value the attacker gave it to encrypt and always returns the encryption of x with the key k.

The computational method captures, and hence defends against, a wide range of possible attacks, including guessing attacks. However, proofs in this model can be difficult and are impossible to automate. The aim of our work is to provide some of the benefits of computational analysis of guessing attacks without involving the user in any difficult proofs.

3 The pi-calculus with Guessing

This section introduces the pi-calculus extended with guessing, the pi-g calculus. We do this in a number of stages, in the hope of illustrating the motivation behind the design decisions and elucidating some of the finer details. The first stage introduces the guess action, next we consider the cost of multiple guesses, and thirdly we show that the cost of an attack can be reduced if the attacker can get easy a confirmation of their guess. Finally, we note that only certain kinds of actions give reliable confirmations and we build our cost function accordingly.

3.1 A Guessing Rule

Given a protocol with a fixed password, there is a difference between finding a general guessing attack on the protocol and an attack that guesses the password for one particular run of the protocol. The pi-calculus handles this quite neatly, by using a new name for the secret value, for instance, a system in which processes P and Q share a password pwd against an attacker A could be written as:

$$new\ pwd;(\ P \mid Q\ ) \mid A$$

This marks out the password as important to the correctness of the process and ensures that the attack can never come up with the name without being told. However, this can sometimes entrust too much security in the new name; a password consisting of a single bit, for instance, can be easily guessed.

As mentioned above, the computational model randomly samples these names. In this setting we would write:

$$pwd \overset{r}{\leftarrow} D_n\ :\ P(pwd) \mid Q(pwd) \mid A$$

This stops the attacker from just knowing the password at the start of the attack because the same attack must work for different, randomly sampled values of pwd. This model does allow for a brute force attack, however it forces anyone using this method to perform a statistical analysis on the run time complexity of A.

To bring these two methods closer we work in the pi-calculus and allow a new name to be sampled from a domain of a given size, writing

$$new\ pwd : D_n;(\ P \mid Q\ ) \mid A$$

to mean pwd is a new value, shared between P and Q, sampled from a domain of size n. To avoid the need to do nasty analysis we force the attacker to declare when it is attempting to guess a value and pay a cost proportional to the size of the domain. With the reduction rule

$$\begin{aligned} &new\ pwd : D_n;P \mid guess\ x : D_n;A \\ &\qquad \rightarrow_{pwd:n} new\ pwd : D_n;(P \mid A[g_{pwd}/x]) \end{aligned}$$

which means that the attacker guesses the name pwd at a cost of n.

The distinct name g_{pwd} is a correct guess of the name pwd. We do not substitute pwd for x directly as we will later need to find out when a guess is confirmed. This label on the reduction, as with all the other labels, records information necessary for calculating the cost of the attack; no label ever affects the reduction of a process. A new name declared of the form $new\ a : D_n$ binds both a and g_a, furthermore we only allow names of the form g_a to be declared with the $guess$ operator.

Whereas before, in the pi-calculus, an attacker could only acquire knowledge of a new name by being told it, now the attacker can also use the guess action and pay the cost of a brute force attack on a value of the given domain size.

3.2 Multiple Guesses

Now we have allowed the attacker to make a single guess we must calculate the cost of multiple guesses. Below we give a simple system in which a process shares a 128 bit password with another process Q. An attacker A will try to guess this password one bit at a time.

$$\begin{aligned} Process \equiv\ & new\ chn, rew, b_1 : D_2 ... b_{128} : D_2; (send\ a\langle chn, rew\rangle \mid rec\ chn(x_1); [x_1 = b_1]; ... \\ & ... rec\ chn(x_{128}); [x_{128} = b_n]; send\ rew\langle reward\rangle) \mid Q\ ) \end{aligned}$$

$$Attacker \equiv rec\ a(chn, rew); (!(guess\ b : D_2; send\ chn\langle b\rangle) \mid rec\ rew(x))$$

The password process announces a secure channel, *chn*. It then listens on this new channel for the bits of the password. One by one, the received bits are tested against the bits of the true password and if they all match the *reward* is broadcast.

