Cryptographic Protocol Analysis on Real C Code*

Jean Goubault-Larrecq[1] Fabrice Parrennes[1,2]

[1] LSV/CNRS UMR 8643 & INRIA Futurs projet SECSI & ENS Cachan
61 avenue du président-Wilson, F-94235 Cachan Cedex
goubault@lsv.ens-cachan.fr Phone: +33-1 47 40 75 68
[2] RATP, EST/ISF/QS LAC VC42 40 bis Roger Salengro, F-94724 Fontenay-sous-Bois
fabrice.parrennes@ratp.fr Phone: +33-1 58 77 04 65

Abstract. Implementations of cryptographic protocols, such as OpenSSL for example, contain bugs affecting security, which cannot be detected by just analyzing abstract protocols (e.g., SSL or TLS). We describe how cryptographic protocol verification techniques based on solving clause sets can be applied to detect vulnerabilities of C programs in the Dolev-Yao model, statically. This involves integrating fairly simple pointer analysis techniques with an analysis of which messages an external intruder may collect and forge. This also involves relating concrete run-time data with abstract, logical terms representing messages. To this end, we make use of so-called *trust assertions*. The output of the analysis is a set of clauses in the decidable class $\mathcal{H}_1$, which can then be solved independently. This can be used to establish secrecy properties, and to detect some other bugs.

1 Introduction

Cryptographic protocol verification has come of age: there are now many ways of verifying cryptographic protocols in the literature (see [12] for a sampler). They all start from a fairly abstract specification of the protocol. However, in real life, what you use when you type `ssh` or when you connect to a securized site on your Web browser is not a 5-line abstract protocol but a complete program. While this program is intended to implement some protocol, there is no guarantee it actually implements it in any way. The purpose of this paper is to make a few first steps in the direction of analyzing cryptographic protocols directly from source code.

To make things concrete, here is a specification of the public-key Needham-Schroeder protocol in standard notation (right). The goal is for A and B to exchange their secret texts $N_{\mathtt{A}}$ and $N_{\mathtt{B}}$ while authenticating themselves mutually [19]. It is well-known that there is an attack against this protocol (see [17]). This attack also makes $N_{\mathtt{B}}$ available to the intruder, although $N_{\mathtt{B}}$ was meant to remain secret.

1. $\mathtt{A} \to \mathtt{B}$: $\{N_{\mathtt{A}}, \mathtt{A}\}_{\mathtt{pub(B)}}$
2. $\mathtt{B} \to \mathtt{A}$:$\{N_{\mathtt{A}}, N_{\mathtt{B}}\}_{\mathtt{pub(A)}}$
3. $\mathtt{A} \to \mathtt{B}$:$\{N_{\mathtt{B}}\}_{\mathtt{pub(B)}}$

Fig. 1. The NS protocol

Figure 1 reads as follows: any agent implementing A's role will first create a fresh *nonce* $N_{\mathtt{A}}$, typically by drawing a number at random, then build the pair $(N_{\mathtt{A}}, \mathtt{A})$ where A is taken to be A's identity (some string identifying A uniquely by convention), then encrypt the result using B's public key $\mathtt{pub(B)}$. The encrypted text $\{N_{\mathtt{A}}, \mathtt{A}\}_{\mathtt{pub(B)}}$ is then sent out. If traffic is not diverted, this should reach B, who will decrypt this using his

* Partially supported by the ACI jeunes chercheurs "Sécurité informatique, protocoles cryptographiques et détection d'intrusions" and the ACI cryptologie "Psi-Robuste". Work done while the second author was at LSV.

private key $\mathtt{prv(B)}$, and send back $\{N_\mathtt{A}, N_\mathtt{B}\}_{\mathtt{pub(A)}}$ to A. A waits for such a message at step 2., decrypts it using her private key $\mathtt{prv(A)}$, checks that the first component is indeed $N_\mathtt{A}$, then sends back $\{N_\mathtt{B}\}_{\mathtt{pub(B)}}$ at step 3. for confirmation.

Compare this specification (Figure 1) with excerpts from an actual C implementation of A's role in it (Figure 2). First, the C code is longer than the specification (although Figure 2 only implements message 1 of Figure 1). Difficulties in analyzing such a piece of C code mainly come from other, less visible, problems:

```
1  int create_nonce (nonce_t *nce)
2  {
3    RAND_bytes(nce->nonce,SIZENONCE);
4    /* % *nce rec nonce(CTX) | context(CTX). % */
5    return(0);
6  }
7
8  int encrypt_mesg(msg1_t *msg, BIGNUM *key_pub,
9                   BIGNUM *key_mod, BIGNUM *cipher)
10 {
11   BIGNUM *plain;
12   int msg_len;
13   BN_CTX *ctx;
14   ctx = BN_CTX_new();
15   msg_len = sizeof (msg1_t);
16   plain = BN_bin2bn((const unsigned char *)msg, msg_len, NULL);
17   BN_CTX_init(ctx);
18   BN_mod_exp(cipher, plain, key_pub, key_mod, ctx);
19
20   /* % *cipher rec crypt(M,K) | *msg rec M, *key_pub rec K. % */
21
22    return (0);
23 }
24
25 int create_mesg1(msg1_t *mesg, nonce_t *n1, int *id, int *dest)
26 {
27   /* First copy nonce. */
28   memcpy (&msg->nonce_msg1, n1, sizeof(nonce_t));
29
30   /* copy id... */
31   msg->id_1[0] = id[0]; msg->id_1[1] = id[1];
32   msg->id_1[2] = id[2]; msg->id_1[3] = id[3];
33   /* ... and dest. */
34   msg->dest_1[0] = dest[0]; msg->dest_1[1] = dest[1];
35   msg->dest_1[2] = dest[2]; msg->dest_1[3] = dest[3];
36
37   /* % *msg -> nonce_msg1 rec U and
38        *msg -> is_1       rec V and
39        *msg -> dest_1     rev W     | *n1 rec U, *id rec V,
40                                       *dest rec W. % */
41   return(0);
42 }
43
44 int write(int fd, const void *buf, int count)
45 {
46   write (fd, buf, count);
47   /* % knows rec B | *buf rec B. % */
48   return(0);
49 }
```

```
50 int main(int argc, char *argv[])
51 {
52   int conn_fd;  // The communication socket.
53   msg1_t mesg1; // Message
54   nonce_t nonce;
55   BIGNUM * cipher1; // Cipher Message
56   BIGNUM * pubkey;  // Keys
57   BIGNUM * prvkey;  // Keys
58   BIGNUM * modkey;  // Keys
59   unsigned int ip_id[4];   // A's name
60   unsigned int ip_dest[4]; // B's name as seen from A.
61
62   /* Init ip_id and ip_dest. */
63   ip_id[0]   = 192; ip_id[1]   = 100;
64   ip_id[2]   = 200; ip_id[3]   = 100;
65   ip_dest[0] = 192; ip_dest[1] = 100;
66   ip_dest[2] = 200; ip_dest[3] = 101;
67   /* % ip_id   rec CTX(Agent(A)).  % */
68   /* % ip_dest rec CTX(Agent(B)).  % */
69   // Open connection to B
70   conn_fd = connect_socket(ip_dest, 522);
71
72   init_keys(&pubkey, &prvkey, &modkey, PUBALICESERV,
73   MODALICESERV, PRIVALICESERV);
74   /* % *pubkey rec pub(Y)  | ip_dest rec Y. % */
75   /* % *prvkey rec priv(X) | ip_id rec X.   % */
76
77   /*** 1. A -> B : {Na, A}_pub(B) ***/
78   create_nonce (&nonce);
79   create_mesg1(&mesg1, &nonce, ip_id, ip_dest);
80   cipher1 = BN_new();
81   encrypt_mesg(&mesg1, pubkey, modkey, cipher1);
82   write(conn_fd, cipher1, 128);
83
84   /** ...Remaining code omitted... **/
85 }
```

Fig. 2. A piece of code of a sample C implementation of the NS protocol

- First, C is a real programming language, with memory allocation, aliasing, pointer arithmetic; all this is absent from protocol specifications, and must be taken into account. E.g., in Figure 2, line 80, the pointer `cipher1` is set to the address allocated by `BN_new()`; at line 81, the encryption function `encrypt_mesg` expects to encrypt its first argument with the key in second and third arguments, putting the result at the address pointed to by its fourth argument `cipher1`.
- C programs are meant to be linked to external libraries, whose code is usually unavailable (e.g., `memcpy`, `strcpy`, `strncmp`, `read`, `write` in Figure 2) and cannot be analyzed. More subtly, low-level encryption functions should *not* be analyzed, simply because we do not know any way to recognize that some given

bit-mangling code implements, say, RSA or DES. We shall take the approach that such functions should be *trusted* to do what they are meant to do.

