Formal Verification Of Cryptographic Protocols

Brendan Gordon and Ted Morley

April 2017

1 Introduction

The design of security protocols is a process fraught with error, as can be seen throughout their history. The methods by which cryptographic protocols are broken often work by taking advantage of very subtle flaws. In fact, many examples exist in the literature of protocols used for years, or even decades, before major flaws were revealed. In light of these issues, and the growing importance of cryptography in the modern world, the verification of such protocols becomes a matter of utmost importance.

In our paper, we give a critical overview of technologies on the forefront of this field, by examining a soon to be presented paper by Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet, titled "Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach". (Kobeissi, Bhargavan, & Blanchet, 2017)

Through their paper, the researchers introduce a novel methodology for the development of cryptographic messaging protocols, which requires no preexisting knowledge of formal verification techniques. They use this to demonstrate the existence of novel attacks on the widely used Signal protocol.

For the purpose of explaining the techniques and tools used in the paper, we will use the example of the Denning-Sacco protocol in order to explain how each of the technologies used within the paper would interact with it. It is our hope that this paper provides a fair view of ProScript at its current state of development, as well as an overview of the ProVerif model checker on which it relies.

1.1 Methodology

The technologies in the paper serve as a proof of concept for automated cryptographic verification which does not require direct construction of a model by the user.

Instead, the user writes their protocol in a language called ProScript – A functional subset of JavaScript – which automatically generates a model within a variant of the Applied Pi-Calculus. The Pi Calculus is a formal language which describes the actions of processes, and is widely used in the modelling of cryptographic protocols. This Applied Pi-Calculus model can then be used

without modification with another tool developed by Blanchet, called ProVerif, which automatically generates proofs of attacks against the protocol under a symbolic model. These proofs correspond directly to traces of the attacks, which can be used by the programmer to fix the protocol.

Additionally, the Applied Pi-Calculus model can be manually modified in order to be given as input to another existing tool by Blanchet, CryptoVerif, which also generates proofs of attacks automatically, but under the computational model.

1.2 Denning-Sacco Protocol

A brief overview of the Denning-Sacco protocol is given here, in order to familiarize readers, and to serve as reference throughout the paper. However, we've simplified it by ignoring the timestamps, which makes it a bit easier to represent.

The protocol is designed for the purpose of mutual authentication. Two users who wish to communicate with each other would want to verify that each person is indeed who they say they are. We'll call these two people Abagail and Basha, or A and B. Abagail and Basha each have their own and the other's public keys (Pk_a, Pk_b) and public signing keys (Psk_a, Psk_b). They also have their own private keys (sk_A, sk_B) and private signing keys (ssk_a, ssk_b). Let's see how Abagail and Basha would open up a channel of communication to discuss their research project.

First, Abagail takes Basha's public key, and a key k which Abagail generates. We assume that Abagail has access to Basha's public key. So then Abagail uses an asymmetric encryption function, aenc, and uses Basha's public key to encrypt a message containing k, which is signed with Abagail's secret signing key. This encrypted signed message is sent to Basha.

Basha now decrypts it, and uses Abagail's public signing key to verify Abagail's signature, before sending back a message m encrypted with the key (k) which Abagail sent to her!

Now they have set up a channel, and can talk about their research to their heart's content!

2 ProScript

2.1 What is ProScript?

The main focus of Kobeissi, Bhargavan, and Blanchet's paper is to introduce ProScript through an analysis of the Signal Protocol. ProScript is a strict subset of JavaScript, which limits users to a purely functional syntax.

2.2 Why is ProScript Useful?

ProScript is the front-facing facet of the writers' desire to create, as they state in their abstract 'A novel methodology that allows protocol designers, imple-

menters, and security analysts to collaboratively verify a protocol using automated tools.' (Kobeissi et al., 2017)

ProScript (And, by extension, its compiler) serves as a prototype technology which gives the lay programmer access to the powerful tools of formal verification for cryptographic protocols. It requires no preexisting knowledge of formal verification methods in order to use, since once the user writes the desired cryptographic protocol in ProScript, the process of a model being created and checked against various attacks is all automated.

Due to this, the novel methodology proposed by the researchers becomes clear. Programmers implement a protocol in ProScript, receive an automatically generated model in ProVerif, and attacks are tested against it using ProVerif, revealing possible flaws via generated symbolic proofs.

2.3 Syntax and Semantics

The purely functional nature of ProScript is not simply a matter of taste, it serves an important function. Previous work in automatic verifiers encountered issues in state-space explosion due to having to account for memory being worked with in a more concrete fashion (Blanchet, 2001). On the other hand, the functional paradigm allows for the researchers to sidestep this issue, and create a very efficient translation from ProScript to model in the Pi Calculus. We believe this is an excellent decision, as the properties of a functional approach make it ideal for this situation.