The attacker A cannot know the bits of the password because they are new names bound to P and Q. However, it can guess each bit with a cost of 2 (or a 1 in 2 chance of being correct). After the attacker has made a guess at all 128 bits of the password it will have a 1 in 2^{128} chance of having correctly guessed the password and hence the cost of these guesses should be 2^{128}. So, in this case, we multiply the cost of each successive guess. It should be noted that the process A does not perform an attack of cost 2^{128} on the process P, rather 2^{128} is the cost of performing a successful attack along the lines of the attack attempted by A.

3.3 Confirmation of a Guess

Multiplying the costs of all guesses in a trace will sometimes overestimate the cost of an attack. The multiplication of the costs reflects the idea that all possible combinations of guesses must be tried. However, if the attacker can get one of their guesses confirmed as correct, when only part of the way through the attack, then the confirmed guess will not have to be retried and will therefore not contribute to the cost of future guesses.

For example, consider the fairly moronic extension to our prior password system given below. The processes function in the same way as before, except this time an acknowledgment is broadcast after receiving each bit.

$$\begin{aligned} Process \equiv\ & new\ chn, rew, b_1 : D_2 ... b_{128} : D_2; \\ & (send\ a\langle chn, rew\rangle \mid rec\ chn(x_1); (\text{ send ack } \mid [x_1 = b_1]; (... \\ & ... rec\ chn(x_{128}); (\text{ send ack } \mid [x_{128} = b_{128}]; (send\ rew\langle reward\rangle)...) \mid Q) \end{aligned}$$

Because the system tests each bit before listening for the next bit these acknowledgments confirm the guess of the previous bit as correct. This means that any brute force attack based on guessing and then listening for an acknowledgement would only have to try a guess at each bit once making the total cost for the new attacker $128 + 1$. These confirmations effectively prune the search tree of all the possible combinations of guesses.

To allow dependences to be tracked we extend the syntax of the calculus with a dependence marker. This marker takes the form: $(g_a)P$ and behaves in exactly the same way as the processes P. The point of the marker is to record that the process P is only running because the guess g_a has been shown to equal a. The key semantic rule for this is:

$$[g_a = a];P \rightarrow_{(a)} (g_a)P$$

The label (a) signals that this action build a dependence on the guess being correct.

A guess is confirmed when a signal that depends on the guess being correct passes from the process to the attacker. So, we make the attacker and victim explicit by separating them using a double bar $||$. The following reductions illustrate both these mechanisms:

$$\begin{array}{lll}
\mathit{new}\ a{:}D_n;\ \mathit{rec}\ b(x);[x=a];\mathit{send}\ b\langle c\rangle & \rightarrow_{a:n} & \mathit{new}\ a{:}D_n;\ (\ \mathit{rec}\ b(x);[x=a];\mathit{send}\ b\langle c\rangle \\
\quad ||\ \mathit{guess}\ y{:}D_n;\mathit{send}\ b\langle y\rangle\ |\ \mathit{rec}\ b(x);A & & \qquad ||\ \mathit{send}\ b\langle g_a\rangle\ |\ \mathit{rec}\ b(x);A) \\
 & \rightarrow & \mathit{new}\ a{:}D_n;([g_a=a];\mathit{send}\ b\langle c\rangle\ ||\ \mathit{rec}\ b(x));A \\
 & \rightarrow_{(a)} & \mathit{new}\ a{:}D_n;(\ (g_a)\mathit{send}\ b\langle c\rangle\ ||\ \mathit{rec}\ b(x);A\ ) \\
 & \rightarrow_{a:\overline{b}\langle\vec{c}\rangle} & \mathit{new}\ a{:}D_n;\ (\ 0\ ||\ A\ )
\end{array}$$

The name a, from a domain of size n, is only known by the process on the left hand side of the double bar. In the first reduction the attacker guesses this name, hence the reduction is labelled with $a : n$ to indicate that the name a has been guessed at cost n. The second step is a communication between the attacker and the process. Now, we come to the match $[g_a = a]$. This match will go ahead as, for the sake of computing the cost, we consider the guesses to be correct. However, we must leave a dependence marker to say that the resulting process is only running because a guess has been shown to be correct. In the final step another communication happens and as the *send* is dependent on a guess of a being correct and as the communication takes place between the main process and the attacker, the guess of the name a is possibly confirmed by the communication of $\vec{c}$ over the channel b. We indicate this by the $a : \overline{b}\langle\vec{c}\rangle$ label on the reduction. It is left to the cost function to work out if the confirmation is reliable. This results in the rule for confirmations:

$$(\overrightarrow{g_a})\mathit{send}\ b\langle\vec{c}\rangle\ ||\ \mathit{rec}\ b(\vec{x})A \rightarrow_{(\vec{a}:\overline{b}\langle\vec{c}\rangle)} P\ ||\ A[b/x]$$