– Even without looking at the intricacies of statically analyzing C code, we usually only have the source code of *one* role at our disposal. For example, the code of Figure 2 implements `A`'s role in the protocol of Figure 1, not `B`'s, not anyone else's either. So we shall analyze C code modulo an abstract description of the world around it. This so-called *external trust model* will state what malicious intruders can do, and what honest agents are trusted to do (e.g, if `B` is assumed to be honest, he should only be able to execute the corresponding steps in Figure 1).

  Alternatively, we could also analyze the source code of two or more roles. But we would still need an external trust model, representing malicious intruders, and honest agents of other protocols which may share secrets with the analyzed programs.

What we do in this paper. We analyze reachability properties of C code implementing roles of cryptographic protocols. Amongst all reachability properties, we shall concentrate on (non-)*secrecy*, i.e., the ability for a malicious intruder to get hold of some designated, sensitive piece of data. All problems considered here are undecidable: we therefore concentrate on upper approximations of behaviors of programs, i.e., on representations that contain at least all behaviors that the given program may exhibit—in a given external trust model, and a given execution model (see below). In particular, we aim at giving *security guarantees*. When none can be given by our techniques, just as in other static analyses, it may still be that the analyzed program is in fact safe.

What we do not do. First, we do *not* infer cryptographic protocols from C code, i.e., we do not infer Figure 1 from Figure 2. This might have seemed the most reasonable route: when Figure 1 has been reconstructed, use your favorite cryptographic protocol verifier. We do not believe this is practical. First, recall that we usually only have the source code of *some* of the roles. Even is we had code for all roles, real implementations use many constructs that have no equivalent in input languages for cryptographic protocol verification tools. To take one realistic example, implementations of SSL [10] such as `ssh` use conditionals, bindings from conventional names such as `SSL_RSA_WITH_RC4_128_MD5` to algorithms (i.e., records containing function pointers, initialized to specific encryption, decryption, and secure hash functions), which are far from what current cryptographic protocol verification tools offer.

Second, we do *not* guarantee against any arbitrary attack on C code. Rather, our techniques are able to guarantee that there is no attack on a given piece of C code *in* a given trust model, stating who we trust, and *in* a given execution model, i.e., assuming a given, somewhat idealized semantics of C. In this semantics, writing beyond the bounds of an array never occurs. If we did not rely on such idealized semantics, essentially every static analysis would report possible security violations, most of them fake. It follows that buffer overflow attacks will not be considered in this paper. While buffer overflows are probably the most efficient technique of attack against real implementations (even not of cryptographic protocols; for hackers, see [11]), they can be and have already been analyzed [25, 24]. On programs immune to buffer overflows, we believe our idealized semantics to be a fair account of the semantics of C. Programs should be

checked against buffer overflows before our techniques are applied; we consider buffer overflows as an important but independent concern.

Outline After reviewing related work in Section 2, we introduce the subset of C we consider in Section 3, augmented with *trust assertions*—the cornerstone of our way of describing relations between in-memory values and Dolev-Yao-style messages. Its concrete semantics is described in Section 4, including trust assertions and the external trust model. We describe the associated abstract semantics in Section 5, which approximates C programs plus trust models as sets of Horn clauses, and describe our implementation in Section 6. We conclude in Section 7.

2 Related Work

Analyzing cryptographic protocols directly from source code seems to be fairly new. As far as we know, the only previous attempts in this direction are due to El Kadhi and Boury [16, 6], who propose a framework and algorithms to analyze leakage of confidential data in Java applets. They consider a model of cryptographic security based on the well-known Dolev-Yao model [8], just as we do. While we use Horn clauses as a uniform mechanism to abstract program semantics, intruder capabilities, and security properties alike, El Kadhi and Boury use a dedicated constraint format, and use a special constraint resolution calculus [16].

Analyzing cryptographic *programs* is not just a matter of analyzing cryptographic *protocols*. El Kadhi and Boury analyze Java applets (from bytecode, not source), and concentrate on a well-behaved subset of Java, where method calls are assumed to be inlined. Aliasing in Java is simpler to handle in Java than in C: the only aliases that may occur in Java arise from objects that can be accessed through different access paths (e.g., different variables); in C, more complex aliases may occur, such as through pointers to variables (see `&mesg1` for example in Figure 2). The StuPa tool [6] uses different static analysis frameworks to model the Dolev-Yao intruder and to analyze information flow through the analyzed applet; we use a uniform approach based on Horn clauses.

Finally, the security properties examined in [6] are models of leakage of sensitive data: sensitive data are those data stored in specially marked class fields, and are tracked through the program and the possible actions of the intruder; data can be leaked to the Dolev-Yao intruder, or more generally to untrusted classes in the programming environment. The aim of [6] is to detect whether some sensitive piece of data can be leaked to some untrusted class. Because we use Horn clauses, any property which can be expressed as a conjunction of atoms can be checked in our approach (as in [7]), in particular secrecy or leakage to some untrusted part of the environment.

Cryptographic Protocol Analysis. If we are just interested in cryptographic *protocols*, not programs, there are now many methods available: see [12] for an overview. One of the most successful models today is the *Dolev-Yao model* [8], where all communication channels are assumed to be rerouted to a unique *intruder*, who can encrypt and decrypt any message at will—provided it knows the inverse key in the case of decryption. Every message sent is just given to the intruder, and every message received is obtained from the intruder. This is the basis of many papers. One of the most relevant to our work is

Blanchet's model [3], where a single predicate `knows` (called `attacker` in op.cit.) is used to model what messages may be known to the intruder at any time. The abilities of the intruder are modeled by the following Horn clauses (in our notation):

$$\texttt{knows}(\texttt{nil}) \qquad \text{Intruder can} \qquad (1)$$

$$\texttt{knows}(\texttt{cons}(X,Y)) \Leftarrow \texttt{knows}(X), \texttt{knows}(Y) \qquad \text{build lists.} \qquad (2)$$

$$\texttt{knows}(X) \Leftarrow \texttt{knows}(\texttt{cons}(X,Y)) \qquad \text{Intruder can read} \qquad (3)$$

$$\texttt{knows}(Y) \Leftarrow \texttt{knows}(\texttt{cons}(X,Y)) \qquad \text{all elements of a list.} \qquad (4)$$

$$\texttt{knows}(\texttt{crypt}(X,Y)) \Leftarrow \texttt{knows}(X), \texttt{knows}(Y) \qquad \text{Intruder can encrypt.} \qquad (5)$$

$$\texttt{knows}(X) \Leftarrow \texttt{knows}(\texttt{crypt}(X,\texttt{pub}(Y))), \texttt{knows}(\texttt{prv}(Y)) \qquad \text{Intruder can decrypt} \qquad (6)$$

$$\texttt{knows}(X) \Leftarrow \texttt{knows}(\texttt{crypt}(X,\texttt{prv}(Y))), \texttt{knows}(\texttt{pub}(Y)) \qquad \text{provided he knows} \qquad (7)$$

$$\texttt{knows}(X) \Leftarrow \texttt{knows}(\texttt{crypt}(X,\texttt{sk}(Y,Z))), \texttt{knows}(\texttt{sk}(Y,Z)) \qquad \text{the inverse key.} \qquad (8)$$

$$\texttt{knows}(\texttt{pub}(X)) \qquad \text{Intruder knows public keys} \qquad (9)$$

We shall use a Prolog-like notation throughout: identifiers starting with capital letters, such as X or Y, are universally quantified variables; `nil` is a constant, `cons` and `crypt` are function symbols. Clause (5), for example, states that whenever the intruder knows (can deduce) X and Y, then he can deduce the result $\texttt{crypt}(X,Y)$ of the encryption of X with key Y. Clauses (6) through (8) state that he can deduce the plaintext X from the ciphertext $\texttt{crypt}(X,k)$ whenever he knows the inverse of key k; $\texttt{prv}(A)$ is meant to denote A's private key, $\texttt{pub}(A)$ is A's public key, and $\texttt{sk}(A,B)$ is some symmetric key to be used between agents A and B.