The syntax of ProScript defined in the paper is presented here. (Kobeissi et al., 2017)

$\langle \nu \rangle$ ::== (values)
- x (variables)
- n (numbers)
- s (strings)
- true, false (booleans)
- undefined, null (predefined constants)

$\langle e \rangle$ ::== (expressions)
- ν (values)
- $\{x_1 : \nu_1, ..., x_n : \nu_n\}$ (object literals)
- $\nu.x$ (field access)
- $[\nu_1, ..., \nu_n]$ (array literals)
- $\nu[n]$ (array access)
- Lib.l$(\nu_1, ..., \nu_n)$ (library call)
- f$(\nu_1, ..., \nu_n)$ (function call)

$\langle \sigma \rangle$ ::== (statements)
- var $x; \sigma$ (variable declaration)
- $x = e; \sigma$ (variable assignment)
- const $x = e; \sigma$ (constant declaration)

| if (ν_1 === ν_2) $\{\sigma_1\}$ else $\{\sigma_2\}$ (if-then-else)
| return e (return)

$\langle\gamma\rangle$ ::== (globals)
| const $x = e$ (constants)
| const f = function$(x_1, ..., x_n)$ $\{\sigma\}$ (functions)
| const $Type\_x = \{...\}$ (user types)

$\langle\mu\rangle$ ::== $\gamma_0; ...; \gamma_n$ (modules)

The operational semantics of ProScript is the same as that as JavaScript, and it does in fact run on unmodified JavaScript interpreters, since it is strictly a subset of the language. The tables included detailing this are included in the paper (Kobeissi et al., 2017), but aren't especially relevant here. To show the simplicity of the syntax, an example drawn from the paper's implementation of the Signal Protocol is included in figure one (Kobeissi et al., 2017).

```
  recv: function(me, them, msg) {
    var me = Type_me.assert(me)
    var them = Type_them.assert(them)
    var msg = Type_msg.assert(msg)
    const themMsg = {them: them, msg: msg}
    if ((msg.status === 2) && (them.status === 0)) {
      return recvHandlers.AKEResponse(me, them, msg)
    }
    else if ((msg.status === 3) && (them.status === 2)) {
      return recvHandlers.message(recvHandlers.completeAKE
           (me, them, msg))
    }
    else if ((msg.status === 3) && (them.status === 3)) {
      return recvHandlers.message(themMsg)
    }
    else { return {
       them: Type_them.construct(), output: Type_msg.
            construct(), plaintext: ''
    }}
  }
}
```

Figure 1: ProScript: Signal Receive Function Example

In figure one, we see the recv function, which is one of the object literals defined in a global, laid out just as in the syntax table, as it has a list of variables (the input to the function), and a body consisting of statements (σ).

2.4 Denning Sacco Snippets in ProScript

As an example, lets look at a small portion of Denning Sacco written in the syntax of ProScript.

```
const Type_keypair = {
    //this describes what a keypair object looks like.
    construct: function() {
        return {
            priv: Type_key.construct(),
            pub: Type_key.construct()
        };
    },

    //we assert that the private and public parts of the key are in fact of
    //Type_key, or the key type.
    assert: function(a) {
        a.priv = Type_key.assert(a.priv);
        a.pub = Type_key.assert(a.pub);
        return a;
    },

    //This describes how to clone a, where a is a keypair.
    clone: function(a) {
        var b = Type_keypair.construct();
        b.priv = Type_key.clone(a.priv);
        b.pub = Type_key.clone(a.pub);
        return b;
    }
};
```

This portion sets up a global type used in the protocol, that of a keypair, which has both public and private parts of a key. Inside this type, we see that there are three object literals whose value is given by the return value of a function.

2.5 Development of ProScript: Compiler to ProVerif Model

The key component of ProScript is in fact the compiler which provides the automatic translation into the Pi Calculus. In this section, we explore that.

It is important to note that the soundness of this translation is not proven, though the researchers do cite a paper as evidence for a plausible approach to follow in proving it. The rules for translation given in the paper (Kobeissi et al., 2017) are presented here:

$M_\nu ::== \nu | \{x_1 : \nu_1, ..., x_n : \nu_n\} — [\nu_1, ..., \nu_n]$

(Values to terms)

