Dynamic Model Checking of C Cryptographic Protocol Implementations

Alan Jeffrey[⋆1] and Ruy Ley-Wild[⋆⋆2]

[1] Bell Labs, Lucent Technologies
[2] Computer Science Department, Carnegie Mellon University

Abstract. We describe the Dolev–Yao C (DYC) cryptographic protocol message API. In addition to generating executable protocol implementations, DYC can be used to generate constraints on an attacker against the protocol. The resulting constraints can be handed to a constraint solver, which (if successful) will find an executable attack against the protocol. To our knowledge, this is the first attempt to automate the discovery of flaws with executable cryptographic protocol implementations, rather than high-level protocol specifications or simulations.

1 Introduction

This paper is a first step in merging two separate fields: *model checking cryptographic protocols* and *dynamic software model checking*.

Model checking [8] is a well-established field, based on efficient state-space exploration techniques. Lowe [22, 23], Marrero, Clarke and Jha [24, 9], Denker, Millen and Ruess [10] and Blanchet [6], among others, have shown that model-checking techniques can be used to find flaws in cryptographically-secured communications protocols specified in high-level domain-specific languages such as CSP [19] or the spi-calculus [2].

Software model checking is a more recent field, in which the state-space exploration techniques of model checking are applied to software artefacts directly, rather than on domain-specific languages. Godefroid and Klarlund [14] characterize software model checkers as *dynamic* (which work by generating constraints from directed runs of instrumented executables) or *static* (which perform static analysis of source code). Dynamic tools include Godefroid's VeriSoft [12], Havelund and Pressburger's Java PathFinder [17], Musuvathi *et al.*'s CMC [26], Robby, Dwyer and Hatcliff's Bogor [29] and Godefroid, Klarlund and Sen's DART [15]. Static tools include Ball and Rajamani's SLAM [3, 4], Henzinger *et al.*'s BLAST [18] and Clarke, Kroening and Lerda's CBMC [7].

In this paper, we provide a first step at using dynamic software model-checking techniques to verify executable implementations of cryptographic protocols. The closest prior works to this are:

⋆ This material is partly based upon work supported by the National Science Foundation under Grant No. 0208549.
⋆⋆ Supported by a Bell Labs Graduate Research Fellowship from Lucent Technologies.

– Godefroid *et al.*'s [13, 15] use of C model checkers to analyze the Needham–Schroeder public key authentication protocol [27]. This analysis, however, was performed on a simulation of the protocol which did not perform any cryptographic operations.
– Goubault-Larrecq and Parrennes' [16] use of static program analysis to analyze cryptographic protocols. Their work is orthogonal to ours, in that they perform shape analysis to extract constraints, rather than executing the program.
– Bhargavan *et al.*'s [5] use of Blanchet's [6] ProVerif model-checker to analyze F# cryptographic protocol implementations. Their work is based on a static tool `fs2pv` which extracts the ProVerif model, rather than model-checking the execution of the symbolic prototype.

In this work, we give a C API for implementing messages in a cryptographic protocol, with a natural abstract model based on the Dolev–Yao [11] model of cryptography. In addition to generating executables which link against the `libgcrypt` [21] cryptographic library, DYC supports a simple form of dynamic model checking: executions can generate constraints on an attacker. We use successive runs of the protocol to generate increasingly refined constraints, until we have generated constraints for an attack on the authenticity goals of the protocol. A solution to these constraints constitutes an attack which can be executed in C.

This exploration is only a first step, and has a number of limitations, notably the large amount of programmer involvement and the requirement that honest agents are straight-line code. We hope that the techniques discussed here can be integrated into a software verification tool which performs state-space exploration.

Acknowledgements. Thanks to Patrice Godefroid for insightful discussions.

2 Using DYC

Dolev–Yao C (DYC) [20] is a collection of C APIs which links against `libgcrypt` [21] for cryptographic functions and against XSB Prolog [28] for constraint satisfaction. The C APIs can be used without any model checking, in which case they just provide a wrapper around `libgcrypt`. Enabling model-checking will use constraint satisfaction to synthesize an attack on the protocol.

In the description of the API, we will ignore details such as initialization functions and memory de-allocation, which are relatively routine. We will also ignore some "sanity" checks performed by the implementation, such as ensuring that memory allocation succeeds.

2.1 Honest Agents API

The honest agents in a protocol are implemented in C, using the DYC honest agents API. This API provides type `data_t` for immutable byte arrays, together with functions including cryptography and assertions.

The `data_t` type, together with its associated non-cryptographic functions, is given in Figure 1. We require all uses of `data_t` to use the API, in particular this means that `data_t` objects are immutable, as there are no functions for performing in-place update.

```
typedef struct _data_t { size_t len; byte_t * val; } * data_t;

data_t data_from_string (char *str);
data_t data_concat (data_t data0, data_t data1);
data_t data_sub (data_t data, size_t off, size_t len);

void data_assert_nz (data_t data);
void data_assert_eq (data_t data0, data_t data1);
void data_assert_length (data_t data, size_t len);
```

Fig. 1. Data functions

```
data_t crypto_pcreate (void);
data_t crypto_pkey_public (data_t key);
data_t crypto_pkey_secret (data_t key);
data_t crypto_pencrypt (data_t key_p, data_t pdata);
data_t crypto_pdecrypt (data_t key_s, data_t cdata);

data_t crypto_nonce_create (void);
```

Fig. 2. Cryptographic functions

```
void principal_register (data_t name, data_t key_p);
data_t principal_lookup_key_p (data_t name);
```

Fig. 3. Principal functions

We provide assert functions for use by honest agents. These can be used to detect attempts by an attacker to subvert the protocol, for example `data_assert_eq` will often be used to ensure that a response to a nonce challenge is correct, and `data_assert_nz` (short for non-zero) will be used to ensure that an operation such as substring succeeded. Failure of one of these assert functions is considered not to violate the security goals of the protocol, as the honest agents have detected an unsuccessful attack.

The cryptographic functions for `data_t` are given in Figure 2. The current API only supports public key encryption and nonce generation: we expect that features such as symmetric encryption, signing and hashing could be added with little complication, but this is left for future work.

The cryptographic functions all return 0 in the case of failure such as an attempt to encrypt with a key of the wrong length, or an attempt to decrypt with the incorrect key. These functions link against libgcrypt, and so inherit its limitations: in particular, public key encryption is limited to ciphertexts of length 157. We expect that a different cryptographic library could be used, which would remove these limitations: our techniques do not depend on them.