Most roles in cryptographic protocols are sequences of rules $M \Rightarrow M'$ (not to be confused either with implication $\Leftarrow$ or the arrows $\rightarrow$ shown in Figure 1), meaning that the role will wait for some (optional) message matching M, then (optionally) send M'. For example, role `A` in Figure 1 implements the rules $\Rightarrow \{N_{\texttt{A}}, \texttt{A}\}_{\texttt{pub(B)}}$ (step 1.) and $\{N_{\texttt{A}}, N_{\texttt{B}}\}_{\texttt{pub(A)}} \Rightarrow \{N_{\texttt{B}}\}_{\texttt{pub(B)}}$. This is easily compiled into Horn clauses. A rule $M \Rightarrow M'$ is simply compiled as the clause $\texttt{knows}(M') \Leftarrow \texttt{knows}(M)$, modulo some details. For example, and using Blanchet's trick of coding nonces as function symbols applied to parameters in context (e.g., $N_{\texttt{A}}$ will be coded as $\texttt{na}(B)$, in any session where `A` talks to some agent B), the role of `A` in Figure 1 may be coded as:

$$\texttt{knows}(\texttt{crypt}(\texttt{cons}(\texttt{na}(B), \texttt{cons}(\texttt{a}, \texttt{nil})), \texttt{pub}(B)) \qquad (10)$$

$$\texttt{knows}(\texttt{crypt}(Nb, \texttt{pub}(B))) \Leftarrow \texttt{knows}(\texttt{crypt}(\texttt{cons}(\texttt{na}(B), \texttt{cons}(Nb, \texttt{nil})), \texttt{pub}(\texttt{a}))) \qquad (11)$$

Finally, secrecy properties are encoded through negative clauses. For instance, given a specific agent `b`, that $N_{\texttt{A}}$ remains secret when `A` is talking to `b` will be coded as $\bot \Leftarrow \texttt{knows}(\texttt{na}(\texttt{b}))$. More complicated queries are possible, e.g., $\bot \Leftarrow \texttt{knows}(\texttt{na}(B))$, $\texttt{honest}(B)$ asks whether $N_{\texttt{A}}$ remains secret whatever agent `A` is really talking to, provided this agent is honest, for some definition of honesty (see [7] for example). We won't explore all the variants, and shall be content to know that we can use at least one. Note that the encodings above are upper approximations of the actual behavior of the protocol; this is needed in any case, as cryptographic protocol verification is undecidable [9, 1].

Program analysis. There is an even wider literature on static program analysis. Our main problem will be to infer what variables contain what kind of data. As these variables are mostly pointers to structures allocated on the heap, we have to do some kind of shape analysis. The prototypical such analysis is due to Sagiv *et al.* [22]. This analysis gives very precise information on the shape of objects stored in variables. It is also rather costly. A crucial observation in [22] is that store shapes are better understood as formulae. We shall adapt this idea to a much simplified memory model.

At the other end of the spectrum, Andersen's *points-to* analysis [2] gives a very rough approximation of what variables may point to what others, but can be computed extremely efficiently [14]. (See [15] for a survey of pointer analyses.) We shall design an analysis that is somewhere in between shape analysis and points-to analysis as far as precision is concerned: knowing whether variable `x` may point to `y` is not enough, e.g. we need to know that once lines 77–82 of Figure 2 have been executed, `cipher1` points to some allocated record containing `A`'s identity as `ip_id` and that the field `mesg1.msg.msg1.nonce` contains `A`'s nonce $N_{\mathtt{A}}$. (This is already non-trivial; we also need to know that this record actually denotes the term $\mathtt{crypt}(\mathtt{cons}(\mathtt{na}(B), \mathtt{cons}(\mathtt{a}, \mathtt{nil})), \mathtt{pub}(B))$ when seen from the cryptographic protocol viewpoint.) While this looks like what shape analysis does, our analysis will be flow-insensitive, just like standard points-to analyses.

One of our basic observations is that such pointer analyses can be described as generating Horn clauses describing points-to relations. Once this is done (Section 5), it will be easier to link in the cryptographic protocol aspects (e.g., to state that `cipher_1` *denotes* $\mathtt{crypt}(\mathtt{cons}(\mathtt{na}(B), \mathtt{cons}(\mathtt{a}, \mathtt{nil})), \mathtt{pub}(B)$, as stated above).

3 C Programs, and Trust Assertions

We assume that C programs are represented as a set of control flow graphs G_f, one for each function f. We assume that the source code of each function f is known—at least all those that we don't want to abstract away, such as communication and cryptographic primitives. We also consider a restricted subset of C, where casts are absent, and expressions are assumed to be well-typed. We do definitely consider pointers, and in particular pointer arithmetic, one of the major hassles of C semantics.

Formally, we define a *C program* as a map from function names f to triples (in_f, loc_f, G_f), where in_f is the list of f's formal parameters, loc_f is the list of f's local variables, and G_f is f's control flow graph. We assume that the node sets of each control flow graph G_f are pairwise disjoint.

A *control flow graph (CFG)* is a directed graph G with a distinguished *entry node* $I(G)$ and a distinguished *exit node* $O(G)$. Edges are labeled with *instructions*. The set of instructions in Figure 3 will be enough for our purposes, where x, y, z, ..., range over names of local variables, c ranges over integer and floating-point constants, f over function names, a over struct field names, and op ranges over primitive operations (arithmetic operations, bitwise logical operations, comparisons): The instructions $x = \&y[z]$ and $x = \&y \to a$ implement pointer arithmetic. The first adds the integer z to the pointer y, yielding x. The second adds the offset of field a to the pointer y. More complex instructions can be broken down to sequences of instructions as above. For ex-

$i \in Instr$	$::=$	$x = y$	variable copy
	$\mid$	$x = c$	storing constant c into x
	$\mid$	$x = f$	storing the address of function f into x
	$\mid$	$x = \&y$	storing the address of variable y into x
	$\mid$	$x = *y$	reading from a pointer
	$\mid$	$*x = y$	storing into a pointer
	$\mid$	$x = \&y[z]$	taking the address of entry z of array y
	$\mid$	$x = \&y \rightarrow a$	taking the address of field a in struct y
	$\mid$	$x = g(x_1, \ldots, x_n)$	calling function g
	$\mid$	$x = (*y)(x_1, \ldots, x_n)$	indirect call
	$\mid$	$x = op(x_1, \ldots, x_n)$	primitive call
	$\mid$	$?x == 0$	zero test
	$\mid$	$?x \,!= 0$	non zero test
	$\mid$	$\texttt{trust}\ A \Leftarrow A_1, \ldots, A_n$	trust assertion
$A \in Atom$	$::=$	$x\ \texttt{rec}\ t$	x is trusted to denote t
	$\mid$	$P(t)$	term t is trusted to obey property P

Fig. 3. Syntax of core C

ample, `msg->id_1[0] = id[0]` can be translated to the sequence of instructions $z = 0$, $x_1 = \&\texttt{id}[z]$, $x_2 = *x_1$, $x_3 = \&\texttt{msg} \rightarrow \texttt{id\_1}$, $x_4 = \&x_3[z]$, $*x_4 = x_2$. This of course presumes a given scheduling of elementary instructions; to verify output from a given C compiler, the same scheduling should be used. The test instructions $?x == 0$ and $?x \,!= 0$ do nothing, but can only be executed provided x is zero, resp. non-zero; they are used to represent `if` and `while` branches.