$\mathcal{V}[\![M_\nu]\!] \to M$

$\mathcal{V}[\![\nu]\!] = \nu$
$\mathcal{V}[\![\{x_1 : \nu_1, ..., x_n : \nu_n\}]\!] = Obj_t(\nu_1, ..., \nu_n)$
$\mathcal{V}[\![[\nu_1, ..., \nu_n]]\!] = Arr_t(\nu_1, ..., \nu_n)$
(Expressions to terms)
$\mathcal{E}[\![e]\!] \to m$
$\mathcal{E}[\![M_\nu]\!] = \mathcal{V}[\![M_\nu]\!]$
$\mathcal{E}[\![\nu.x]\!] = get\_x(\nu)$
$\mathcal{E}[\![\nu[i]]\!] = get\_i(\nu)$
$\mathcal{E}[\![Lib.l(\nu_1, ..., \nu_n)]\!] = Lib\_l(\mathcal{V}[\![\nu_1]\!], ..., \mathcal{V}[\![\nu_n]\!])$
$\mathcal{E}[\![f(\nu_1, ..., \nu_n)]\!] = f(\mathcal{V}[\![\nu_1]\!], ..., \mathcal{V}[\![\nu_n]\!])$
(Statements to enriched terms)
$\mathcal{S}[\![\sigma]\!] \to E$
$\mathcal{S}[\![\text{var } x; \sigma]\!] = \mathcal{S}[\![\sigma]\!]$
$\mathcal{S}[\![x = e; \sigma]\!] = \text{let } x = \mathcal{E}[\![e]\!] \text{ in } \mathcal{S}[\![\sigma]\!]$
$\mathcal{S}[\![\text{ const } x = e; \sigma]\!] = \text{let } x = \mathcal{E}[\![e]\!] \text{ in } \mathcal{S}[\![\sigma]\!]$
$\mathcal{S}[\![\text{return } \nu]\!] = \mathcal{V}[\![\nu]\!]$
$\mathcal{S}[\![\text{ if } (\nu_1 === \nu_2)\{\sigma_1\} \text{ else } \{\sigma_2\}]\!] = \text{if } \mathcal{V}[\![\nu_1]\!] = \mathcal{V}[\![\nu_2]\!] \text{ then } \mathcal{S}[\![\sigma_1]\!] \text{ else } \mathcal{S}[\![\sigma_2]\!]$
$\mathcal{F}[\![\gamma]\!] \to \Delta$ (Types and functions to Declarations)
$\mathcal{F}[\![\text{ const } f = \text{function}(x_1, ..., x_n)\{\sigma\}]\!] = \text{letfun } f(x_1, ..., x_\nu) = \mathcal{S}[\![\sigma]\!]$
$\mathcal{F}[\![\text{ const } Type\_t = \{...\}]\!] = \text{type t}$
(Constants to top-level-process)
$\mathcal{C}[\![\mu]\!](P) \to P$
$\mathcal{C}[\![\epsilon]\!](P) = P$
$\mathcal{C}[\![\text{ const } x = e; \mu]\!] = \text{let } x = \mathcal{E}[\![e]\!] \text{ in } \mathcal{C}[\![\mu]\!](P)$
(Modules to scripts)
$\mathcal{M}[\![\mu]\!](P) \to \Sigma$
$\mathcal{M}[\![\mu]\!](P) = \mathcal{F}[\![\gamma_1]\!], ..., \mathcal{F}[\![\gamma_n]\!].\mathcal{C}[\![\mu_c]\!](P)$

'Where μ_c contains all the globals of the form const $x = e$ in μ, and $\gamma_1, ..., \gamma_n$ consist of all other globals of μ.'

For example, if we wanted to translate a portion of the Denning Sacco keypair definition above, we would do it like this.

First, the type declaration of 'const Type_keypair= {...}' would cause the translator to write a type definition in the ProVerif version of the Pi Calculus, of the form 'type keypair.' As the reader can see, much of the translation is fairly direct.

The inside of the type declaration is actually not used in the translation.

Instead, if we wanted to translate a function, such as:

```
const fakeSend = function(mySigningKeys, theirIdentityKeys, k){
    var mySigningKeys= Type_keypair.assert(mySigningKeys)
    var theirSigningKeys = Type_keypair.assert(theirSigningKeys)
    var k = Type_key(k)

    return [mySigningKeys, theirIdentityKeys, k]
 }
```

Send would be translated similarly to:

letfun send(mySigningKeys, theirIdentityKeys, k) =
let mySigningKeys =
get_mySigningKeys(Type_keypair) in let theirSigningKeys =
get_theirSigningKeys(Type_keypair) in let k =
Type_key(l) in Arr_t(mySigningKeys, theirIdentityKeys, k)

Note: The get functions are just approximations here – In the paper they state that they are automatically generated by the compiler, but the process by which they are automatically generated isn't immediately clear. The main point of the above example, however, was to show that the ProVerif input is rather verbose. In our opinion, it seems much more intuitive to write in the ProScript syntax, showing the value of this compiler.

3 ProVerif

ProVerif is an automated model checker for cryptographic protocols that was developed by Bruno Blanchet. It has seen considerable use within the academic community since its creation in 2001. We present the underlying mechanisms by which the tool works in this section.

Traditionally, cryptographic protocols are verified manually by treating cryptographic primitives as functions between bitstrings, and adversaries as being arbitrary polynomial-time deterministic or probabilistic algorithms. In this model, called the "computational model", a protocol is considered to be broken if a poly-time adversary can violate a security property of the protocol under some assumption of time and computational power. Until recently, most of these security proofs had to be done manually. However, an alternative way to test the security of a protocol is through the use of the "symbolic model", more commonly known as the "Dolev-Yao" model. The model works by making several simplifying assumptions. First, all cryptographic primitives are assumed to be black boxes. The second assumption is that an attacker, Mallory, is given complete control of the network. Mallory is able to intercept all messages, use any publicly known function, and send any message they want to any principal.