Communication is performed in the same away as the pi-calculus, except the dependence marker can be passed from one part of the system to another so as to catch indirect comfirmations.

$$(\overrightarrow{g_c})\mathit{send}\ a\langle\vec{b}\rangle\ |\ (\overrightarrow{g_d})\mathit{rec}\ a(\vec{x});P \rightarrow (\overrightarrow{g_c},\overrightarrow{g_d})P[b/x]$$

A final semantics rule, not given here, allows any of the other rules to be applied inside an unguarded context.

The dependence markers are added to the calculus purely to aid the analysis of a process. They would not be present in any implemented system. When an attacker receives a message from an action of the form $(g_a)\mathit{send}\ a\langle b\rangle$ they only see the $\mathit{send}\ a\langle b\rangle$

part. So, if along with a confirming signal there are other similar signals that do not confirm a guess, the attacker could not be sure which one they have received and so the signal does not definitely show the guess to be correct. As an example, we ask the reader to again consider the password process given above. There, a correct guess is confirmed by a signal on the channel *ack*. However, if we added a continually repeating output on the channel *ack* in parallel with the orginal process then the receipt of an *ack* signal by the attacker would be meaningless and would not help it prune down the search tree of possible guesses. So, when the cost function is presented with a possible confirmation it uses the state of the process from when the dependence was first built, to test if the confirming action would be possible without the guess being correct.

3.4 Cost of a Trace

When an attacker finds a guessing attack for a protocol, the attacking process does not have to model the ways in which it may try different guesses and keep track of the guesses already made. This would lead to a more complicated system, which would require the kinds of analysis seen in the computational model. Instead, we restrict the only type of computation that can be used to find secret names to the *guess* action. We can then calculate the cost of these guess actions separately. It should be noted that we do not calculate the chance of the attacking process being correct. Rather, our cost function calculates the cost in terms of computing power or number of tries, a brute force attacker would need to successfully follow the same path as the attacking pi-g calculus process. The length of the trace cannot be a function of the security parameter. Therefore we only considered traces of finite length, where a polynomial or exponential cost of an attack comes from the use of the guess action.

The cost of guesses are calculated and added to the total as they are confirmed and the cost of the unconfirmed guesses is calculated at the end of the trace. We calculate the cost of a trace, defined by the grammar $T ::= P \rightarrow_{\alpha} T|P$, using an auxiliary function that has a list of current guesses and another list that stores the states in which the dependences where first build.

$$cost\ of\ trace(\ T\ ) = cost(\ T, [\,], [\,]\ )$$

The *cost* function has five cases. The first is that of a guess being made by the attacker.

$$cost(P \rightarrow_{a:n} T, gu, comf) = cost(T,\ (a,n); gu, comf)$$

The guess and its cost are added to the list of current guesses, *gu*. In the second case a dependence on a guess being correct is built up inside the protocol and we add the process state to the *comf* list.

$$\begin{array}{ll} cost(P \rightarrow_{(a)} T, gu, comf) = & \\ \quad cost(T,\ gu,\ (a:P); comf) & if\ (a:Q)\ \notin comf \\ \quad cost(T,\ gu, comf) & if\ (a:Q) \in comf \end{array}$$

The confirmation rule is the most complicated. To simplify matters we introduce a short hand for the produce of the costs (the second elements) of the list of current guesses: $\Pi\ snd\ [(a_1,n_1),...,(a_i,n_i)] = n_1 \times n_2 \times ... \times n_i$. The action tag $(\vec{a},\overline{b}\langle\vec{c}\rangle)$ tells us that guesses of the elements of the set $\vec{a}$ are potentially being confirmed by a communication of the names $\vec{c}$ over the channel b. We then check in the confirmation list to see if the system could have made the same communication if each guess had been wrong. If it could not have been then the confirmation is real.