The only non-standard instruction above is the trust assertion. This is one of the main ingredients we use to link concrete C data with abstract Dolev-Yao style messages that they are meant to denote. A trust assertion $\texttt{trust}\ x\ \texttt{rec}\ t \Leftarrow x_1\ \texttt{rec}\ t_1, \ldots, x_n\ \texttt{rec}\ t_n$ relates the value of C variables (x, x_1, …, x_n) to terms (messages; t, t_1, …, t_n) that they are meant to denote. Intuitively, this states that the value of x denotes the term t, as soon as x_1 denotes t_1, and … and x_n denotes t_n. While atomic formulae $x\ \texttt{rec}\ t$ state that the value of x denotes t, other atomic formulae $P(t)$ (e.g., $\texttt{knows}(t)$, see Section 2) will be defined by the external trust model (see Section 4.2).

We have chosen to let the programmer state trust relations in the C source code using special comments; they are enclosed between `/* %` and `% */` in Figure 2. For example, the comment at line 20 translates to the trust statement $\texttt{trust cipher rec crypt}(M, K) \Leftarrow \texttt{msg rec}\ M, \texttt{key\_pub rec}\ K$, and states that, if `msg` points to a memory zone where message M is stored, and if `key_pub` points to some zone containing K, then `cipher` will be filled with the ciphertext $\texttt{crypt}(M, K)$; in other words, `encrypt_mesg` computes the encryption of `*msg` using key `*key_pub` and stores it into `*cipher`.

We *do* require trust assertions. Otherwise, there is no way to recognize statically that the call to `BN_mod_exp` on line 18 actually computes modular exponentiation on arbitrary sized integers (“bignums”, of type `BIGNUM`), and much less that this encrypts its second argument `plain` using the key given as third and fourth arguments `key_pub`,

`key_mod`, storing the result into the first argument `cipher`. In fact, there is no way to even define a sensible map from bignums to terms that would give their purported meaning in the Dolev-Yao model.

We need such trust assertions for two distinct purposes. The first is to describe the *effect of functions in the API* in terms of the Dolev-Yao model; in particular, to abstract away the effect of low-level cryptographic functions that are used in the analyzed program (e.g., the OpenSSL crypto lib), or of the standard C library (see the comment on line 47, which abstracts away the behavior of the `write` function, stating that any message sent to `write` through the buffer `buf` will be known to the Dolev-Yao intruder). The second purpose of trust assertions is to state *initial security assumptions*: see the comment on line 67, which states that the array `ip_id` is trusted to contain A's identity, initially. (The notation `CTX(Agent(A))` refers to A's identity as given in a global context `CTX`; we shall not describe this in detail here.)

4 Concrete Semantics

We first describe the memory layout. Let $Addr$ be a denumerable set of so-called *addresses*. A *store* $\mu \in Store$ is any map from adresses to zones. Intuitively, addresses are those memory addresses returned by memory allocation functions, e.g., `malloc`. (As a technical aside, we assume that declaring a local C variable `x` in a C function has the effect of allocating some memory, too, for holding `x`'s value, at an address that is usually written `&x` in C. We do this because, contrarily to, say, Java, you can take the address of variables in C, and modify them through pointer operations.) *Zones* describe the layout of data stored at given addresses, and are described by the following grammar:

$z ::=$	`code` f	code for function f
$\mid$	`int` n	integer n
$\mid$	`float` x	floating-point value x
$\mid$	`ptr` ℓ	pointer, pointing to location ℓ
$\mid$	`struct` $\{lab_1 = z_1, \ldots, lab_n = z_n\}$	structure, with labels lab_i, $1 \leq i \leq n$
$\mid$	`array`$(z_1, \ldots, z_n)$	array of n sub-zones

Let $Zone$ be the set of all zones. *Locations* ℓ, as used in pointers, are strings $a.sel_1.\ldots.sel_k$, where $a \in Addr$, and sel_j, $1 \leq j \leq k$, are *selectors*, namely either *labels* $lab \in Lab$ or integers $n \in \mathbb{Z}$. For example, in Figure 4, if a is the address of `x`, a`.data.t.2` is the location of the cell shown in red.

Let Loc be the set of all locations. Let $Store = Addr \to Zone$ be the set of all stores. Any store μ extends in a unique way to a map $\hat{\mu}$ from *locations* to zones: if $a \in Addr$, then $\hat{\mu}(a) = \mu(a)$; $\hat{\mu}(\ell.lab_i) = z_i$ provided $\hat{\mu}(\ell)$ is defined and of the form `struct` $\{lab_1 = z_1, \ldots, lab_n = z_n\}$, $1 \leq i \leq n$; and $\hat{\mu}(\ell.j) = z_j$ provided $\hat{\mu}(\ell)$ is defined and of the form `array`$(z_1, \ldots, z_n)$, $1 \leq j \leq n$. E.g., `x.data` has a location, namely a`.data`, mapped by $\hat{\mu}$ to the zone shown in Figure 4, top right.

Given a C program mapping each function f to (in_f, out_f, G_f) (Section 3), we define its semantics as a transition system. *Transitions* (inside G_f) are defined by judgments $q, \rho, \mu \xrightarrow{i} q', \rho', \mu'$, one for each edge $q \xrightarrow{i} q'$ in G_f, where ρ and ρ' are *en-*

```
1  typedef enum  { RSA, DES } Myenum;
2
3  typedef struct Mystruct1 {
4    int m;
5    char t[4];
6    void *next;
7  } Mystruct1;
8
9  typedef struct Mystruct2 {
10   Myenum    cryptfun;
11   Mystruct1 data;
12 } Mystruct2;
```

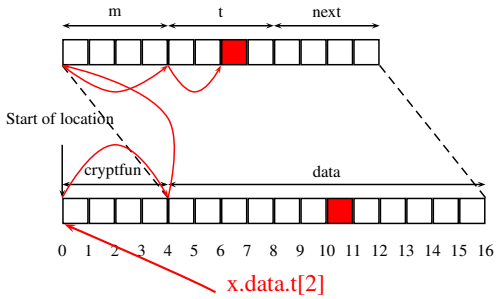

Fig. 4. Sample memory zone

$$
\begin{array}{lll}
q, \rho, \mu & \xrightarrow{x=y} & q', \rho, \mu[\rho(x) \mapsto \mu(\rho(y))] \\
q, \rho, \mu & \xrightarrow{x=c} & q', \rho, \mu[\rho(x) \mapsto c] \\
q, \rho, \mu & \xrightarrow{x=f} & q', \rho, \mu[\rho(x) \mapsto a] \quad \text{if } \mu(a) = \texttt{code}\ f \text{ for some } a \in \text{dom}\ \mu \\
q, \rho, \mu & \xrightarrow{x=\&y} & q', \rho, \mu[\rho(x) \mapsto \texttt{ptr}(\rho(y))] \\
q, \rho, \mu & \xrightarrow{x=*y} & q', \rho, \mu[\rho(x) \mapsto \hat{\mu}(\ell)] \quad \text{if } \mu(\rho(y)) = \texttt{ptr}\ \ell \text{ for some } \ell \in Loc \\
q, \rho, \mu & \xrightarrow{*x=y} & q', \rho, \mu[\ell \mapsto \mu(\rho(y))] \quad \text{if } \mu(\rho(x)) = \texttt{ptr}\ \ell \text{ for some } \ell \in Loc \\
q, \rho, \mu & \xrightarrow{x=\&y[z]} & q', \rho, \mu[\rho(x) \mapsto \texttt{ptr}\ (\ell.(j+1))] \quad \text{if } \rho(y) = \texttt{ptr}\ \ell, \\
 & & \mu(\ell) = \texttt{array}\ (z_1, \ldots, z_n), \text{and } \mu(\rho(z)) = \texttt{int}\ j, 0 \leq j \leq n \\
q, \rho, \mu & \xrightarrow{x=\&y \to a} & q', \rho, \mu[\rho(x) \mapsto \texttt{ptr}\ (\ell.a)] \quad \text{if } \rho(y) = \texttt{ptr}\ \ell, \\
 & & \text{and } \mu(\ell) = \texttt{struct}\ \{\ldots, a = z, \ldots\} \\
q, \rho, \mu & \xrightarrow{x=g(x_1,\ldots,x_n)} & q', \rho', \mu' \text{iff: see main text} \\
q, \rho, \mu & \xrightarrow{x=(*y)(x_1,\ldots,x_n)} & q', \rho', \mu' \text{iff: see main text} \\
q, \rho, \mu & \xrightarrow{x=op(x_1,\ldots,x_n)} & q', \rho, \mu[\rho(x) \mapsto \widehat{op}(\mu(\rho(x_1)), \ldots, \mu(\rho(x_n)))] \\
q, \rho, \mu & \xrightarrow{?x==0} & q', \rho, \mu \text{ if } \mu(\rho(x)) = \texttt{int}\ 0 \\
q, \rho, \mu & \xrightarrow{?x \models 0} & q', \rho, \mu \text{ if } \mu(\rho(x)) \neq \texttt{int}\ 0 \\
q, \rho, \mu & \xrightarrow{\texttt{trust}\ A \Leftarrow A_1,\ldots,A_n} & q', \rho, \mu
\end{array}
$$