Using the Dolev-Yao model as a framework, ProVerif works by translating a model in a variant of the Applied Pi Calculus, to a set of horn clauses. These horn clauses are queried according to some pre-defined security properties. If no derivation exists for the queries, the properties are found to be true. If they can be found, then the property is false and thus the protocol is insecure. However, it should be noted that Proverif is complete only for secrecy and authentication security properties. For all other properties it is incomplete i.e. there are some properties it is not able to provide a derivation for. If it is not able to prove the derivation, it outputs the fact that it doesn't know.

(Blanchet, 2010)

3.1 Applied Pi-Calculus: Syntax and Informal Semantics

Central to a protocol defined under the Dolev-Yao model are the black-box cryptographic primitives that the protocol uses. In the Applied Pi-Calculus, these primitives are given by a finite set of functions.

M, N ::==
x, y, z - variables
a, b, c - names
$f(N_1, ...N_k)$ - function application

Terms are either variables, or names which represent data, or the application of a function on other terms. One type of data referred to by a name is a "channel", which are exactly as they describe - a route for communication. Any processes with access to the same channel name are able to communicate with one another. Functions that can be used to construct new terms from other terms are called "constructors". For example, a hash function might be represented as a unary function $hash(x)$. The hash of a message m would be defined as $hash(m)$. Functions that manipulate terms in order to produce new terms are called "destructors". Manipulations are done according to a set of rewrite rules, and allow us to express the behaviour of cryptographic primitives. For example, if we have the constructor $enc(m, k)$ representing a ciphertext, the destructor $dec(enc(m, k), k) \rightarrow m$ shows how decryption works. In summary, terms are best thought of as data that the protocol uses or passes around.

D ::==
M, N - terms
$f(D_1, ...D_k)$ - function application
$fail$

Expressions are a super set of terms that contain destructor application of expressions, and a special failure expression denoted "fail". The

P, Q ::==
0 - nil process
$!P$ - replication
$P|Q$ - parallel composition
$in(N, x : T); P$ - input
$out(N, M); P$ - output
$newa : T; P$ - name restriction
$letx = DinP; elseQ$ - evaluation
$IfMThenPElseQ$ - Conditional

(Blanchet, 2016) The formalism of processes is precisely what makes the Applied Pi-Calculus so suited to modelling protocols. At its core, a cryptographic

protocol is simply a means of establishing of a channel of communication between parties. The in process and out process model that exactly. in takes an incoming message on channel N and binds it to the variable x. out sends a message M on channel N. The nil process does nothing. Parallel composition runs processes P and Q in parallel. Replication runs an infinite number of processes P in parallel. Replication is precisely what allows the Pi-Calculus to model an unbounded number of sessions. restriction creates a new name of type T. expression evaluation evaluates statement D and binds it to variable of Type T if D doesn't evaluate to fail, and then runs P. If D does evaluate to fail, or returns an expression not of type T, then Q is run.

Unlike the original Pi-Calculus, the Applied Pi-Calculus variant used by ProVerif has an extensive type system.It supports user defined types.

The ProScript translation also makes use of several syntactic constructs that exist in the variant of the Applied Pi-Calculus that ProVerif uses as its input language. Namely, tables, events, and phases.

Tables simply store tuples of terms. These terms represent the state at a current point in the execution of a process.

An event can be used to register that some step occurred, but has no noticeable effect on the protocol. They are useful for proving authentication, as we will explain in section 3.4.

Phases allow the grouping of different processes in a protocol. For instance, phase 0 may be a key exchange where both parties share ephemeral keys. Once keys are exchanged, it could then proceed to phase 1, where all the processes deal with the communication of nonces or encrypted messages.

3.2 Applied Pi-Calculus: Semantics

The semantics by which one uses the Applied-Pi Calculus can be expressed either using structural congruences and a reduction relation, or by a semantic configuration and a reduction relation. Bruno Blanchet recommends that the latter choice offers a more accurate representation, so that is what we show here. (Blanchet, 2014)

A semantic configuration is a pair consisting of E and $\mathcal{P}$. E, called the Environment, is a pair of finite sets of names: $\mathcal{N}_{public}$ and $\mathcal{N}_{private}$. An Adversary is then thought of as any process that solely has acess to the set N_public. The set $\mathcal{P}$ is the set of all processes that have no free variables, also known as closed processes. Informally, these can be thought of as the processes that are currently "running". As will be seen below, most reduction rules take the form of adding one or several processes to the set $\mathcal{P}$. This can be thought of as "executing" a process. A configuration is valid if two conditions are met. The first, is that the sets $\mathcal{N}_{private}$ and $\mathcal{N}_{public}$ are disjoint. The second is that the set of free names in the the process P $\subseteq \mathcal{N}_{private} \cup \mathcal{N}_{public}$.