$$\begin{aligned} cost(P \rightarrow_{(\vec{a},\overline{b}\langle\vec{c}\rangle)} T, gu, comf) = {} & \Pi\ snd\ (gu[(a_i,m_i-1)/(a_i,m_i)]) \\ & +\ cost(T, gu \backslash \{(a_1,m_1)...,a_j,m_j)\}, comf) \end{aligned}$$

where $a_1,...,a_j$ are all the names in $\vec{a}$ such that
$(a_i,Q) \in comf$ and $(a_i,m_i) \in gu$ and $new\ d;Q\{d/g_{a_i}\} \not\Rightarrow C[send\ b\langle\hat{c}\rangle]$

If the attacker knows the contents of the message c, the attacker can use this to look for confirmations, so $\hat{c} = \vec{c}$. However, if any of $\vec{c}$ are secret names within the protocol then the attacker will not be able to distinguish it from any other secret name from the same domain. So that element of $\hat{c}$ can be any bound name that is unknown to the attacker and is drawn from the same domain. The cost produce by the confirmation is the produce of the costs of each guess, with one subtracted from all the confirmed guesses (as one path for each confirmed guess will continue) plus the cost of the rest of the trace, with the confirmed guesses removed from the list of current guesses. It would be possible for the confirmation action to come from a second dependency not from the state Q, however Q could reduce to the process that made this second dependence and so could perform the same actions.

It is unnecessary for this rule to confirm a guess of the messages contents. If the broadcast of this message was dependent on a guess of c then the guess of c will be automatically confirmed. If the broadcast was not dependent on the guess then the attacker can read the value of c from this action and so does not have to guess it at all.

We do not give the attacker full control over the protocols scheduler; as we do not wish to find attacks that only work for particular, pathological schedulers. We also do not demand that the scheduler gives equal weight to each possible action, as this is an unrealistic assumption. Rather, our aim is to find attacks that will work for any reasonable scheduler on which a protocol may be run, so we define our system for any scheduler that assigns a non-zero probability to any possible action.

It would be quite possible for an attacker to make a guess and wait for an action it believes confirms a guess but in fact receive a reply from a different reduction that does not confirm the guess at all. For now, we say that it is unsafe for the security of the processes to depend on the scheduler avoiding these confirming states, but we will also give a computational argument for this in Section 4 that proves, that for any fair scheduler, this definition of a confirming state is enough to correctly find a process insecure.

When we come to the end of a trace we multiply the remaining guesses to get the final cost.

$$cost(P, gu, comf) = \Pi\ snd\ gu$$

If a reduction does not fit into any of the above categories it has no effect on the cost, as shown by the last case.

$$cost(P \rightarrow T) = cost(T, gu, comf)$$

3.5 Encryption

Encryption is an essential part of many interesting protocols. We can add encryption to our calculus in the manner of the spi-calculus, by adding the name $\{a\}_k$ to mean the name a encrypted with the key k, and adding the operator *decrypt* a *as* $\{x\}_k;P$ to decrypt, encrypted messages. Decryption adds another way in which a guess can be confirmed and dependences built up. Firstly, when a protocol uses decryption with a real key and a guess it sets up a dependence:

$$\begin{array}{c} decrypt\ \{b\}_a\ as\ \{x\}_{g_a};P \rightarrow_{(a)} (g_a)P[b/x] \\ decrypt\ \{b\}_{g_a}\ as\ \{x\}_a;P \rightarrow_{(a)} (g_a)P[b/x] \end{array}$$

Secondly, an attacker's guess at a key can be confirmed directly by attempting to decrypt a message encrypted with the real key. This leads to a secondary decryption rule just for the attacker:

$$decrypt\ \{b\}_a\ as\{x\}_{g_a};A \rightarrow_{(g_a,0)} A[b/x]$$

The 0 indicates that this action always confirmation the guess and these action is treated as an exception by the cost function.

As mentioned in the introduction, some formal definitions would not consider this a correct confirmation of g_a if the attacker did not already know the encrypted value b. However, the truth of this assumption will depend on the exact implementation. So we err on the side of caution, and allow the attacker to confirm their guess by decryption alone.

4 The Correctness of the pi-g Calculus

This section gives a computational base to the calculus. We allow domain sizes to be parameterized on a security parameter and then show that if there is a successfully, finite attack in the pi-g calculus with less than exponential cost in the security parameter, then the process is unsafe in the computational setting. In particular, it can be defeated by a brute force guessing attack. This result justifies the design of the cost function given in Subsection 3.4, and in particular the reduction in the cost of an attack due to the confirmation of a guess.