Fig. 5. Concrete semantics

vironments mapping variables to their addresses. This is shown in Figure 5. The notation $\rho[x \mapsto z]$ denotes the map sending x to z, and every other $y \in \mathrm{dom}\ \rho$ to $\rho(y)$. Similarly for $\mu[a \mapsto z]$, where $a \in Addr$ and $z \in Zone$. To model writing into arbitrary locations ℓ, not just addresses, we extend this notation by letting $\mu[\ell.lab_i \mapsto z] = \mu[\ell \mapsto \mathtt{struct}\ \{lab_1 = z_1, \ldots, lab_i = z, \ldots, lab_n = z_n\}]$ whenever $\hat{\mu}(\ell) = \mathtt{struct}\ \{lab_1 = z_1, \ldots, lab_n = z_n\}$, and $\mu[\ell.i \mapsto z] = \mu[\ell \mapsto \mathtt{array}(z_1, \ldots, z, \ldots, z_n)]$ with z at position i, whenever $\hat{\mu}(\ell) = \mathtt{array}(z_1, \ldots, z_n)$ and $1 \leq i \leq n$. (This is then partially defined.) This extension is used in the semantics of $*x = y$.

The rules deserve some explanation. E.g., the semantics of $x = y$ consists in fetching the address $\rho(y)$ at which the contents of variable y is stored in memory (the address usually referred to as $\&y$ in C code), and copying it into the address $\rho(x)$ at which x is stored. In effect, $x = y$ in C really means $*(\&x) = *(\&y)$. To lighten up the semantics, we agree that mentioning any expression entails that it is defined. In other words, the mere fact that we are writing $\mu[\rho(x) \mapsto \mu(\rho(y))]$ in the semantics of $*x = y$ really means that we must first check that $y \in \mathrm{dom}\ \rho$ and $\rho(y) \in \mathrm{dom}\ \mu$ and $x \in \mathrm{dom}\ \rho$ and $\rho(x) \in \mathrm{dom}\ \mu$.

The semantics of $x = f$ presumes that there is an address at which the code for the function f is stored; whichever such a is then stored into x. This is taken care of by starting the program in a store that contains such a mapping from addresses to code fragments (and similarly contains storage for string constants).

Most other entries are self-explanatory. In the semantics of primitive calls $x = op(x_1, \ldots, x_n)$, we assume that the semantic $\widehat{op}$ of op is given separately.

Figure 5 leaves out the semantics for function calls $x = g(x_1, \ldots, x_n)$. We let $q, \rho, \mu \xrightarrow{x=g(x_1,\ldots,x_n)} q', \rho', \mu'$ if and only if $I(G_g), \rho_I, \mu_I \longrightarrow^* O(G_g), \rho_O, \mu_O$, where μ_I is obtained by allocating one new structure for formal parameters, one for local variables, and one for the return value (i.e., $\mu_I = \mu[a_{in} \mapsto \mathtt{struct}\ \{\&\mathtt{x}_1 = z_1, \ldots, \&\mathtt{x}_n = z_n\}, a_{loc} \mapsto \mathtt{struct}\ \{\&\mathtt{y}_1 = z'_1, \ldots, \&\mathtt{y}_m = z'_m\}, a_{ret} \mapsto \mathtt{struct}\ \{\&\mathtt{return} = z\}]$, where a_{in}, a_{loc}, and a_{ret} are distinct fresh addresses, $\mathtt{x}_1$, ..., $\mathtt{x}_n$ are the formal parameters, $\mathtt{y}_1, \ldots, \mathtt{y}_m$ are the local variables, and $z_1, \ldots, z_n, z'_1, \ldots, z'_m, z$ are appropriate zones, considering the types of variables), where ρ_I maps each $\mathtt{x}_i$ to $a_{in}.\&\mathtt{x}_i$, each $\mathtt{y}_j$ to $a_{loc}.\&\mathtt{y}_j$, and the fresh variable `return` (used to actually return a value from g) to $a_{ret}.\&\mathtt{return}$, where μ' is μ_O restricted to $\mathrm{dom}\ \mu_O \setminus \{a_{in}, a_{loc}, a_{ret}\}$, and where $\rho' = \rho[x \mapsto \hat{\mu}_O(a_{ret}.\&\mathtt{return})]$. Note that we encode `return` v as an assignment $\mathtt{return} = v$. We deal with indirect calls $x = (*y)(x_1, \ldots, x_n)$ similarly: the only change is that we check that $\mu(\rho(y)) = \mathtt{code}\ g$ for some function g.

At the level of zones and pointers which we consider in this section, trust assertions just do nothing. We shall extend this semantics in the next section to properly handle trust assertions.

4.1 Semantics of Trust Assertions

The purpose of trust assertions is to define the denotation of concrete C data as Dolev-Yao style messages. A given piece of C data z may have one such denotation, or zero

(e.g., if z just denotes, say, some index into a table, with no significance, security-wise), or several (e.g., if only for cardinality reasons, there are infinitely many terms but only finitely many 128-bit integers; concretely, even cryptographic hash functions have collisions.) Therefore we model the semantics of trust assertions as generating a *trust relation* $\mathcal{R}$—a binary relation between C values and ground first-order terms—together with a *trust base* $\mathcal{B}$—a set of ground first-order atoms. Let $Term_0$ be the set of all ground terms, $Atom_0$ be the set of ground atoms, and Val the set of C values, so a trust relation $\mathcal{R}$ is a subset of $ValTerm_0$, and a trust base $\mathcal{B}$ is a subset of $Atom_0$.

A difficulty here is in defining what a C value is. Typically, an integer n should be a C value, and two integers should be equal as values if and only if they are equal as integers. In general, it is natural to think that zones should somehow represent C values. This implies that a zone of the form $\texttt{ptr}(\ell)$, i.e., a pointer, should also represent a C value. This is needed: in Figure 2, we really want to understand the *pointer* `cipher1` (l.55) as denoting a message. But only the contents of the zone pointed to by `cipher1` should be relevant, not the location ℓ.

The irrelevance of ℓ is best handled through the notion of *bisimilarity*, which we define by imitation from [18]. A *bisimulation* is a binary relation $\sim$ on $LocStore$, together with a binary relation (again written $\sim$) on $ZoneStore$, such that:

- if $(\ell, \mu) \sim (\ell', \mu')$ then either $\ell \notin \text{dom } \hat{\mu}$ and $\ell' \notin \text{dom } \hat{\mu}'$, or $\ell \in \text{dom } \hat{\mu}$, $\ell' \in \text{dom } \hat{\mu}'$ and $(\hat{\mu}(\ell), \mu) \sim (\hat{\mu}'(\ell'), \mu')$;
- if $(z, \mu) \sim (z', \mu')$ then either $z = z'$ is of the form $\texttt{code } f$ or $\texttt{int } n$ or $\texttt{float } x$; or z is of the form $\texttt{ptr}(\ell)$, z' is of the form $\texttt{ptr}(\ell')$, and $(\ell, \mu) \sim (\ell', \mu')$; or $z = \texttt{struct } \{lab_1 = z_1, \ldots, lab_n = z_n\}$, $z' = \texttt{struct } \{lab_1 = z'_1, \ldots, lab_n = z'_n\}$, and $(z_i, \mu) \sim (z'_i, \mu')$ for every i, $1 \leq i \leq n$; or $z = \texttt{array}(z_1, \ldots, z_n)$, $z' = \texttt{array}(z'_1, \ldots, z'_n)$, and $(z_i, \mu) \sim (z'_i, \mu')$ for every i, $1 \leq i \leq n$.