Public-key cryptography often requires a binding between principals and their associated public keys. We provide a very simple API for registering principals, given in

Figure 3. The attacker is allowed to register their own public key, but is not allowed to register other principals, so cannot corrupt principal lookup.

Example: Needham–Schroeder In Figure 4, we provide an implementation of the Needham–Schroeder public key authentication protocol messages [27], together with Lowe's variant [22]. The original Needham–Schroeder protocol is given by:

$$A \to B : \{|N_A, A|\}_{\mathsf{enc}(K_B)}$$
$$B \to A : \{|N_A, N_B|\}_{\mathsf{enc}(K_A)}$$
$$A \to B : \{|N_B|\}_{\mathsf{enc}(K_B)}$$

and Lowe's variant adds B's name to the second message:

$$B \to A : \{|N_A, N_B, B|\}_{\mathsf{enc}(K_A)}$$

Each message mN is implemented in C using two functions: ns_msgN builds the Nth message, and ns_unmsgN deconstructs the message.

Note that in C style, results of functions are provided call-by-reference, and that functions which find sub-arrays are given using offset-and-length. Both of these features are exemplified by the assignment:

```
*Na = data_sub (pair, 0, NS_NONCE_NBYTES);
```

in ns_unmsg1.

2.2 Test Harness For Honest Agents

In a real protocol implementation, messages are communicated via network sockets. In future work, we hope to extend DYC to include a socket API, but for now we require the programmer to provide a test harness for a single run of the protocol. In this section, we consider the honest agents, and in Section 2.3 we extend this to include the attacker.

The honest agents in a single run are first initialized, then for each message in the protocol, we construct it (acting as the sending agent) then deconstruct it (acting as the receiving agent). The most interesting feature of the honest agent test harness is the specification of the goals of the protocol. There are a number of possible goals we could aim for, including confidentiality, integrity, authenticity, and non-repudiation. In this work, we will restrict attention to a simple case: validating one authenticity property per protocol.

We model authenticity properties by *correspondence assertions* [30]. Consider a scenario where honest agent A wishes to contact agent X, and honest agent B believes they have been contacted by agent Y. At the end of a protocol run, either Y is a dishonest agent, or $X = B$ and $Y = A$. This is modeled by having A issue a $\mathsf{begin}(A, X)$ statement, and B issue a $\mathsf{end}(Y, B)$ statement. In a safe run, these statements either match (as shown on the left of the following example), or the recipient has been contacted by a dishonest agent (as shown on the right):

$$\begin{array}{ll}
A \text{ issues } \mathsf{begin}(A,B) & \\
A \to B : \{|N_A, A|\}_{\mathsf{enc}(K_B)} & Z \to B : \{|N_Z, Z|\}_{\mathsf{enc}(K_B)} \\
B \to A : \{|N_A, N_B|\}_{\mathsf{enc}(K_A)} & B \to Z : \{|N_Z, N_B|\}_{\mathsf{enc}(K_Z)} \\
A \to B : \{|N_B|\}_{\mathsf{enc}(K_B)} & Z \to B : \{|N_B|\}_{\mathsf{enc}(K_B)} \\
B \text{ issues } \mathsf{end}(A,B) & B \text{ issues } \mathsf{end}(Z,B)
\end{array}$$

```
void ns_msg1 (data_t A, data_t Na, data_t Ka_p, data_t Kb_p, data_t *m1) {
  data_t pair = data_concat (Na, A);
  *m1 = crypto_pencrypt (Kb_p, pair);
}

void ns_unmsg1 (data_t *A, data_t *Na, data_t *Ka_p, data_t Kb_s, data_t m1) {
  data_t pair = crypto_pdecrypt (Kb_s, m1);
  data_assert_nz (pair);
  data_assert_length (pair, NS_NONCE_NBYTES + NS_NAME_NBYTES);
  *Na = data_sub (pair, 0, NS_NONCE_NBYTES);
  *A = data_sub (pair, NS_NONCE_NBYTES, NS_NAME_NBYTES);
  *Ka_p = principal_lookup_key_p (*A);
  data_assert_nz (*Ka_p);
}

void ns_msg2 (data_t B, data_t Na, data_t Nb, data_t Ka_p, data_t *m2) {
  data_t tuple = data_concat (Na, Nb);
  if (NS_LOWE) {
    tuple = data_concat (tuple, B);
  }
  *m2 = crypto_pencrypt (Ka_p, tuple);
}

void ns_unmsg2 (data_t B, data_t Na, data_t *Nb, data_t Ka_s, data_t m2) {
  data_t tuple = crypto_pdecrypt (Ka_s, m2);
  data_assert_nz (tuple);
  if (NS_LOWE) {
    data_assert_length
     (tuple, NS_NONCE_NBYTES + NS_NONCE_NBYTES + NS_NAME_NBYTES);
    data_t B_alleged = data_sub
      (tuple, NS_NONCE_NBYTES + NS_NONCE_NBYTES, NS_NAME_NBYTES);
    data_assert_eq (B, B_alleged);
  } else {
    data_assert_length (tuple, NS_NONCE_NBYTES + NS_NONCE_NBYTES);
  }
  data_t Na_alleged = data_sub (tuple, 0, NS_NONCE_NBYTES);
  data_assert_eq (Na, Na_alleged);
  *Nb = data_sub (tuple, NS_NONCE_NBYTES, NS_NONCE_NBYTES);
}

void ns_msg3 (data_t Nb, data_t Kb_p, data_t *m3) {
  *m3 = crypto_pencrypt (Kb_p, Nb);
}

void ns_unmsg3 (data_t Nb, data_t Kb_s, data_t m3) {
  data_t Nb_alleged = crypto_pdecrypt (Kb_s, m3);
  data_assert_nz (Nb_alleged);
  data_assert_eq (Nb, Nb_alleged);
}
```

Fig. 4. Messages in the Needham–Schroeder public-key protocol

```
void principal_register_honest (data_t name);
void principal_assert_honest (data_t name);

void ca_auth_begin (data_t send, data_t recv);
void ca_auth_end (data_t send, data_t recv);
```