The counterpart to this theorem, that the lack of a sub-exponential cost attack implies safety in the computational model, is harder to prove because the Turing machine attacker may be able to carry out attacks outside the model of the calculus. We could prove this safety theorem by way of the computational correctness of the spi-calculus. However, we are only interested in showing that the guessing extensions to the spi-calculus are correct. So, we show that the safety of the spi-calculus is enough to prove the safety of the pi-g calculus, and hence our extensions do not introduce any new errors.

We wish to pit a Turing Machine attacker against a protocol written in the process calculus. We do this by modelling the attacker as a Turing machine with an oracle that can run the process being attacked any number of times, for this we write A^P or $A^{P(\vec{a})}$ for the process P parameterise on the names $\vec{a}$.

The oracle can compute reductions of the calculus process P using the semantic rules for the calculus. The Turing machine A interacts with the oracle by means of two tapes, one for output, onto which the oracle will write outputs visible to the attacker and another tape for input, from which the oracle can read messages from the attacker and insert them in to the appropriate part of the process. Names are implemented as random bit strings that are at least as long as the security parameter, hence making them hard to guess in sub-exponential time. The process P would have no other access to the security parameter, unlike the attacker. To start a new run of the process the attacker writes a special symbol onto the tape along with a bit string. Appending this bit string then distinguishes the bit string names of this new concurrent run. Mitchell et al. describe a simple implementation of the pi-calculus as a Turing machine [MRST01] where, unlike us, they aim to show the polynomial time reductions of a sub-set of the pi-calculus.

Definition 1. *For a spi-calculus process P the term $A^{(P_n)}$ is the Turing machine A which has access to an oracle that behaves in a similar manner to the spi-calculus process P, as outlined above, in particular all bound names in P are mapped to bit strings at least as long as the security parameter n.*

It should be noted that the behaviour of the encoding of P will not exactly match the behaviour of the spi-calculus process, because the attacking Turing machine A can find a bound name in exponential time. Instead, we match safety in the spi-calculus with safety against a polynomial time attacker in the computational setting.

To formalise exactly what we mean by a successful attack we consider the set of spi-calculus processes that are parameterised on a given constant $P(c)$. In the spi-calculus, a process is considered safe when the process $P(c_1)$ indistinguishable from the process $P(c_2)$ for all names c_1 and c_2. To relate this definition of safe to the computational model we reformalise this to say that a process $P(c)$ is safe if and only if there does not exist a process $A(\_)$ such that $P(c) \mid\mid A(c)$ performs an output on c and $P(a) \mid\mid A(c)$ does not perform an output on c.

In the computational setting the attacker is given access to an oracle with a secret bit string, and the attacker is also given another, possibly different bit string. The attacker will answer "1" if it believes the value it was given is the same as the secret and "0" otherwise. The difference between these probabilities, of the attacker getting it right and getting it wrong, is the attackers advantage:

$$Adv(n) = Pr[s \overset{r}{\leftarrow} D_n \ : \ A^{P_n(s)}(n, fn(P), s) = 1] - Pr[s,t \overset{r}{\leftarrow} D_n \ : \ A^{P_n(s)}(n, fn(P), t) = 1]$$

The attacker has access to the free, public names of the process. We slightly abuse our notation here by using letters to represent names in the calculi processes and the bit strings that represent those names in the computational setting.

We say that the process P is safe in the computational setting if the advantage function $Adv(n)$ is negligible for all probabilistic, polynomial time Turing machines (PRTMs) A, otherwise we say it is unsafe. As mentioned above, we are interested in finding attacks on protocols that work in any reasonable situation, therefore we require the attacker A to work with non-negligible probability for any scheduler than assigns a non-zero, constant probability to any possible action. Processes in the pi-g calculus can be treated in the same way. Except pi-g calculus names are mapped to random bit-strings with the same length as the size of their domains.

Now that we have a model in which a protocol can be attacked, we need to be sure that a successful attack in the model implies the possibility of a successful attack in an implementation of the protocol. We show this by proving that, if there exists an attack on a protocol in the pi-g calculus then the computational equivalent system can be broken by a polynomial time Turing machine.