Let $\cong$ (*bisimilarity*) be the largest bisimulation, with respect to inclusion of binary relations. A pair (ℓ, μ) of a location and a store μ describes a rooted graph in memory, whose root is ℓ, and whose edges are given by following pointers. Then, each rooted graph can be unfolded to yield an infinite tree. It is standard that bisimilarity relates (ℓ, μ) to (ℓ', μ') if and only if the unfolded infinite trees corresponding to each graph are isomorphic. It is natural to equate C values with such unfolded infinite trees (up to isomorphism), hence to pairs (ℓ, μ) up to bisimilarity: we therefore let Val be the quotient $(LocStore)/\cong$. We let $[\ell, \mu]$ be the equivalence class of (ℓ, μ) under $\cong$.

We need to modify our semantics of C so that it takes into account trust assertions. For each instruction i in function f, except trust assertions, define the new transition relation $q, \rho, \mu, \mathcal{R}, \mathcal{B} \xrightarrow{i} q', \rho', \mu', \mathcal{R}', \mathcal{B}'$ (which now deals additionally with $\mathcal{R}, \mathcal{R}' \subseteq ValTerm_0$ and $\mathcal{B}, \mathcal{B}' \subseteq Atom_0$) by: $q, \rho, \mu \xrightarrow{i} q', \rho', \mu'$ as defined in Figure 5, and $\mathcal{R}' = \mathcal{R}$ and $\mathcal{B}' = \mathcal{B}$. That is, ordinary C instructions do not modify the trust relation or the trust base, and otherwise behave in the standard way.

When i is the trust assertion $\texttt{trust } A \Leftarrow A_1, \ldots, A_n$, do the following. First, fix a set of definite clauses $\mathcal{M}$. (For now, just imagine $\mathcal{M}$ is empty. $\mathcal{M}$ is the external trust model, which we shall explain in Section 4.2.) The trust assertion simply adds to $\mathcal{R}$ and $\mathcal{B}$ all the new consequences deducible from the current $\mathcal{R}$ and $\mathcal{B}$, using the clauses $A \Leftarrow A_1, \ldots, A_n$ and the clauses in $\mathcal{M}$.

Formally, given any atom A', say that $\rho, \mu, \mathcal{R}, \mathcal{B} \models A'$ if and only if A' is of the form x `rec` t and $([\rho(x), \mu], t) \in \mathcal{R}$, or A' is of the form $P(t)$ and $P(t) \in \mathcal{B}$. For each definite clause C of the form $A \Leftarrow A_1, \ldots, A_n$, let $T_C^{\rho,\mu}(\mathcal{R}, \mathcal{B})$ be the smallest pair $(\mathcal{R}', \mathcal{B}')$ in the componentwise subset ordering such that, for every substitution σ such that $A\sigma$, $A_1\sigma$, ..., $A_n\sigma$ are ground and such that $\rho, \mu, \mathcal{R}, \mathcal{B} \models A_i\sigma$ for each i, $1 \leq i \leq n$, then $\rho, \mu, \mathcal{R}', \mathcal{B}' \models A\sigma$. For every set $\mathcal{M}$ of definite clauses, let $T_{\mathcal{M}}^{\rho,\mu}(\mathcal{R}, \mathcal{B})$ be the sup over all $C \in \mathcal{M}$ of $T_C^{\rho,\mu}(\mathcal{R}, \mathcal{B})$. (This is the familiar T_P operator of Prolog semantics.) Let lfp $T_{\mathcal{M}}^{\rho,\mu}(\mathcal{R}, \mathcal{B})$ be the least fixpoint of $T_{\mathcal{M}}^{\rho,\mu}$ above $(\mathcal{R}, \mathcal{B})$.

Then, when i is the trust assertion `trust` $A \Leftarrow A_1, \ldots, A_n$, we define $q, \rho, \mu, \mathcal{R}, \mathcal{B} \xrightarrow{i} q', \rho', \mu', \mathcal{R}', \mathcal{B}'$ if and only if $q \xrightarrow{i} q'$ is an edge of G_f, $\rho = \rho'$, $\mu = \mu'$ (so trust statements behave as no-operations in the standard C semantics), and

$$(\mathcal{R}', \mathcal{B}') = \text{lfp } T_{\mathcal{M}}^{\rho,\mu}(T_{A \Leftarrow A_1, \ldots, A_n}^{\rho,\mu}(\mathcal{R}, \mathcal{B})) \tag{12}$$

To simplify things a bit, imagine that $\mathcal{M}$ is empty. So $(\mathcal{R}', \mathcal{B}') = T_{A \Leftarrow A_1, \ldots, A_n}^{\rho,\mu}(\mathcal{R}, \mathcal{B})$. In particular, if i is `trust` x `rec` $t \Leftarrow x_1$ `rec` $t_1, \ldots, x_n$ `rec` t_n, then $\mathcal{B}' = \mathcal{B}$ and

$$\mathcal{R}' = \mathcal{R} \cup \{([\rho(x), \mu], t\sigma) | ([\rho(x_1), \mu], t_1\sigma) \in \mathcal{R} \text{ and } \ldots \text{ and } ([\rho(x_n), \mu], t_n\sigma)\}$$

where σ ranges over all substitutions such that $t\sigma$, $t_1\sigma$, ..., $t_n\sigma$ are ground terms. In other words, remembering that the C value of a variable y is $[\rho(y), \mu]$, this states that we trust that the C value of x should denote any message that is a ground instance $t\sigma$ of t, as soon as the C value of x_1 denotes $t_1\sigma$ and ... and the C value of x_n denotes $t_n\sigma$.

Trust assertions are given as special C comments. E.g., the trust assertion on line 20 of Figure 2 states that `encrypt_mesg` really encrypts: we *trust* that, at the end of `encrypt_mesg`, `cipher` points to the encryption `crypt(M,K)` of the plaintext pointed to by `msg` with key pointed to by `key_pub`. Line 47 states that we trust `write` to make anything the contents of the buffer `buf` available to the Dolev-Yao intruder.

4.2 The External Trust Model

As we have already said in the introduction, programs such as SSL or the one of Figure 2 cannot be analyzed in isolation. We have to describe how the outside world, i.e., the other programs with which the analyzed programs communicates, behaves. This is in particular needed because the canonical trust statement for `write` is to declare that `knows`(t) holds whenever its input argument is trusted to denote message t; and the canonical trust statement for `read` is to declare that the contents of the buffer given as input will denote any message t such that `knows`(t). (This is the standard assumption in the Dolev-Yao model, that all communication is to and from an all powerful intruder.)

Concretely, in particular, we have to describe the semantics of the `knows` predicate, meant to represent all messages that a Dolev-Yao intruder may build. We do this by providing clauses such as (1)–(9), but also such as (10)–(11) to describe an abstract view of the roles of *honest* principals participating in the same or other protocols, and which are believed to share secrets with the analyzed program. Such clauses can be built from spi-calculus terms for example, following either Blanchet's [3] or Nielson et al.'s [20] approaches. (We tend to prefer the latter for pragmatic reasons: the output clauses are always in the decidable class $\mathcal{H}_1$; more detail later.)

In any case, we parameterize our analysis by an *external trust model*, which is just a set $\mathcal{M}$ of definite Horn clauses given in advance. The concrete semantics of programs is defined relatively to $\mathcal{M}$, see (12). The effect of applying $\mathrm{lfp}\, T_{\mathcal{M}}^{\rho,\mu}$ is to close all facts in $\mathcal{R}$ and $\mathcal{B}$ under any finite number of applications of intruder and honest principal rules from the outside world.