We now present the semantics of ProVerif's Applied Pi-Calculus. (Blanchet, 2014) Nil Reduction: $E, \mathcal{P} \cup \{0\} \rightarrow E, \mathcal{P}$

Parallel Composition Reduction: $E, \mathcal{P} \cup \{P|Q\} \rightarrow E, \mathcal{P} \cup \{P, Q\}$

Replication Reduction: $E, \mathcal{P} \cup \{!P\} \rightarrow E, \mathcal{P} \cup \{P, !P\}$

Restriction Reduction: $(\mathcal{N}_{private}, \mathcal{N}_{public}), \mathcal{P} \cup \{\text{new a;} P\} \rightarrow (\mathcal{N}_{private}, \mathcal{N}_{public} \cup \{a'\}), \mathcal{P} \cup \{P\{a'/a\}\}$ where a' $\notin \mathcal{N}_{private} \cup \mathcal{N}_{public}$

Input/Output Reduction: $E, \mathcal{P} \cup \{out(N, M); Q, in(N, x); P\} \rightarrow E, \mathcal{P} \cup \{Q, P\{M/x\}\}$

Evaluation Reduction Rules:

$E, \mathcal{P} \cup \{\text{let } x = D \text{ in P else Q}\} \rightarrow E, \mathcal{P} \cup \{P\{M/x\}\}$ if D $\downarrow$ M

$E, \mathcal{P} \cup \{\text{let } x = D \text{ in P else Q}\} \rightarrow E, \mathcal{P} \cup \{Q\}$ if D $\downarrow$ fail

Conditional Reduction Rules:

$E, \mathcal{P} \cup \{\text{if true then P else Q}\} \rightarrow E, \mathcal{P} \cup \{P\}$

$E, \mathcal{P} \cup \{\text{if M then P else Q}\} \rightarrow E, \mathcal{P} \cup \{Q\}$ if M $\neq$ true

(Blanchet, 2016)

Each reduction rule for each operator is expressed as the adding of processes to the set $\mathcal{P}$ of closed processes. Intuitively, this can be thought of as a protocol making progress. The nil process doesn't do anything, so no process is added. Contrast that with the Parallel composition operator or replication. The former adds processes P and Q to $\mathcal{P}$, and the latter uses a recursive application of the rule to add an infinite number of process P to $\mathcal{P}$. Because there are an unbounded number of processes P that are being created, this construct is used by ProVerif to model a protocol running over an unbounded number of sessions.

The restriction reduction rule introduces a new name a, by first introducing an element a', which does not currently exist in the set of free names, into the set of private names. $P\{a'/a\}$ means to replace every a in P with a'. Therefore, if there is a process P a that requires the use of name a, and a hasn't been instantiated yet, we replace every occurrence of it in P with an instantiated version a'.

The input/output rule is what allows the Applied Pi-Calculus to express communication. In the context of this rule, the reduction works as follows: A message M is output on channel N before proceeding to process Q. The output process is completed, and so is not added to P. The in process then accepts a term M on channel N, and binds x to M before proceeding to process P with the bound x. P is added to $\mathcal{P}$. Informally, this says that if an input and output process have access to the same channel name, they can communicate and pass terms.

The $\downarrow$ operator in the context $D \downarrow M$ means the expression D evaluates to M. The two evaluation reduction rules demonstrate how the evaluation of D can succeed, leading to the binding of x to M and the running of process P with M substituted for x. Or, the expression D can fail, leading to process Q being executed.

The conditional reduction rule operates similarly to the evaluation one. If the Boolean value in the conditional is the constant value true, then the process P is added to the set of closed processes. Otherwise, process Q is added.

3.3 Applied Pi-Calculus Example

Here we present an example of the Denning-Sacco protocol implemented in the Applied Pi-Calculus. (Blanchet, Smyth, & Cheval, 2016)

First, we define a finite set of constructor and destructor definitions.
functions: {
pk(x:skey),
aenc(x,k:skey),
sign(m,k:skey),
check(sign(m,k:skey),pk(k:skey)) → true,
sdec(senc(m,k:key),k:key) → m,
adec(aenc(m,k:skey),k:skey) → m,
}

Next, we define an initialization process P_0 that creates new secret signing and encryption keys, before broadcasting them along public channel c. Any adversary for this model has access to the public channel c, the functions defined above, as well as any messages broadcast on it. P_A and P_B are then started with their respective secret keys, and each others public keys as part of their environments.

P_0 =new ssk_A: skey; new sk_B:skey; let $spk_A = pk(ssk_A)$ in let $pk_B = pk(sk_B)$ in $out(c, spk_A); out(c, pk_B)$; $(P_A(ssk_A, pk_B)|P_B(sk_B, spk_A))$

Process A generates a new key k, and outputs the signature of its encryption to B, as in the protocol. It then awaits a message from B that it can decrypt with the now shared key k.

Process A $P_A(ssk_A, pk_B) =$! (new k: key; $out(c, aenc(sign(k, ssk_A), pk_B))$; $in(c, x : bitstring)$; let z =$sdec(x, k)$ in 0)

As in the Denning-Sacco protocol, B accepts the signature of the encryption of the key that it decrypts with its private key. It then sends a message s back across public channel c under the encryption of the shared key k. Like P_A, P_B is run under replication to represent an unbounded number of sessions.