Fig. 5. Correspondence assertion functions

In an unsafe run, the statements do not match, and the recipient has been fooled into believing they have been contacted by an honest agent, for example:

$$\begin{array}{ll} A \text{ issues } \mathsf{begin}(A,Z) & \\ A \to Z : \{|N_A,A|\}_{\mathsf{enc}(K_Z)} & Z \to B : \{|N_A,A|\}_{\mathsf{enc}(K_B)} \\ B \to Z : \{|N_A,N_B|\}_{\mathsf{enc}(K_A)} & Z \to A : \{|N_A,N_B|\}_{\mathsf{enc}(K_A)} \\ A \to Z : \{|N_B|\}_{\mathsf{enc}(K_Z)} & Z \to B : \{|N_B|\}_{\mathsf{enc}(K_B)} \\ & B \text{ issues } \mathsf{end}(A,B) \end{array}$$

The C API for this form of correspondence assertion is given in Figure 5. A typical use of these functions is:

```
principal_register_honest (A); principal_register_honest (B); ...
ca_auth_begin (A,X); ...
principal_assert_honest (Y); ca_auth_end (Y,B);
```

The goal of the attacker is to find an unsafe run of the protocol, that is one which reaches the `ca_auth_end (Y,B)` statement in a state where $X \neq B$ or $Y \neq A$.

Example: Test Harness For Needham–Schroeder A test harness for one run of the Needham–Schroeder honest agents is given in Figure 6. Note that since we are just executing the honest agents, A gets to choose the identity of X (in this case B) and that at the end of the run, Y will be bound to A, and so this run is safe.

2.3 Test Harness For The Attacker

We will now introduce the attacker into our model. In this work, we require the programmer to explicitly model the intervention of the attacker and to keep track of the attacker knowledge. In future work, we hope to require less programmer intervention.

The important feature of the attacker model is that the programmer is *not* required to construct the messages built by the attacker. Instead, these are synthesized by DYC using a constraint satisfaction engine.

The attacker API is very simple: it consists of one new datatype `derivation_t` (discussed in Section 3.2) and the function `derivation_build(d, k)` in Figure 7. This takes as arguments a derivation `d[]` and the attacker knowledge `k`, and returns a new message which can be built by the attacker.

A typical use of this function is as follows: we track the attacker knowledge `k`, and allow the attacker to mutate each message `mN` in transit:

```
void ns_run_once () {
  /* Honest agents */
  data_t A = data_from_string ("A"); data_t B = data_from_string ("B");
  data_t Ka = crypto_pcreate (); data_t Kb = crypto_pcreate ();
  data_t Ka_p = crypto_pkey_public (Ka); data_t Ka_s = crypto_pkey_secret (Ka);
  data_t Kb_p = crypto_pkey_public (Kb); data_t Kb_s = crypto_pkey_secret (Kb);
  principal_register (A, Ka_p); principal_register (B, Kb_p);
  principal_register_honest (A); principal_register_honest (B);

  data_t Na = crypto_nonce_create (); data_t Nb = crypto_nonce_create ();
  data_t X, Kx_p, Nx, Y, Ky_p, Ny, m1, m2, m3;

  /* A determines who X is */
  X = B;

  /* At A, talking to X */
  Kx_p = principal_lookup_key_p (X); data_assert_nz (Kx_p);
  ca_auth_begin (A, X); ns_msg1 (A, Na, Ka_p, Kx_p, &m1);
  /* At B, talking to Y */
  ns_unmsg1 (&Y, &Ny, &Ky_p, Kb_s, m1); ns_msg2 (B, Ny, Nb, Ky_p, &m2);
  /* At A, talking to X */
  ns_unmsg2 (X, Na, &Nx, Ka_s, m2); ns_msg3 (Nx, Kx_p, &m3);
  /* At B, talking to Y */
  ns_unmsg3 (Nb, Kb_s, m3); principal_assert_honest (Y); ca_auth_end (Y, B);
}
```

Fig. 6. Test harness for the Needham–Schroeder honest agents

```
data_t derivation_build (derivation_t d, data_t k);
```

Fig. 7. Attacker function

```
msgN(..., &mN);  // Sending honest agent builds message mN
k = data_concat (k, mN);  // Attacker adds mN to the attacker knowledge
mN = derivation_build (d[N], k);  // Attacker mutates message mN
unmsgN(..., mN);  // Receiving honest agent deconstructs message mN
```

The problem of finding an attack on a protocol comes down to finding an appropriate array of derivations d[].

Example: Attacking Needham–Schroeder A test harness for one run of the Needham–Schroeder protocol, including the attacker, is given in Figure 8. Note that the attacker gets to choose the identity of X, and gets to mutate each message in transit.

```
void ns_run_once (derivation_t *d) {
  ...
  /* Dishonest agent */
  data_t Z = data_from_string ("Z"); data_t Kz = crypto_pcreate ();
  data_t Kz_p = crypto_pkey_public (Kz); principal_register (Z, Kz_p);
  /* Initial attacker knowledge */
  data_t k = data_concat (Ka_p, data_concat (Kb_p, Kz));
  /* Attacker determines who X is */
  X = derivation_build (d[0], k);

  /* At A, talking to X */
  Kx_p = principal_lookup_key_p (X); data_assert_nz (Kx_p);
  ca_auth_begin (A, X); ns_msg1 (A, Na, Ka_p, Kx_p, &m1);
  /* Attacker mutates m1. */
  k = data_concat (k, m1); m1 = derivation_build (d[1], k);
  /* At B, talking to Y */
  ns_unmsg1 (&Y, &Ny, &Ky_p, Kb_s, m1); ns_msg2 (B, Ny, Nb, Ky_p, &m2);
  /* Attacker mutates m2. */
  k = data_concat (k, m2); m2 = derivation_build (d[2], k);
  /* At A, talking to X */
  ns_unmsg2 (X, Na, &Nx, Ka_s, m2); ns_msg3 (Nx, Kx_p, &m3);
  /* Attacker mutates m3. */
  k = data_concat (k, m3); m1 = derivation_build (d[3], k);
  /* At B, talking to Y */
  ns_unmsg3 (Nb, Kb_s, m3); principal_assert_honest (Y); ca_auth_end (Y, B);
}
```

Fig. 8. Test harness for Needham–Schroeder including the attacker

2.4 Finding An Attack

So far, we have seen how to execute one run of the protocol, including the attacker, given an array of message derivations d[]. To find an attack on the protocol, we repeatedly run the protocol, hoping to find an appropriate d[].