Theorem 1. *Given a process with a secret value P(s), if there exists a sub-exponential cost, finite attack on the process in the pi-g calculus then there also exists a probabilistic, polynomial Turing machine that makes the advantage function $Adv(n)$ non-negligibly. In particular this attack takes polynomial time multiplied by the cost of the attack in the pi-g calculus.*

Proof. (Sketch)

Assuming the existence of a sub-exponential cost attack in the pi-g calculus; we sketch the construction of a polynomial time Turing machine that will find the secret value with small but non-negligible probability.

The attacking Turing machine will intercept the same outputs and send the same inputs to the oracle as attacking pi-g calculus process sent to the original process. The trace generated in the pi-g calculus represents just one possible reduction of the process. However each action was found in a finite number of steps so there is a non-negligible probability that the protocol will behave in the same way as it did in the pi-g calculus trace.

When a pi-g calculus process guesses a value from domain of size n, the attacking Turing Machine will make n copies of itself. Each of these copies then reruns the protocol to get to the same point in the attack. These reruns can be done in polynomial time, in the security parameter, and with a non-negligible chance of successes.

When one of the Turing machines sees an action that verifies a guess in the pi-g calculus attack, that machine assumes that its guess is correct and halts all the machines that corresponded to other guesses. There is a non-negligible probability that the guess has been correctly confirmed.

So, we have a parallel Turing machine that mimics the attack found in the pi-g calculus and takes polynomial time multiplied by the cost of the attack in the pi-g calculus. Hence, if the cost for the attack is sub-exponential, we have the Turing machine required by the computational criterion and the computation process is unsafe.

By extending and changing the spi-calculus model to make the pi-g calculus there is a possibility that we have allowed a new class of attacks that are outside the model of the pi-g calculus but are accounted for by the spi-calculus. This is especially possible with the switch from representing values as bit strings at least as long as the security parameter, as we do in the spi-calculus encoding, to representing values as possibly constant length strings, as we may in the pi-g calculus. We address this in our second theorem:

Theorem 2. *If, for a spi-calculus process, the advantage function $Adv(n)$ is negligible for all PRTMs only when there exist a successful spi-calculus attacker then:*

if a pi-g process P does not admit a sub-exponential cost attack then the advantage function pi-g $Adv(n)$ is negligible for all PRTMs.

Proof. (Sketch)

Let P_g be a pi-g process for which there are no sub-exponential attack. Assume, for contradiction, that there exists an PRTM A such that $Adv(n)$ is non-negligible

As the cost of the attack is sub-exponential we know that the attacking process only guesses a finite number of sub-exponential names. So, we can write the process P_g as *new* $a_1 : D_{n_1}, .., a_i : D_{n_i}; Q$ where a_1 to a_i are the names guessed by the attacker. We use structural equivalence to unwind any replications that generated the guessed names.

As the PRTM A breaks P_g, we can construct another machine, A_2 that breaks Q, by giving the true values of a_1 to a_i to A. However the part of Q involved in the attack can be considered as a spi-calculus process, and as we are assuming that the computational encoding of the spi-calculus is correct, there must exists a spi-calculus attacker that defeats Q, say R. Then the pi-g calculus process *guess* $a_1 : D_{n_1}, .., a_i : D_{n_i}; R$ defeats P_g with sub-exponential cost, giving us our contradiction.

5 Conclusions

We have presented an extension of the pi-calculus that can model simple guessing attacks. The new name operator in the pi-calculus is equated with random sampling in the computational model with the result that a new name can be guessed. We only allow a correct guess to originate from a *guess* operator, this simplifies the cost analysis of attacks. In order to show correctness we related our work to the computational model. Our first theorem proved that if the cost of an attack in our calculus is less than exponential then the process is unsafe because we can construct a PRTM attacker that defeats the process with non-negligible probability. Our other main result showed that if there are no sub-exponential attacks on a process then it is safe from guessing attacks.

Further work will include proving the computational correctness of the spi-calculus. This would follow the same lines as previous work on the computational correctness of the Dolev-Yao model.

An interesting extension of the calculus would be to allow the protocol and attacker to directly use the security parameter. It would then be necessary to limit the calculus to polynomial run time in much the same way as Mitchell et al.'s polynomial-time pi-calculus [MRST01]. Another interesting direction might be to make the protocols scheduler explicit and to look for probabilistic conformations for particular schedulers.