$P_B(sk_B, spk_A)$ =!$(in(c, y : bitstring); Lety_0 = sdec(y, sk_B)$ in let$xk = check(y0, spk_A)$ in $out(c, senc(s, xk)))$

The same protocol implemented in ProVerif looks slightly different, but otherwise the logic is almost completely identical.

As an example, we present it here. The key difference is just how syntactic constructs are defined.

P_0 =new ssk_A: skey; new sk_B:skey; let $spk_A = pk(ssk_A)$ in let $pk_B = pk(sk_B)$ in $out(c, spk_A); out(c, pk_B)$; $(P_A(ssk_A, pk_B)|P_B(sk_B, spk_A))$

```
(*defining name c as a channel*)
free c: channel

type key

type skey

type pkey

type sskey
type spkey


(*constructor definitions*)
fun pk(skey):pkey
fun aenc(bitstring,pkey): bitstring
fun senc(bitstring,key): bitstring
fun sign(bitstring,sskey): bitstring

(*destructor definitions*)
reduc forall m: bitstring, k:skey; adec(aenc(m,pk(k)),k) = m
reduc forall m: bitstring, k:key; sdec(senc(m,k),k) = m
reduc forall m: bitstring, k:sskey; check(sign(m,k),spk(k)) = m

(*Defining Process A*)
let A(sskA:sskey, pkB:pkey) = !(new k:key;
    out(c,aenc(sign(k,sskA));in(c,x:bitstring);
let z = sdec(x,k);0).


(*Defining Process B*)
let B(skB:skey,spkA:spkey) = !(in(c,y:bitstring);
   let y0 = sdec(y,skB) in
   let xk=check(y0,spkA) in
   out(c,senc(s,xk))).


(*Defining Process_0, which is done with the main process*)
process
   new sskA: skey;
   new skB:skey;
   let spkA = pk(sskA)
   let pkB = pk(skB)

   (A(sskA,pkB)) | (B(skA,spkA))
```

3.4 Defining Security Properties

Protocols are often designed with a specific goal in mind. When that goal is cryptographic security, that goal is achieved by guaranteeing certain properties hold. The foremost of these is secrecy. An adversary should be unable to read or accurately interpret the messages the principals are sending one another. Another key property is authentication. A principal should have confidence that they are communicating with the party they want to, and not an impostor. A third property, often desired for its theoretical value is indistinguishably. An adversary should be unable to distinguish messages based on the ciphertexts they observe. Indistinguishability is best described intuitively as preventing the adversary from gaining any "leaked" information about the secret keys underlying the crypto-scheme. Any adversary looking at ciphertexts, even when they know the original messages that were encrypted to produce those ciphertexts, should be unable to determine which message resulted in which encryption.

Under the framework of the Applied Pi-Calculus, where we can express data, ways to manipulate it, and the means of sending it between processes, these properties can be quite intuitive to model. For example, consider a name k representing a key. If a protocol was to be designed where the designer wanted to ensure the security of the key k, they would put the following statement in the declaration section of their ProVerif model.

not query attacker(k)

The function attacker(k) means that the adversary has access to k, and is able to output it on a public channel. If the horn-clause resolution described in the next section is able to complete the protocol such that the attacker does not have access to k, then k is guaranteed to be secret.

Authentication is checked through the use of events and correspondence assertions. Correspondence assertions are claims that if an event e was executed, then the event e' was executed prior. ProVerif also supports injective-correspondence assertions, which requires that if event e was executed, then only one instance of event e executed. In ProVerif, we check for correspondences in a very intuitive way:

(* regular correspondence assertion*)

query event(e($M_1, ...M_j$))==>event(e'($N_1, ...N_k$))

(*injective correspondence assertion*)

query inj-event(e($M_1, ...M_j$))==>inj-event(e'($N_1, ...N_k$))

(Blanchet et al., 2016)

Authentication is checked for by declaring events and correspondence assertions. If the protocol is able to be completed without the assertions holding true, as found by horn clause resolution, then the protocol does not guarantee authentication.

Indistinguishability is a harder notion to verify. In the context of the Pi Calculus, indistinguishability is guaranteed if the property of Observational Equivalence holds. We won't discuss Observational Equivalence much, except to say that it is undecidable in general. (Blanchet, 2016)As a result, ProVerif proves diff-equivalence, which stronger than observaional-equivalence. Consequently ProVerif is incomplete with regards to equivalence properties, and it derives attacks that are "false" i.e. attacks that wouldn't exist if observational equivalence could be proved.

Diff-equivalence, which is defined as an equivalence between two processes that differ only by their terms but otherwise share the complete syntactic structure is what ProVerif actually provides derivations for. It's decideable for all syntactic structures defined earlier, except for those with else branches. (Cheval & Blanchet, 2013)

As mentioned before, previous attempts to integrate ProVerif into the development of a cryptographic protocol require a high amount of developer overhead. Changes to the protocol might require a redefining of the security properties to be derived. It is also worth noting that ProVerif can only derive one class of security properties at a time, which necessitates the manual creation of separate files. With every update to the protocol, each one of these files would need manual updating. By comparison, a change in the implementation of the protocol, a user of ProScript would only need to specify which properties they want to prove and the ProScript to ProVerif compiler takes care of the rest.