We build a collection of constraints which represent the invariants maintained by the honest agents. Any successful attack on the system must satisfy these constraints and violate the correspondence assertion. The initial constraints are trivial and the attacker knowledge is seeded with the public keys, thus the derivation array is initialized with garbage messages. The DYC functions implicitly accumulate constraints and the attacker knowledge is accumulated explicitly in the array k.

Most runs of the protocol will fail to find an attack, either because an assert fails, or because the run is safe. In either case, each run generates additional constraints, which we hand over to a constraint solver. If the constraint solver fails to find a solution, then we declare victory to the honest agents, otherwise we try again with the new value for d[] generated from the solution to the new constraints. A typical main loop for DYC is:

```
derivation_t d[N];
while (cassert_loop ()) { constraint_resolve (d, N); ... one run ... break; }
```

```
jmp_buf cassert_env;
#define cassert_loop() (setjmp (cassert_env) || 1)
#define cassert(cond) do { if (!cond) { longjmp (cassert_env, 1); } } while (0)
void constraint_resolve (derivation_t * d, size_t n);
```

Fig. 9. Top-level functions

```
int32_t main () {
  derivation_t d[4];
  while (cassert_loop ()) { constraint_resolve (d, 4); ns_run_once (d); break; }
  return 0;
}
```

Fig. 10. Top-level attack on Needham–Schroeder

$$K,L,M,N ::= b \mid k \mid n \mid (M_1,\ldots,M_j) \mid \{|M|\}_N \mid \mathsf{dec}(M) \mid \mathsf{enc}(M)$$

b ranges over bytes $0,\ldots,255$ k ranges over a set of keypairs n ranges over a set of nonces

Fig. 11. Grammar of messages

This makes use of the top-level functions given in Figure 9. The cassert_loop macro uses setjmp to restart the protocol from the beginning, which occurs when a cassert fails and calls longjmp. The constraint_resolve function calls out to XSB Prolog to find appropriate values for d[] based on constraints generated by the previous run of the protocol.

Example: Finding The Attack On Needham–Schroeder The main loop for finding the attack on Needham–Schroeder is given in Figure 10. It finds the attack on the unmodified protocol in 11 runs, and fails to find an attack on Lowe's variant in 14 runs.

3 Implementing DYC

We will now look under the hood of DYC, to see how the APIs are implemented. The interesting parts of the implementation are how the honest agents generate constraints and the model of the attacker, including derivations.

3.1 Implementing Honest Agents

The implementation of the honest agents is relatively straightforward: we provide a simple formal definition of messages manipulated by the honest agents, and have each function in the honest agent API generate constraints.

The grammar of messages is given in Figure 11. Atomic messages are either bytes, keypairs or nonces, and can be built up by tupling, encryption, or by accessing the decryption or encryption key from a keypair.

```
data_t crypto_pencrypt (data_t key_p, data_t pdata) {
  data_t cdata; ... raw libgcrypt ...
  constraint_pencrypt (cdata, key_p, pdata); return cdata;
}
```

Fig. 12. Implementation of public key encryption

```
prolog_term query;

void constraint_append (prolog_term x) {
  /* query = ,(query, x) */
  prolog_term tmp = constraint_functor (",", 2);
  constraint_arg_term (tmp, 1, query); constraint_arg_term (tmp, 2, x);
  query = tmp;
}

void constraint_pencrypt (data_t c, data_t pk_p, data_t p) {
  /* =(c, pencrypt(pk_p, p)) */
  prolog_term pencrypt, eq;
  pencrypt = constraint_functor ("pencrypt", 2);
  constraint_arg_data (pencrypt, 1, pk_p); constraint_arg_data (pencrypt, 2, p);
  eq = constraint_functor ("=", 2);
  constraint_arg_data (eq, 1, c); constraint_arg_term (eq, 2, pencrypt);
  constraint_append (eq);
}
```

Fig. 13. C implementation of constraints for public key encryption

```
prolog_term constraint_functor (char * f, size_t n);
void constraint_arg_data (prolog_term fn, size_t argn, data_t d);
void constraint_arg_term (prolog_term fn, size_t argn, prolog_term t);
```

Fig. 14. C API for building Prolog queries

```
ispencrypt(dec(K), X, pencrypt(enc(K), X)).
```

Fig. 15. Prolog implementation of constraints for public key encryption

```
proj([M|_], 0, L, M) :- length(M, L).
proj([M|Ms], O, L, N) :- length(M, Lm), proj(Ms, O-Lm, L, N).

length(key(_), 699). length(dec(_), 528). length(enc(_), 171).
length(pencrypt(_,_), 157). length(nonce(_), 16). length(byte(_), 1).
length([], 0). length([M,Ms], N+Ns) :- length(M, N), length(Ms, Ns).
```

Fig. 16. Prolog implementation of constraints for tuples

We represent messages as Prolog terms, for example $\{|N_A, N_B, B|\}_{\mathsf{enc}(K_A)}$ is represented in Prolog as:

```
pencrypt(enc(key(1)), [ nonce(1), nonce(2), byte(65) ])
```

Each of the honest agent functions generates a constraint, in addition to performing its computation. For example, the implementation of public key encryption in Figure 12 generates constraints using the functions given in Figure 13. This in turn makes use of a C API for building Prolog queries given in Figure 14. The constraints are built up as a comma-separated list in a prolog_term variable query. For example, the function calls:

```
m = crypto_pencrypt (k_p, x); y = crypto_pdecrypt (k_s, m);
```

generates constraints:

```
Vm = pencrypt(Vk_p, Vx), ispencrypt(Vk_s, Vy, Vm), ...
```

where Vm is a variable name generated from the memory location of variable m. The Prolog constraint for public key encryption is given in Figure 15.

When a fresh key or nonce is generated, for example:

```
k = crypto_pcreate ();
```

we increment a counter, and generate an appropriate constraint such as:

```
Vk = key(1), ...
```

For example, Needham–Schroeder generates nonce(1) for N_A, nonce(2) for N_B, key(1) for K_A, and so on.

The C API for tuples is based on offset-and-length projections, for example:

```
x = data_sub (y, off, len);
```

which generates a Prolog constraint:

```
proj(Vy, Voff, Vlen, Vx), ...
```

defined in Figure 16.[3]

The rest of the implementation of honest agents is unsurprising: most of the novelty of DYC is in the implementation of attackers.