We are particularly interested in using this calculus to distinguish between real secure systems where the leaking, or guessing, of a small number of values leads to the entire system being compromised and secure systems in which all secret values must be guessed in order to compromise the whole system. An automated checking system would make the analysis of large protocols in the pi-g calculus much more practical.

Bibliography

[AG97] 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.

[AR00] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *International Conference on Theoretical Computer Science (IFIP TCS2000)*, 2000.

[BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13:850–864, 1984.

[BPW03] M. Backes, B. Pfitzmann, and M. Waidner. A composable cryptographic library with nested operations. In *10th ACM Conference on Computer and Communications Security (CCS)*, 2003.

[CMAFE03] R. Corin, S. Malladi, J. Alves-Foss, and S. Etalle. Guess what? Here is a new tool that finds some new guessing attacks. In *Workshop on Issues in the Theory of Security (WITS)*, 2003.

[DJ04] S. Delauune and F. Jacquemard. A theory of guessing attacks and its complexity. Technical report, ENS de Cachan, 2004.

[DY83] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[GLNS93] L. Gong, M. A. Lomas, R. M. Needham, and J. H. Saltzer. Protecting poorly chosen secrets from guessing attacks. *IEEE Journal on Selected Areas in Communications*, 11(5):648–656, 1993.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

[Lau04] Peeter Laud. Symmetric encryption in automatic analyses for confidentiality against active adversaries. In *Proceedings of 2004 IEEE Symposium on Security and Privacy*, pages 71–85, 2004.

[Low02] Gavin Lowe. Analysing protocols subject to guessing attacks. In *Workshop on Issues in the Theory of Security (WITS)*, 2002.

[MRST01] J. Mitchell, A. Ramanathan, A. Scedrov, and V. Teague. A probabilistic polynomial-time calculus for analysis of cryptographic protocols. In *Mathematical Foundations of Programming Semantics*, 2001.

[MW04] D. Micciancio and B. Warinschi. Soundness of formal encryption in the presence of active adversaries. In *Proceedings of the Theory of Cryptography Conference*, pages 133–155. Springer, 2004.

[ZD04] Roberto Zunino and Pierpaolo Degano. A note on the perfect encryption assumption in a process calculus. In *Foundations of Software Science and Computation Structures: 7th International Conference, FOSSACS 2004*, 2004.

A Technical Summary

$$\begin{array}{lcl}
\text{Network} & ::= & P \mid\mid A \\
 & & \\
\text{Processes } P, Q, A & ::= & 0 \\
\text{\& Attackers} & \mid & \mathit{send}\ a\langle\vec{b}\rangle \\
 & \mid & \mathit{rec}\ a(\vec{x}); P \\
 & \mid & !P \\
 & \mid & (\ P \mid Q\ ) \\
 & \mid & [a = b]; P \\
 & \mid & \mathit{new}\ a : D_n; P \\
 & \mid & \mathit{guess}\ x : D_n; P \\
 & \mid & \mathit{decrypt}\ a\ \mathit{as}\ \{x\}_k; P
\end{array}$$

In general only attackers will use the *guess* operation and only processes will use secret names. The full semantics of the calculus is given in Figure 1.

$$(\vec{g_c})\mathit{send}\ a\langle\vec{b}\rangle \mid (\vec{g_d})\mathit{rec}\ a(\vec{x}); P \rightarrow (\vec{g_c}, \vec{g_d})P[b/x]$$

$$(\vec{g_a})\mathit{send}\ b\langle\vec{c}\rangle \mid\mid \mathit{rec}\ b(\vec{x})A \rightarrow_{(\vec{a}:\overline{b}\langle\vec{c}\rangle)} 0 \mid\mid A[b/x]$$

$$[a = a]P \rightarrow P \qquad\qquad [g_a = a]P \rightarrow_{g_a} (g_a)P$$

$$\mathit{new}\ a : D_n; (P \mid\mid \mathit{guess}\ x; A) \rightarrow_{a:n} \mathit{new}\ a : D_n; (P \mid\mid A[g_a/x])$$