3.5 Translation

The translation can be thought of as a function, which takes as input two distinct objects defined in the process calculus. The first is the protocol which is to be checked, and the second consists of the definitions of the security properties that the protocol purportedly preserves. The translator takes these, and outputs three sets of rules in the form of Horn clauses, which describe the 'rules' for the attacker (What they can compute, how they can encrypt messages), the facts for attacker (Whose names and public keys they know, the fact that they have a name), and the 'rules' for the protocol (How the sending of each message works). The rules for the attacker describe its abilities as laid out in the Dolev-Yao model. For the protocol, the rules are made up of the translation of the processes. As stated in Abadi and Blanchet, 2005, process translation can be thought of as follows: 'Thus, the translation of a process is, very roughly, a set of rules that enable us to prove that it sends certain messages.' (Abadi & Blanchet, 2005)

These constructed rules use two predicates, attacker(p) and message(p,p'). attacker(p) means that the attacker may have access to p, and message(p,p') means that the message p' may appear on channel p.

Each of these p is a pattern, which is defined as follows:

p ::== (pattern)
x, y, z (variable)

$a[p_1, ..., p_n]$ (name)
$f(p_1, ..., p_n)$ (constructor application)

(Abadi & Blanchet, 2005)

For example, lets see how the translation of one of the messages in the Denning-Sacco protocol would look. The rules for translating specific processes will be described as they are used. We'll examine the last part of P_0, defined above in the Pi Calculus.

First, the parallel composition of P_A and P_B is run. This is translated as the union of the translation of P_A and P_B, since we want to represent any messages sent by P_A and P_B, as well as messages that result from them interacting.

Then, P_A is defined using replication, which is the ! symbol. This means that it can represent an unbounded number of sessions. However, since this is being translated into classical logic, the replication is ignored, because any rule can be applied as many times as needed, so the replication is already inherent in the replication.

So how would part of the inside of P_A be translated? For example, we see that a name is generated (k), of type key. This is a restriction, and is translated with the pattern k[...] where the ... is generated depending on the inputs received before the generation of the name. In this case, k is named before anything has happened in the process, so it is translated as k[].

To see what some of the clauses used would look like, let us try writing out the initial knowledge of the attacker. One piece of information the attacker would have at the start is a name. This would be expressed by:

attacker(a[])

Another piece of information the attacker would know is the public key of the other members of the network. So this could be expressed by:

attacker(pk(sk_A[]))

3.6 Horn Clauses

Once the model has been translated into a set of Horn Clauses, ProVerif is ready to start creating proofs. It does this through a variant of the classic approach to proof-generation in logic-based programming, Resolution.

Horn Clauses are defined as clauses which contain at most one positive literal. This divides them into two classes, definite clauses and goal clauses. A definite clause has exactly one positive literal, and a goal clause has no positive literals.

Resolution is an inference rule which takes two clauses, and produces a new clause which contains all the literals in each of the input clauses, but leaves out any complementary literals in the clauses. For example:

attacker(a) $\vee$ attacker(b), $\neg$ attacker(a) $\vee$ attacker(b)

Produces:

attacker(b)

However, ProVerif uses a special variant of resolution, called resolution with Free Selection. This is an algorithm written by Blanchet, and consists of resolution guided by a Selection algorithm, which decides which clauses to resolve

first. Importantly, resolution with Free Selection is used because it has been shown, by Blanchet and Podelski in 2003, that it always terminates on a special class of protocols called tagged protocols. The basic idea of the Free Selection algorithm used in ProVerif is that it avoids selecting facts of the form attacker(x) for resolution, because they can cause the process not to terminate. (Blanchet, 2011)

Tagged protocols are protocols in which every appearance of a cryptographic primitive has a unique name.

Horn Clauses are used in proof-generation contexts because the resolvent of a goal clause and a definite clause is always a goal clause, since the process of resolution removes the single positive literal from the clause. This is key for ProVerif, because a resolution resulting in the trivial goal clause (the empty clause) is the main goal of the resolution algorithm. The empty clause is the goal because the algorithm has established security properties, and had formulated the protocol inside this logical format, so if the conjunction of the security properties with the protocol leads to a contradiction (empty clause), the security property must be false for the protocol. The proof by resolution of this contradiction can be easily represented as a trace, since it expresses exactly what the attacker may have access to at several points, and what clauses justify that access. This trace then creates the basis for the user to modify and fix their protocol.

4 Analysis

Using their ProScript to ProVerif compiler, the researchers were able to verify actual production code in the CryptoCat WebApp. Not only that, they used their tool to verify the Telegram and Signal Protocols as well. (Kobeissi et al., 2017) This is an impressive result, and is a clear demonstration of ProScript's potential usefulness. However, we were unable to achieve the same results even when run on the same code. The readme on the Github repository where the ProScript to ProVerif compiler is hosted makes it abundantly clear that it is experimental software. Because it is still in development, we feel that our inability to get the code running on our machines should not be counted against the project or its aims in any way. The code was not available publicly, and it was provided to us at our request.