3.2 Implementing Attackers

The attacker model is based on Abadi and Fiore's [1] model, which in turn is based on Clarke, Jha and Marrero's algorithm [9] for Dolev–Yao [11] message derivability. In addition, we use a technique for termination based on Millen and Shmatikov [25]. The resulting system allows us to make use of the built-in Prolog constraint satisfaction

[3] Unfortunately, our implementation currently restricts this predicate so that it will not instantiate Vx as a list: removing this restriction causes XSB to routinely run out of memory. We leave this for future study.

algorithm, in comparison to Millen and Shmatikov, who build a constraint satisfaction and unification algorithm on top of Prolog's.

Unfortunately, we cannot quite use Abadi and Fiore's algorithm off-the-shelf, as it uses negative conditions (of the form $M \notin S$) which do not interact well with Prolog. Most of the algorithm is the same, and depends on proof normalization, where introduction steps are never used as sub-goals of elimination steps. The difference between our algorithm and theirs is the mechanism for achieving termination, as even normalized proofs can diverge:

$$\dfrac{\dfrac{}{\{|\mathsf{dec}(k)|\}_{\mathsf{enc}(k)} \vdash \{|\mathsf{dec}(k)|\}_{\mathsf{enc}(k)}} \quad \dfrac{\vdots}{\{|\mathsf{dec}(k)|\}_{\mathsf{enc}(k)} \vdash \mathsf{dec}(k)}}{\{|\mathsf{dec}(k)|\}_{\mathsf{enc}(k)} \vdash \mathsf{dec}(k)}$$

Such divergent normalized proofs have the property that they are cyclic: there is a sub-derivation containing a strict sub-sub-derivation with the same conclusion (in this case $\mathsf{dec}(k)$). In Abadi and Fiore's terms, such proofs are not *simple*. Abadi and Fiore use a tabulation technique to detect such cycles: they keep track of a set S of messages which have already appeared as conclusions, and appropriate conditions $M \notin S$ to judgments with conclusion M. This ensures acyclicity, and hence termination. For example, the previous divergence now becomes a failed derivation:

$$\dfrac{\cdots \quad \dfrac{\textbf{false}}{\{|\mathsf{dec}(k)|\}_{\mathsf{enc}(k)} \vdash \mathsf{dec}(k) \notin \{\mathsf{dec}(k)\}}}{\{|\mathsf{dec}(k)|\}_{\mathsf{enc}(k)} \vdash \mathsf{dec}(k) \notin \emptyset}$$

We avoid cycles in a different fashion, similar to Millen and Shmatikov's [25] use of a "special term" $\lceil M \rceil_{\mathsf{enc}(k)}$ to replace a ciphertext $\{|M|\}_{\mathsf{enc}(k)}$ while deriving the key $\mathsf{dec}(k)$ (their system used this technique for symmetric encryption rather than asymmetric, but that does not impact its applicability). We do not require any properties of this special term, and can use any placeholder such as 0 in its stead. For example, the previous divergence now becomes a failed derivation:

$$\dfrac{\cdots \quad \dfrac{\textbf{false}}{\{|0|\}_{\mathsf{enc}(k)} \vdash \mathsf{dec}(k)}}{\{|\mathsf{dec}(k)|\}_{\mathsf{enc}(k)} \vdash \mathsf{dec}(k)}$$

The main formal result of this paper is that this variant of Abadi and Fiore's algorithm is sound and complete, and guarantees termination of proof search.

The *message derivation* relation $K \vdash M$ is defined in Figure 17. A *derivation tree* D is a (possibly infinite) tree with each node labeled by a message derivation rule name. We will write $r(D_1, \ldots, D_n)$ for the derivation tree with root node labeled r and children $D_1, \ldots, D_n$. We say that $D \rhd K \vdash M$ is a *valid derivation* whenever D is a finite derivation tree for the judgment $K \vdash M$, that is $r(D_1, \ldots, D_n) \rhd K \vdash M$ is a valid derivation whenever there is an instantiation of rule r of the form:

$$\dfrac{K_1 \vdash M_1 \quad \cdots \quad K_n \vdash M_n}{K \vdash M}\,[r]$$

$$\frac{}{M \vdash M}[\text{ID}] \quad \frac{K \vdash (M_1,\ldots,M_n)}{K \vdash M_i}[\text{TUPLEE}_i] \quad \frac{K \vdash \{|M|\}_{\mathsf{enc}(k)} \quad K \vdash \mathsf{dec}(k)}{K \vdash M}[\text{PENCRYPTE}]$$

$$\frac{K \vdash \mathsf{enc}(M) \quad K \vdash \mathsf{dec}(M)}{K \vdash M}[\text{ENCDECE}] \quad \frac{}{K \vdash b}[\text{BYTEI}_b] \quad \frac{K \vdash M_1 \quad \cdots \quad K \vdash M_n}{K \vdash (M_1,\ldots,M_n)}[\text{TUPLEI}]$$

$$\frac{K \vdash M_1 \quad K \vdash \mathsf{enc}(M_2)}{K \vdash \{|M_1|\}_{\mathsf{enc}(M_2)}}[\text{PENCRYPTI}] \quad \frac{K \vdash M}{K \vdash \mathsf{enc}(M)}[\text{ENCI}] \quad \frac{K \vdash M}{K \vdash \mathsf{dec}(M)}[\text{DECI}]$$

Fig. 17. Message derivation

where $D_i \rhd K_i \vdash M_i$ are valid derivations for $1 \leq i \leq n$.

We say that $D \rhd K \vdash M$ is a *diverging derivation* whenever D is an infinite derivation tree showing that the search for judgment $K \vdash M$ may fail to terminate, that is $r(D_1,\ldots,D_j) \rhd K \vdash M$ is a diverging derivation whenever there is an instantiation of rule r of the form:

$$\frac{K_1 \vdash M_1 \quad \cdots \quad K_n \vdash M_n}{K \vdash M}[r]$$

where $D_i \rhd K_i \vdash M_i$ are valid derivations for $1 \leq i < j \leq n$ and $D_j \rhd K_j \vdash M_j$ is a diverging derivation. As noted above, there is a diverging derivation for $D \rhd \{|\mathsf{dec}(k)|\}_{\mathsf{enc}(k)} \vdash \mathsf{dec}(k)$ given by the infinite derivation tree $D = \text{PENCRYPTE}(\text{ID}, D)$.