$$\mathit{decrypt}\ \{b\}_a\ \mathit{as}\ \{x\}_a; P \rightarrow P[b/x] \qquad\qquad \mathit{decrypt}\ \{b\}_a\ \mathit{as}\{x\}_{g_a}; A \rightarrow_{(g_a,0)} A[b/x]$$

$$\mathit{decrypt}\ \{b\}_a\ \mathit{as}\ \{x\}_{g_a}; P \rightarrow_{(a)} (g_a)P[b/x] \qquad\qquad \mathit{decrypt}\ \{b\}_{g_a}\ \mathit{as}\ \{x\}_a; P \rightarrow_{(a)} (g_a)P[b/x]$$

$$\text{For unguarded } C[\_] \text{ if } P \rightarrow_\alpha P' \text{ then } C[P] \rightarrow_\alpha C[P']$$

Fig. 1. The Semantics of the pi-g calculus

Structural equivalence is the smallest congruence relation satisfying the following equalities:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}l}
P \mid Q & \equiv & Q \mid P & (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\
P \mid 0 & \equiv & P & \mathit{rec}\ a(\vec{b}); P & \equiv & \mathit{rec}\ a(\vec{c}); P[\vec{c}/\vec{b}] & \vec{c} \cap fn(P) = \{\} \\
!P & \equiv & P \mid !P & \mathit{new}\ a : D_n; P & \equiv & \mathit{new}\ b : D_n; P[b/a] & \{b, g_a\} \cap fn(P) = \{\} \\
\mathit{new}\ a : D_n; 0 & \equiv & 0 & \mathit{new}\ a : D_n; P \mid Q & \equiv & \mathit{new}\ a : D_n; (P \mid Q) & \{a, g_a\} \cap fn(Q) = \{\}
\end{array}$$

$$\mathit{new}\ a : D_n; \mathit{new}\ b : D_n; P \equiv \mathit{new}\ b : D_n; \mathit{new}\ a : D_n; P$$

$$cost(P \rightarrow_{a:n} T, gu, comf) = cost(T,\ (a,n); gu, comf)$$

$$\begin{array}{ll} \multicolumn{2}{l}{cost(P \rightarrow_{(a)} T, gu, comf) =} \\ \quad cost(T,\ gu,\ (a:P); comf) & if\ (a:Q)\ \notin comf \\ \quad cost(T,\ gu, comf) & if\ (g_a : Q) \in comf \end{array}$$

$$\begin{aligned} cost(P \rightarrow_{\alpha} T, gu, comf) = \Pi\ snd\ (gu[(a_i, m_i - 1)/(a_i, m_i)]) \\ +\ cost(T, gu \backslash \{(a_1, m_1) \ldots, a_j, m_j)\}, comf) \end{aligned}$$

where if $\alpha = (\vec{a}, \overline{b}\langle\vec{c}\rangle)$ then $a_1, \ldots, a_j$ are all the names in $\vec{a}$ such that
$(a_i, Q) \in comf$ and $(a_i, m_i) \in gu$ and $new\ d; Q\{d/g_{a_i}\} \not\Rightarrow C[send\ b\langle\hat{c}\rangle]$
and if $\alpha = (a, 0)$ then $a_1 = a$

$$cost(P, gu, comf) = \Pi\ snd\ gu \qquad cost(P \rightarrow T) = cost(T, gu, comf)$$

Fig. 2. The Cost of a Trace

A Note on Encryption

We require that the encryption function used in the computational model is repetition concealing, which-key concealing and message-length concealing. This extremely strong kind of encryption is termed type-0 security by Abardi and Rogaway [AR00] and is defined as making the following criterion negligible for all PRTMs.

$$\begin{aligned} Adv\_Enc(n) \ =\ & Pr[k, k' \overset{r}{\leftarrow} Keys_n : A^{E_k(-), E_{k'}(-)}(n) = 1] \\ -\ & Pr[k \overset{r}{\leftarrow} Keys_n : A^{E_k(0), E_k(0)}(n) = 1] \end{aligned}$$

Encrypting a key with itself can be problematic, even when done indirectly [GM84]. So, we deviate slighting from the original spi-calculus and automatically consider processes which do this unsafe.

In later work it might be interesting to consider an extension of our current system that would allow the attacker to easily guess a key from a message containing and encrypted by that key, or even a series of messages that indirectly encrypt one key with itself.