As for ProVerif, it is certainly a very powerful tool, giving detailed traces of attacks with extremely fast speeds, but it requires quite a lot of knowledge to use. For example, not only does one need to know the Pi Calculus, but also the unique syntax of ProVerif's version of the Pi Calculus. Each time the implementation is changed, the model needs to be rewritten, and it must be changed for every security property that the user wants to check. Due to all this, the appeal of a more accessible front-end like ProScript is clear. Also, because ProVerif is not complete, there may be some attacks that exist on the protocol that cannot be proved.

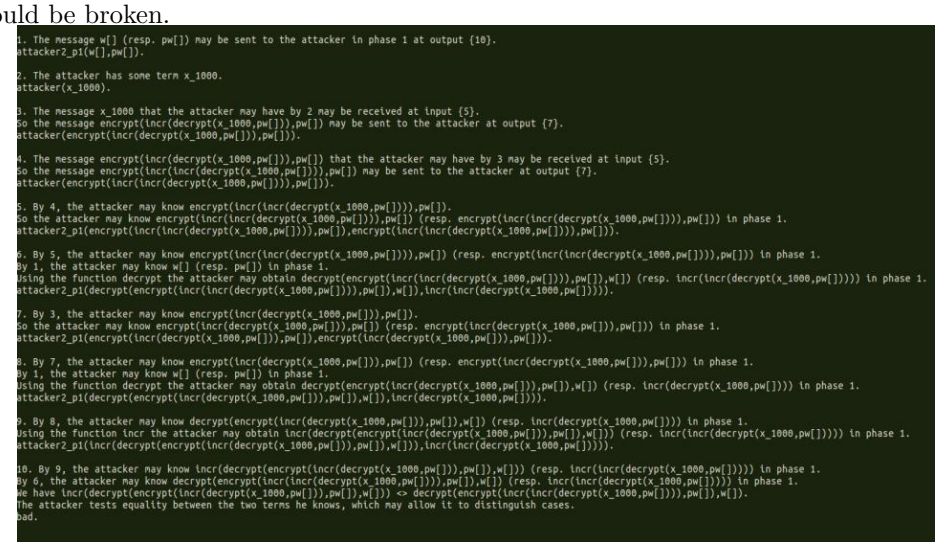
The greatest weakness of ProScript is that the researchers have yet to prove

the soundness of the translations their compiler outputs. In other words, there is no proof that the results of ProVerif verifications done on the output of the compiler actually imply anything about the ProScript implementation. The researchers do address this, and they plan to follow a proof of a similar type, laid out in a paper by K. Bhargavan called 'Verified interoperable implementations of security protocols'.

Ultimately, we find that ProScript, and the novel development methodology of which it is a part are excellent ideas. The fact that the researchers were able to use it on production code demonstrates the potential which automation has for changing the landscape of security, especially on the Internet where so much JavaScript, bad cryptography, and their fell offspring reside.

5 Appendix: Example Trace

In the following image, we show the trace received from a simple protocol being verified with ProVerif, demonstrating to the user the way in which the protocol could be broken.



References

Abadi, M., & Blanchet, B. (2005). Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, *52*.

Blanchet, B. (2001). An efficient cryptographic protocol verifier based on prolog rules. *IEEE Computer Security Foundations Workshop (CSFW-14)*.

Blanchet, B. (2010). *The automatic security protocol verifier proverif.* Retrieved from `http://prosecco.gforge.inria.fr/personal/bblanche/talks/Secret10.pdf` (SecRet Workshop)

Blanchet, B. (2011). *Formal models and techniques for analyzing security protocols*. IOS Press, Amsterdam.

Blanchet, B. (2014). Automatic verification of security protocols in the symbolic model: the verifier proverif. *Lecture Notes in Computer Science*. Retrieved from http://prosecco.gforge.inria.fr/personal/bblanche/publications/BlanchetFOSAD14.pdf

Blanchet, B. (2016, October). Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, *1*(1–2), 1–135. Retrieved from http://dx.doi.org/10.1561/3300000004

Blanchet, B., Smyth, B., & Cheval, V. (2016). Proverif 1.96: Automatic cryptographic protocol verifier, user manual and tutorial [Computer software manual]. (Originally appeared as Bruno Blanchet and Ben Smyth (2011) ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial.)

Cheval, V., & Blanchet, B. (2013, March). Proving more observational equivalences with ProVerif. In D. Basin & J. Mitchell (Eds.), *2nd conference on principles of security and trust (post 2013)* (Vol. 7796, pp. 226–246). Rome, Italy: Springer Verlag. Retrieved from http://prosecco.gforge.inria.fr/personal/bblanche/publications/ChevalBlanchetPOST13.htm

Kobeissi, N., Bhargavan, K., & Blanchet, B. (2017). *Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach.* (Unpublished paper)