5 Abstract Semantics

Let $AbsStore$ and $AbsEnv$ be the set of abstract *stores* and abstract *environments*. It does not matter much really how we represent these. Any static analysis of C code that is able of handling pointer aliases would probably do the job. We choose one that matches the simplicity of points-to analysis as much as we can. We associate an *abstract zone* with each variable (local or global), and with each memory allocation site, in the form of a fresh constant, taken from a finite set. An *atomic formula* $\mathtt{p}(t, t')$, where t is a term, states that t is a location that may point to zone t'. The abstract semantics is then given as Horn clauses stating what new possible values may be found at what abstract zones.

Following the spirit of points-to analyses, we only include gen equations, and no $kill$; this considerably simplifies the abstract semantics. We define the abstract semantics $[\![i]\!]^{\#}\rho^{\#}$ of instruction i in the abstract environment $\rho^{\#}$, mapping variable names to abstract zones, a.k.a., constants, as sets of Horn clauses. The semantics of a function, resp. a whole program, is just the union of the semantics of all instructions in the given control flow graphs.

$$
\begin{aligned}
[\![x = y]\!]^{\#}\rho^{\#} &= \{\mathtt{p}(\mathtt{c}_x, X) \Leftarrow \mathtt{p}(\mathtt{c}_y, X)\} \text{ where } \mathtt{c}_x = \rho^{\#}(x), \mathtt{c}_y = \rho^{\#}(y)\\
[\![x = c]\!]^{\#}\rho^{\#} &= \{\mathtt{p}(\mathtt{c}_x, c)\}\\
[\![x = f]\!]^{\#}\rho^{\#} &= \{\mathtt{p}(\mathtt{c}_x, \mathtt{code}(f))\}\\
[\![x = \&y]\!]^{\#}\rho^{\#} &= \{\mathtt{p}(\mathtt{c}_x, \mathtt{ptr}(\mathtt{c}_y))\}\\
[\![x = *y]\!]^{\#}\rho^{\#} &= \{\mathtt{p}(\mathtt{c}_x, X) \Leftarrow \mathtt{p}(\mathtt{c}_y, \mathtt{ptr}\ Y), \mathtt{p}(Y, X)\}\\
[\![*x = y]\!]^{\#}\rho^{\#} &= \{\mathtt{p}(X, Y) \Leftarrow \mathtt{p}(\mathtt{c}_x, \mathtt{ptr}\ X), \mathtt{p}(\mathtt{c}_y, Y)\}\\
[\![x = \&y[z]]\!]^{\#}\rho^{\#} &= \{\mathtt{p}(\mathtt{c}_x, \mathtt{ptr}(X_{j+1})) \Leftarrow \mathtt{p}(\mathtt{c}_y, \mathtt{ptr}(Y)),\\
&\qquad \mathtt{p}(Y, \mathtt{array}(X_1, \ldots, X_n, X_{n+1}))\\
&\qquad |j \in [\![z]\!]^{\#}_{int}\} \text{ if } y \text{ is an expanded array}\\
&\quad \{\mathtt{p}(\mathtt{c}_x, \mathtt{ptr}(X)) \Leftarrow \mathtt{p}(\mathtt{c}_y, \mathtt{ptr}(X))\} \text{ if } y \text{ is a shrunk array}\\
[\![x = \&y \to a]\!]^{\#}\rho^{\#} &= \{\mathtt{p}(\mathtt{c}_x, \mathtt{ptr}(Z)) \Leftarrow \mathtt{p}(\mathtt{c}_y, \mathtt{ptr}(Y)),\\
&\qquad \mathtt{p}(Y, \mathtt{struct}\ \{\ldots, a = Z, \ldots\})\\
[\![\mathtt{trust}\ A \Leftarrow A_1, \ldots, A_n]\!]^{\#}\rho^{\#} &= \{(A \Leftarrow A_1, \ldots, A_n)\rho^{\sharp}\}
\end{aligned}
$$

Fig. 6. Some abstract semantic equations

In Figure 6, we use the convention that $\mathtt{c}_x = \rho^{\sharp}(x)$, $\mathtt{c}_y = \rho^{\sharp}(y)$. This is recalled in the first rule, and omitted in later rules. In the second and third clauses, we assume that constants c and functions f can also serve as term constants when used in clauses. For the sake of precision, integer constants thus recorded are not used in computing

array indices (instructions $x = \&y[z]$); rather an auxiliary analysis is run, based on a given integral abstract domain, yielding a set of possible integer values $[\![z]\!]^{\#}_{int}$ for the variable z: we follow here [4, 5] in that we distinguish *expanded array cells* (arrays whose length n is completely known, and are handled much like collections of separate global variables) and *shrunk array cells* (arrays thought of as one single abstract cell). In $x = \&y[z]$ and $x = \&y \to a$, we assume the types of all variables to be completely known; this determines the right form of term $\texttt{array}(X_1, \ldots, X_n, X_{n+1})$ or $\texttt{struct}\ \{\ldots, a = Z, \ldots\}$ in the bodies of clauses (in the first case, it yields the length n of the expanded array; the fictitious element X_{n+1} is added so as to cope with the fact that $\&y[n]$ is legal C code although $y[n]$ is not a valid element; while $\texttt{struct}\ \{\ldots, a = Z, \ldots\}$ is syntactic sugar for some term where field labels have been ordered in some way, and Z denotes the entry corresponding to the a label).

The abstract semantics for function calls is implemented as in [14]. We leave its formal expression as a (tedious) exercise. Intuitively, calling the known function g by $x = g(x_1, \ldots, x_n)$ works as though the actual parameters were copied, using run-of-the-mill assignments, into global locations $g.\texttt{in.x}_1$, ..., $g.\texttt{in.x}_n$. A local variable struct $g.\texttt{loc}$ is also used to hold local variables, another $g.\texttt{ret}$ to hold the return value, and the assignment $x = g.\texttt{ret.\&return}$ is simulated. (This matches the names of structs used in the concrete semantics of function calls, see Section 4.) Additional standard optimizations are added, e.g., keeping track of effective call sites when returning from functions to avoid spurious, fake control flow.

One advantage of this points-to-like abstract semantics is that the semantic of trust assertions is as simple as it can be: just add the trust assertion as a clause, replacing all C variables x by their location $\rho^{\#}(x)$. Reading $\rho^{\#}$ as a substitution, this means applying the substitution $\rho^{\#}$ to the entire clause $A \Leftarrow A_1, \ldots, A_n$.

This abstract semantics is of course rather coarse. One may improve somehow the precision of the analysis by renaming local variables after each assignment, in effect using variants of the SSA form.

5.1 Checking Abstract Properties

Once the abstract semantics of the program has been computed, as a set of Horn clauses, add the external trust model $\mathcal{M}$, which specifies all intruder capabilities, as well as behaviors that we trust other honest participants may have on the network. This yields a set S of Horn clauses.

Confidentiality. Assume we want to check that the value of variable $\texttt{x}$ is always secret. This can be checked by verifying that S plus the goal clause $\bot \Leftarrow \texttt{knows}(Y)$, $\texttt{p}(\texttt{c}_\texttt{x}, X), X\ \texttt{rec}\ Y$ is satisfiable. (That security boils down to satisfiability of clause sets, and more precisely to the existence of a model, was first noticed explicitly by Selinger [23].) Indeed, our abstract semantics is an upper approximation of all correct behaviors of our C program in the current trust model. If there is an attack, then there will be a closed term t (denoting the bit-level value of variable $\texttt{x}$, in the sense of Section 4.1) such that $\texttt{p}(\texttt{c}_\texttt{x}, t)$ holds, and a closed term u (denoting the message that we think is one possible reading of the value t) such that $t\ \texttt{rec}\ u$, and which the intruder can discover, namely $\texttt{knows}(u)$.

Conformance. We may also check that specific variables may contain values of a specific form, say values trusted to denote messages matching a given open term t. We can test this by checking whether S plus the goal $\bot \Leftarrow \mathtt{p}(\mathtt{c_X}, X), X \ \mathtt{rec}\ t$ is unsatisfiable, where X is not free in t. This can be used to detect bugs, e.g., when one variable name was mistyped.