The *active contexts* $\mathcal{C}, \mathcal{D}, \mathcal{E}$ are given by:

$$\mathcal{C}, \mathcal{D}, \mathcal{E} ::= \cdot \mid (M_1,\ldots,M_i,\mathcal{C},M_{i+2},\ldots,M_j) \mid \{|\mathcal{C}|\}_M$$

We write $\mathcal{C}[M]$ for the message given by replacing the hole $\cdot$ in the context $\mathcal{C}$ by the message M. The active contexts are ones where it is possible to recover the contents, that is there exists some K such that for any N we have that $(K, \mathcal{C}[N]) \vdash N$. We can say that N is an *active sub-term* of M if we can find some active context $\mathcal{C}$ such that $M = \mathcal{C}[N]$. Abadi and Fiore formulated their algorithm using the notion of active sub-term, where we find it more convenient to use the notion of active context.

The *normalized message derivation* relation $K \vdash_N M$ is given in Figure 18. It differs from the non-normalized message derivation relation in that:

1. The use of introduction rules as sub-derivations of elimination rules is very restricted: they are only allowed in one particular case, whose equivalent in the non-normalized message derivation system is:

$$\frac{K \vdash \{|M|\}_{\mathsf{enc}(k)} \quad \frac{K \vdash k}{K \vdash \mathsf{dec}(k)}[\text{DECI}]}{K \vdash M}[\text{PENCRYPTE}]$$

   This stratification uses a subsystem $K \vdash_E M$ for elimination rules.

$$\frac{}{M \vdash_E M}[\text{ID}] \quad \frac{K = \mathcal{C}[M_i] \quad K \vdash_E (M_1,\ldots,M_j)}{K \vdash_E M_i}[\text{TUPLEE}_i] \quad \frac{K = \mathcal{C}[M] \quad K' = \mathcal{C}[0] \quad K \vdash_E \{|M|\}_{\mathsf{enc}(k)} \quad K' \vdash_E \mathsf{dec}(k)}{K \vdash_E M}[\text{PENCRYPTE}]$$

$$\frac{K = \mathcal{C}[M] \quad K \vdash_E \mathsf{enc}(M) \quad K \vdash_E \mathsf{dec}(M)}{K \vdash_E M}[\text{ENCDECE}] \quad \frac{K = \mathcal{C}[M] \quad K' = \mathcal{C}[0] \quad K \vdash_E \{|M|\}_{\mathsf{enc}(k)} \quad K' \vdash_E k}{K \vdash_E M}[\text{PENCRYPTE}(\cdot,\text{DECI}(\cdot))]$$

$$\frac{K \vdash_E M}{K \vdash_N M} \quad \frac{}{K \vdash_N b}[\text{BYTEI}_b] \quad \frac{K \vdash_N M_1 \quad \cdots \quad K \vdash_N M_n}{K \vdash_N (M_1,\ldots,M_n)}[\text{TUPLEI}]$$

$$\frac{K \vdash_N M_1 \quad K \vdash_N \mathsf{enc}(M_2)}{K \vdash_N \{|M_1|\}_{\mathsf{enc}(M_2)}}[\text{PENCRYPTI}] \quad \frac{K \vdash_N M}{K \vdash_N \mathsf{enc}(M)}[\text{ENCI}] \quad \frac{K \vdash_N M}{K \vdash_N \mathsf{dec}(M)}[\text{DECI}]$$

Fig. 18. Normalized message derivation

2. In any derivation $K \vdash_E M$, we explicitly require M to be an active sub-term of K, rather than having this be a derived property of the system. This avoids diverging derivations such as $D \rhd K \vdash M$ given by the infinite derivation tree $D = \text{TUPLEE}_1(D)$.
3. In the uses of PENCRYPTE, we remove an occurrence of the plaintext M from the attacker knowledge when deriving the key $\mathsf{dec}(k)$. This avoids diverging derivations such as $D \rhd \{|\mathsf{dec}(k)|\}_{\mathsf{enc}(k)} \vdash \mathsf{dec}(k)$ given by the infinite derivation tree $D = \text{PENCRYPTE}(\text{ID}, D)$.

We then show that the normalized message derivation system can be used as an algorithm for message derivation, with proofs given in Appendix A

Proposition 1. *There are no diverging derivations $D \rhd K \vdash_N M$.*

Proposition 2. *If $D \rhd K \vdash_N M$ is a valid derivation, then so is $D \rhd K \vdash M$.*

Proposition 3. *If $D \rhd K \vdash M$ is a valid derivation, then there exists a valid derivation $D' \rhd K \vdash_N M$.*

The Prolog attacker model is almost a direct translation of Figure 18. For example, the implementation of public key encryption is given in Figure 19. The only place which is not a direct translation is the treatment of projections, since C uses offset-and-length to project tuples. This is achieved by including the offset-and-length information in TUPLEE, and making use of the proj predicate from Figure 16.

The C attacker function derivation_build(D,K) builds a message M such that $D \rhd K \vdash M$. There is a direct representation of the derivation tree D in C, and the implementation of derivation_build is a direct recursion over D. Note that we do *not* generate constraints for the message built by the attacker, except that we put down the constraint constraint validN(D,K,M).

The only remaining function is constraint_resolve, which takes the query containing the constraints from the previous run, solves it using XSB Prolog, and translates the solved derivations back from XSB Prolog into C.

```
validE(pencryptE(D1, decI(D2)), K, M) :-
  subtermRemove(K, M, L), validE(D1, K, pencrypt(enc(N), M)), validE(D2, L, N).
validE(pencryptE(D1, D2), K, M) :-
  subtermRemove(K, M, L), validE(D1, K, pencrypt(enc(N), M)), validE(D2, L, dec(N)).

validN(pencryptI(D1, D2), K, pencrypt(enc(N), M)) :-
  validN(D1, K, enc(N)), validN(D2, K, M).

subtermRemove(M, M, byte(0)).
subtermRemove([M|Ms], N, [L|Ms]) :- subtermRemove(M, N, L).
subtermRemove([M|Ms], N, [M|Ls]) :- subtermRemove(Ms, N, Ls).
subtermRemove(pencrypt(K, M), N, pencrypt(K, L)) :- subtermRemove(M, N, L).
```

Fig. 19. Attacker model for public key encryption in Prolog

4 Conclusions and Future Work

We have seen how DYC allows C programs written using an appropriate cryptographic API to be model-checked to find attacks. The attacker model is a direct translation of Dolev–Yao message derivability into Prolog, and of derivation trees into C. We believe that this research direction shows great promise, although there a number of limitations, which we hope to address in future work.

We require honest agents to be straight-line code. The difficult issue here is that the control flow of the honest agents may depend on message contents, so there many be an interaction between constraint generation and state-space exploration. We hope to investigate integration with model checkers such as DART [15] to address this issue. We hope that such an integration would also avoid the current requirement for programmer-specified message flow.

The DYC API is missing cryptographic features such as symmetric encryption, signing, hashing, composite keys and Diffie-Hellman, and is also missing a socket model. We do not expect that these features will cause significant problems, and would allow us to investigate more examples.

Despite its rough edges, we believe that DYC is an interesting first step at merging the fields of software model checking and cryptographic protocol validation, and could lead to tools capable of validating full-fledged cryptographic protocol implementations rather than specifications.

A Proofs

Proposition 1. There are no diverging derivations $D \rhd K \vdash_N M$.

Proof. Define the size of a message as:

$$\begin{aligned} \mathsf{size}(b) = \mathsf{size}(n) &= 1 \\ \mathsf{size}(k) &= 2 \\ \mathsf{size}(M_1, \ldots, M_n) &= 1 + \textstyle\sum_{i=1}^{n} \mathsf{size}(M_i) \end{aligned}$$

$$\mathsf{size}(\{|M|\}_N) = 1 + \mathsf{size}(M) + \mathsf{size}(N)$$
$$\mathsf{size}(\mathsf{enc}(M)) = \mathsf{size}(\mathsf{dec}(M)) = 1 + \mathsf{size}(M)$$

We first note that if $D \rhd K \vdash_E M$ is either a valid or a diverging derivation then M is an active sub-term of K, and so $\mathsf{size}(K) - \mathsf{size}(M)$ is guaranteed to be non-negative and can be used for induction.

We then show by induction on $\mathcal{C}$ that if $K = \mathcal{C}[M]$ and $K' = \mathcal{C}[0]$ then $\mathsf{size}(K) = \mathsf{size}(K') + \mathsf{size}(M) - 1$.

We then show that there are no diverging derivations $D \rhd K \vdash_E M$. We proceed by induction on $\mathsf{size}(K)$, with a nested induction on $\mathsf{size}(K) - \mathsf{size}(M)$. D must be of the form $r(D_1, \ldots, D_j)$ where r is an elimination rule whose jth hypothesis has a diverging derivation $D_j \rhd K_j \vdash_E M_j$. The interesting case is $r = \text{PENCRYPTE}$, for which we have:

$$D_1 \rhd K \vdash_E \{|M|\}_{\mathsf{enc}(k)}$$

and by induction on $\mathsf{size}(K) - \mathsf{size}(\{|M|\}_{\mathsf{enc}(k)})$ we have that $D_1 \rhd K \vdash_E \{|M|\}_{\mathsf{enc}(k)}$ is non-diverging, so we must have:

$$K = \mathcal{C}[M] \quad K' = \mathcal{C}[0] \quad D_2 \rhd K' \vdash_E \mathsf{dec}(k)$$

We now have two cases, depending on whether $\mathsf{size}(K') < \mathsf{size}(K)$ or not. If $\mathsf{size}(K') < \mathsf{size}(K)$ then we can proceed by induction on $\mathsf{size}(K')$. Otherwise, $\mathsf{size}(K) = \mathsf{size}(K')$ and $\mathsf{size}(M) = 1$, so since $\mathsf{size}(\mathsf{dec}(k)) = 3$ we can proceed by induction on $\mathsf{size}(K') - \mathsf{size}(\mathsf{dec}(k))$. In either case, we have that $D_2 \rhd K' \vdash_E \mathsf{dec}(k)$ is non-diverging, and so we have $D \rhd K \vdash_E M$ non-diverging as required. The other cases are either similar, or are straightforward inductions.

Finally, we show that there are no diverging derivations $D \rhd K \vdash_N M$, which is a straightforward induction on M, as all the rules are syntax-directed on M.

Proposition 2. If $D \rhd K \vdash_N M$ is a valid derivation, then so is $D \rhd K \vdash M$.

Proof. We first show by induction on D that if:

$$D \rhd K' \vdash_E M' \quad K = \mathcal{C}[N] \quad K' = \mathcal{C}[0]$$

then either:

$$D \rhd K \vdash_E M'$$

or:

$$D \rhd K \vdash_E M \quad M = \mathcal{D}[N] \quad M' = \mathcal{D}[0]$$

The interesting case is $D = \text{PENCRYPTE}(D_1, D_2)$, for which we have:

$$K' = \mathcal{C}'[M'] \quad K'' = \mathcal{C}'[0]$$
$$D_1 \rhd K' \vdash_E \{|M'|\}_{\mathsf{enc}(k)} \quad D_2 \rhd K'' \vdash_E \mathsf{dec}(k)$$

We can induct on D_2 (and note that there is no $\mathcal{D}'$ such that $\mathcal{D}'[0] = \mathsf{dec}(k)$) to get:

$$D_2 \rhd K' \vdash_E \mathsf{dec}(k)$$

and then again to get:

$$D_2 \rhd K \vdash_E \mathsf{dec}(k)$$

We can then induct on D_1 to get two sub-cases. Either:

$$D_1 \rhd K \vdash_E \{|M'|\}_{\mathsf{enc}(k)}$$

and so:

$$D \rhd K \vdash_E M'$$

as required. Otherwise:

$$D_1 \rhd K \vdash_E L \quad L = \mathcal{E}[N] \quad \{|M'|\}_{\mathsf{enc}(k)} = \mathcal{E}[0]$$

and so we must have $\mathcal{E}$ of the form $\mathcal{E} = \{|\mathcal{D}|\}_{\mathsf{enc}(k)}$, and so we can define $M = \mathcal{D}[N]$ and get:

$$D_1 \rhd K \vdash_E \{|M|\}_{\mathsf{enc}(k)} \quad M = \mathcal{D}[N] \quad M' = \mathcal{D}[0]$$

and so:

$$D \rhd K \vdash_E M \quad M = \mathcal{D}[N] \quad M' = \mathcal{D}[0]$$

as required. The other cases are similar, or are direct inductions.