Checking satisfiability of sets of Horn clauses is in general undecidable. However we notice that all the clauses provided in the abstract semantics are in the decidable class $\mathcal{H}_1$, and in fact in the polynomial-time decidable subclass $\mathcal{H}_2$ [20]. We prefer clauses in the external trust model $\mathcal{M}$, accordingly, to fall in $\mathcal{H}_1$, too. Otherwise, we can approximate them as follows. We assume without loss of generality that only monadic predicate symbols occur; e.g., $\mathtt{p}(u, v)$ is encoded as $\mathtt{p}(c(u, v))$. Given a Horn clause $P(t) \Leftarrow body$, first linearize t by making copies of each variable, copying the corresponding parts of the body as needed. E.g., transform $P(f(X, X)) \Leftarrow Q(X), R(Y)$ into $P(f(X_1, X_2)) \Leftarrow Q(X_1), Q(X_2), R(Y)$. Then, if t is of the form $f(t_1, \ldots, t_n)$ where some t_i at least is not a variable, replace $P(t) \Leftarrow body$ by the clauses $P(f(X_1, \ldots, X_n)) \Leftarrow Q_1(X_1), \ldots, Q_n(X_n)$ and $Q_i(t_i) \Leftarrow body$, $1 \leq i \leq n$, for fresh predicates $Q_1, \ldots, Q_n$, and repeat the process on the latter clauses. This yields clauses in $\mathcal{H}_1$, and is guaranteed to have a least Herbrand model that is an upper approximation of that of the original clause set. (In effect, this defines a set-constraint based typing discipline.) Since most clauses arise from the abstract semantics of the program, we do not lose much precision by doing this second abstraction step. Moreover, past experience in the verification of cryptographic protocols demonstrates that this does not throw away any essential information [13]. We have yet to evaluate whether this abstraction to $\mathcal{H}_1$ remains practical in the context of program verification.

6 Implementation

We have implemented this in the CSur project [21].

In a first phase, a specific *compiler* `csur_cc` reads, manages and generates a control-flow graph for each function of the program. All control flow graphs are stored in a unique table. Starting from the `main` function, the second phase uses a hybrid analyzer (computing abstract memory zones and collecting all Horn clauses for all program points) and performs function calls using the above table. Our tool follows the Compile-Link-Analysis technique of Heintze and Tardieu [14].

For each function, a control flow graph is generated and the compiler collects types for each variable of programs. For all types, the physical representation is also computed (using low level representations, for example field offsets of structures are computed as seen as Figure 4). Finally a linker merges all control flow graphs and types into a unique table. In the same way a library manager `csur_ar` (used just like `ar`) was implemented to help collect control flow graphs as single archives. These tools are defined as `gcc` front-ends to collect compilation options of source file.

The `csur_cc` compiler also collects trust assertions as it analyzes C code, and spits out a collection of Horn clauses which are then fed to an $\mathcal{H}_1$ solver—currently SPASS [27, 26] or the first author's prototype `h1` prover. The fact that most clauses are in $\mathcal{H}_2$, a polynomial class, is a treat: despite several optimizations meant to decrease the number

of generated clauses, a running 229 line implementation (excluding included files) of `A`'s role in the Needham-Schroeder protocol results in a set of 459 clauses.

7 Conclusion

This paper is one of the first attempts at analyzing actual implementations of cryptographic protocols. Our aim is not to detect subtle buffer overflows, which are better handled by other techniques, but to detect the same kind of bugs that cryptographic protocols are fraught with, only on actual implementations. We must say that combining the intricacies of analyzing C code with cryptographic protocol verification is still a challenge. This can be seen from the fact that our abstract semantics for C is still fairly imprecise. First experiments however show that this is enough on the small examples we tested. Despite the shortcomings that our approach clearly still has, and which will be the subject of future work, we would like to stress the importance of *trust assertions* as a logical way of linking the in-memory model of values to the abstract Dolev-Yao model of messages; and the fact that compiling to Horn clauses is an effective, yet simple way of checking complex trust and security properties.

References

1. R. M. Amadio and W. Charatonik. On name generation and set-based analysis in the Dolev-Yao model. In *13th International Conference on Concurrency Theory CONCUR'02*, volume 2421 of *Lecture Notes in Computer Science*, pages 499–514. Springer-Verlag, 2002.
2. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994. (DIKU report 94/19).
3. B. Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.
4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter. In T. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes on Computer Science*, pages 85–108. Springer Verlag, Dec. 2002.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, June 2003. ACM.
6. P. Boury and N. Elkhadi. Static Analysis of Java Cryptographic Applets. In *ECOOP'2001 Workshop on Formal Techniques for Java Programs*. Fern Universität Hagen, 2001.
7. H. Comon-Lundh and V. Cortier. Security properties: Two agents are sufficient. In *Proc. 12th European Symposium on Programming (ESOP'2003)*, pages 99–113, Warsaw, Poland, Apr. 2003. Springer-Verlag LNCS 2618.
8. D. Dolev and A. C. Yao. On security of public key protocols. *IEEE trans. on Information Theory*, IT-30:198–208, 1983.
9. N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Undecidability of bounded security protocols. In N. Heintze and E. Clarke, editors, *Proceedings of the Workshop on Formal Methods and Security Protocols — FMSP, Trento, Italy*, July 1999.

10. A. Freier, P. Karlton, and P. Kocher. The SSL protocol. Version 3.0., 1996. http://home.netscape.com/eng/ssl3/.
11. O. Gay. Exploitation avancée de buffer overflows. Technical report, LASEC, Ecole Polytechnique Fédérale de Lausanne, June 2002. http://diwww.epfl.ch/~ogay/advbof/advbof.pdf.
12. J. Goubault-Larrecq, editor. *Special Issue on Models and Methods for Cryptographic Protocol Verification*, volume 4. Instytut Łącsności (Institute of Telecommunications), Warsaw, Poland, Dec. 2002.
13. J. Goubault-Larrecq. Une fois qu'on n'a pas trouvé de preuve, comment le faire comprendre à un assistant de preuve ? In *Actes 15èmes journées francophones sur les langages applicatifs (JFLA'04)*, Sainte-Marie-de-Ré France, Janvier 2004. INRIA.
14. N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. In *Proc. of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 254–263. ACM Press, 2001.
15. M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61. ACM Press, 2001.
16. N. E. Kadhi. Automatic verification of confidentiality properties of cryptographic programs. *Networking and Information Systems*, 3(6), 2001.
17. G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131–133, 1995.
18. R. Milner. *Communication and concurrency*. Prentice Hall International (UK) Ltd., 1995.
19. R. Needham and M. Schroeder. Using encryption for authentification in large networks of computers. *Communications of the ACM*, 21(12), December 1978.
20. F. Nielson, H. R. Nielson, and H. Seidl. Normalizable horn clauses, strongly recognizable relations, and spi. In *Proceedings of the 9th International Symposium on Static Analysis*, volume 2477, pages 20–35. Springer-Verlag, 2002.
21. F. Parrennes. The CSur project. http://www.lsv.ens-cachan.fr/csur/, 2004.
22. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Sys.*, 24(3):217–298, may 2002.
23. P. Selinger. Models for an adversary-centric protocol logic. *Electronic Notes in Theoretical Computer Science*, 55(1):73–87, July 2001. Proceedings of the 1st Workshop on Logical Aspects of Cryptographic Protocol Verification (LACPV'01), J. Goubault-Larrecq, ed.
24. A. Simon and A. King. Analyzing string buffers in C. In *Intl. Conf. on Algebraic Methods and Software Methodology (AMAST'2002)*, pages 365–379, 2002.
25. D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.
26. C. Weidenbach. Towards an automatic analysis of security protocols. In H. Ganzinger, editor, *16th International Conference on Automated Deduction CADE-16*, volume 1632, pages 378–382. Springer, 1999.
27. C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS version 2.0. In *Proc. 18th Conference on Automated Deduction (CADE 2002)*, pages 275–279. Springer-Verlag LNCS 2392, 2002.