We then show that if $D \rhd K \vdash_N M$ is a valid derivation, then so is $D \rhd K \vdash M$, by induction on D, making use of the previous result in the cases involving PENCRYPTE.

Proposition 3. If $D \rhd K \vdash M$ is a valid derivation, then there exists a valid derivation $D' \rhd K \vdash_N M$.

Proof. Define:

1. The *simple* derivations $D \rhd K \vdash M$ are the valid derivations where any sub-derivation $D' \rhd K \vdash M'$ of D with sub-sub-derivation $D'' \rhd K \vdash M''$ of D' has $M \neq M'$,
2. The *eliminatable* derivations $D \rhd K \vdash M$ are the simple derivations where any use of an introduction rule must be of the form $\text{PENCRYPTE}(\cdot, \text{DECI}(\cdot))$, and
3. The *normalizable* derivations $D \rhd K \vdash M$ are the simple derivations where any use of an introduction rule inside an elimination rule must be of the form $\text{PENCRYPTE}(\cdot, \text{DECI}(\cdot))$.

We then prove a series of results:

1. If $D \rhd K \vdash M$ is valid, then there exists a simple $D' \rhd K \vdash M$. The derivation D' is given by replacing any sub-derivation violating simplicity by the appropriate sub-sub-derivation.
2. Any simple derivation is normalizable. This is direct, by analysis of each possible pair of elimination rules and introduction rules.
3. For any eliminatable derivation: $D \rhd K \vdash N$ where $K = \mathcal{C}[M]$ and $K' = \mathcal{C}[0]$, either:
   (a) $D \rhd K' \vdash N$,
   (b) $D \rhd K' \vdash N'$ where $N = \mathcal{D}[M]$ and $N' = \mathcal{C}'[0]$, or
   (c) $D \rhd K \vdash N$ has sub-derivation $D' \rhd K \vdash M$.
   This is an induction on D.

4. For any eliminatable $D \rhd K \vdash M$, we have valid $D \rhd K \vdash_E M$. This is an induction on D, where the interesting case is $D = \text{PENCRYPTE}(D_1, D_2)$. Since D is eliminatable, we have D_2 is eliminatable, or is of the form $D_2 = \text{DECI}(D_3)$ and D_3 is eliminatable. We shall consider the first case, the second is similar. We have:

$$D_1 \rhd K \vdash \{|M|\}_{\mathsf{enc}(k)} \quad D_2 \rhd K \vdash \mathsf{dec}(k)$$

By induction, and inspection of $\vdash_E$:

$$D_1 \rhd K \vdash_E \{|M|\}_{\mathsf{enc}(k)} \quad K = \mathcal{C}[M]$$

Thus, by the previous result (and the fact that $D \rhd K \vdash M$ is simple, and hence can have no sub-derivation $D' \rhd K \vdash M$) we have:

$$D_2 \rhd K' \vdash \mathsf{dec}(k) \quad K' = \mathcal{C}[0]$$

so by induction: $D_2 \rhd K' \vdash_E \mathsf{dec}(k)$ and hence $D \rhd K \vdash_E M$ as required.
5. For any normalizable $D \rhd K \vdash M$, we have valid $D \rhd K \vdash_N M$. This is a direct induction on D.

The result follows: for any valid $D \rhd K \vdash M$, we first find a simple (and hence normalizable) $D' \rhd K \vdash M$, from which we get $D' \rhd K \vdash_N M$.

References

1. M. Abadi and M. Fiore. Computing symbolic models for verifying cryptographic protocols. In *Proc. IEEE Computer Security Foundations Workshop*, pages 160–173, 2001.
2. M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information And Computation*, 148:1–70, 1999.
3. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. SPIN Workshop on Model Checking of Software*, number 2057 in Lecture Notes in Computer Science, pages 103–122. Springer-Verlag, 2001.
4. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. ACM Symp. Principles of Programming Languages*, pages 1–3, 2002.
5. K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *Proc. IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2006. To appear.
6. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. IEEE Computer Security Foundations Workshop*, pages 82–96, 2001.
7. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer-Verlag, 2004.
8. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.
9. E. M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. IFIP Working Conference on Programming Concepts and Methods*, 1998.
10. G. Denker, J. Millen, and H. Ruess. The CAPSL integrated protocol environment. Technical Report SRI-CSL-2000-02, Computer Science Laboratory, SRI International, 2000.
11. D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Trans. Information Theory*, 29(2):198–208, 1983.

12. P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. ACM Symp. Principles of Programming Languages*, pages 174–186, 1997.
13. P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *Int. J. Software Tools for Technology Transfer*, 6(2):117–127, 2004.
14. P. Godefroid and N. Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In *Proc. Integrated Formal Methods*, volume 3771 of *Lecture Notes in Computer Science*, pages 20–32. Springer-Verlag, 2005.
15. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. ACM Conf. Programming Language Design and Implementation*, pages 213–223, 2005.
16. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *Proc. Int. Conf. Verification, Model Checking and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 363–379. Springer-Verlag, 2005.
17. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Int. J. Software Tools for Technology Transfer*, 2(4), 2000.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and Gregoire Sutre. Software verification with Blast. In *Proc. SPIN Workshop on Model Checking of Software*, number 2648 in Lecture Notes in Computer Science, pages 235–239. Springer-Verlag, 2003.
19. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
20. A. S. A. Jeffrey and R. Ley-Wild. Dolev–Yao C implementation. http://cm.bell-labs.com/who/ajeffrey/dyc.tgz.
21. W. Koch et al. Libgcrypt - cryptographic library. http://directory.fsf.org/security/libgcrypt.html.
22. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lectures Notes in Computer Science*, pages 147–166. Springer, 1996.
23. G. Lowe. Casper: A compiler for the analysis of security protocols. *J. Computer Security*, 6:53–84, 1998.
24. W. Marrero, E. M. Clarke, and S. Jha. Model checking for security protocols. In *Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997.
25. J. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. ACM Conf. Computer and Communication Security*, pages 166–175, 2001.
26. M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proc. Symp. USENIX Operating Systems Design and Implementation*, pages 75–88, 2002.
27. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
28. I. V. Ramkrishnan et al. XSB Prolog. http://xsb.sourceforge.net/.
29. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proc. European Software Engineering Conf. and ACM SIGSOFT Symp. Foundations of Software Engineering*, pages 267–276, 2003.
30. T. Y. C. Woo and S. S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993.