Jean Goubault-Larrecq
Fabrice Parrennes

Cryptographic Protocol Analysis on Real C Code

Research Report LSV-09-18

July 2009

Laboratoire Spécification et Vérification

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

Ecole Normale Supérieure de Cachan
61, avenue du Président Wilson
94235 Cachan Cedex France

Cryptographic Protocol Analysis on Real C Code ⋆

Jean Goubault-Larrecq [a,*] Fabrice Parrennes [b,1]

[a] *LSV/UMR CNRS & ENS Cachan, INRIA Futurs projet SECSI*
61, av. du président-Wilson, 94235 Cachan Cedex, France

[b] *RATP, EST/ISF/QS LAC VC42*
40 bis Roger Salengro, F-94724 Fontenay-sous-Bois, France

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.

Key words: cryptographic protocols, static analysis, Dolev-Yao model, Horn clause, trust assertion, $\mathcal{H}_1$
PACS:

⋆ 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.

* Corresponding author. Fax: +33-1 47 40 75 21.
Email addresses: goubault@lsv.ens-cachan.fr (Jean Goubault-Larrecq), fabrice.parrennes@ratp.fr (Fabrice Parrennes).
URL: http://www.lsv.ens-cachan.fr/~goubault/ (Jean Goubault-Larrecq).
[1] Work done while at LSV, UMR CNRS & ENS Cachan, INRIA Futurs projet SECSI.

1 Introduction

Cryptographic protocol verification has come of age: there are now many ways of verifying cryptographic protocols in the literature (see [1] 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 [2]. It is well-known that there is an attack against this protocol (see [3]). This attack also makes $N_{\mathtt{B}}$ available to the intruder, although $N_{\mathtt{B}}$ was meant to remain secret.

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

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 $\mathtt{pub}(\mathtt{B})$. 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 private key $\mathtt{prv}(\mathtt{B})$, 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 $\mathtt{prv}(\mathtt{A})$, 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:

- 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

```
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(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 *msg, nonce_t *n1, int *id,
26                  int *dest) {
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->id_1       rec V and
39        msg->dest_1     rec W     | n1 rec U, id rec V,
40                                    dest rec W. % */
41   return(0);
42 }
43
44 ssize_t write(int fd, const void *buf, size_t count)
45 {
46   ssize_t n =  _write (fd, buf, count);
47   /* % knows B | buf rec B. % */
48   return n;
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

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 if 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 [4] 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 are able to deal with.

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 [5]), they can be and have already been analyzed [6,7]. 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 7. We conclude in Section 8.

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 [8,9], 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 [10], 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 [8].

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 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 [9] 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 [9] 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 [9] 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 [11]), 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 [1] for an overview. One of the most successful models today is the *Dolev-Yao model* [10], 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 [12], 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):

$$\texttt{knows}(\texttt{nil}) \tag{1}$$

$$\texttt{knows}(\texttt{cons}(X,Y)) \Leftarrow \texttt{knows}(X), \texttt{knows}(Y) \tag{2}$$

(Intruder can build lists.)

$$\texttt{knows}(X) \Leftarrow \texttt{knows}(\texttt{cons}(X,Y)) \tag{3}$$

$$\texttt{knows}(Y) \Leftarrow \texttt{knows}(\texttt{cons}(X,Y)) \tag{4}$$

(Intruder can read all elements of a list.)

$$\texttt{knows}(\texttt{crypt}(X,Y)) \Leftarrow \texttt{knows}(X), \texttt{knows}(Y) \tag{5}$$

(Intruder can encrypt.)

$$\texttt{knows}(X) \Leftarrow \texttt{knows}(\texttt{crypt}(X, \texttt{pub}(Y))), \texttt{knows}(\texttt{prv}(Y)) \tag{6}$$

$$\texttt{knows}(X) \Leftarrow \texttt{knows}(\texttt{crypt}(X, \texttt{prv}(Y))), \texttt{knows}(\texttt{pub}(Y)) \tag{7}$$

$$\texttt{knows}(X) \Leftarrow \texttt{knows}(\texttt{crypt}(X, \texttt{sk}(Y,Z))), \texttt{knows}(\texttt{sk}(Y,Z)) \tag{8}$$

(Intruder can decrypt if he knows the inverse key.)

$$\texttt{knows}(\texttt{pub}(X)) \quad \text{(Intruder knows public keys.)} \tag{9}$$

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:

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

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

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 [11] 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 [13,14].

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.* [15]. This analysis gives very precise information on the shape of objects stored in variables. It is also rather costly. A crucial observation in [15] 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 [16] gives a very rough approximation of what variables may point to what others, but can be computed extremely efficiently [17]. (See [18] 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. (For the latter, we assume that some stubs are provided, e.g., see `write` at lines 44-49, Figure 2, containing trust assertions, see below.) We also consider a restricted subset of C, where casts are absent, and expressions are assumed to be well-typed. We also disallow negative array indices. 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 control-flow graphs G_f. 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 the set $\mathcal{V} = \mathcal{V}_{\text{glob}} \cup \mathcal{V}_{\text{loc}}$ of *program variables*, separated in *local variables* (in $\mathcal{V}_{\text{loc}}$) and *global variables* (in $\mathcal{V}_{\text{glob}}$), with $\mathcal{V}_{\text{loc}} \cap \mathcal{V}_{\text{glob}} = \emptyset$; c ranges over constants of the form `int` n, $n \in \mathbb{Z}$ (representing the integer n) or `float` r, with r a floating-point number; f ranges over function names, a or lab over struct field names, op ranges over primitive operations (arithmetic operations, bitwise logical operations, comparisons), s over some unspecified set $\mathcal{S}$ of so-called *allocation sites*, and zt ranges over *zone types*:

$zt ::=$	`int`	integer
$\mid$	`float`	floating-point
$\mid$	`ptr`	pointer
$\mid$	`struct` $\{lab_1 = zt_1, \ldots, lab_n = zt_n\}$	structure, with pairwise distinct labels lab_i, $1 \leq i \leq n$
$\mid$	`array`$(zt_1, \ldots, zt_n)$	array of n zone types
$\mid$	`array`$^*(zt)$	array, unknown number of entries

Allocation sites and zone types are used only in the memory allocation instruction `new`. Up to some details, zone types are just plain C types, where all pointer types are abstracted to the single `ptr` zone type, and where the length of arrays is made explicit (`array`$(zt_1, \ldots, zt_n)$). The `array`$^*(zt)$ construction denotes arrays of unknown length, with elements of zone type zt.

The idea of this set of instructions is that it is a form of assembly language, of a lower level than C code itself, where variables x, y, ..., denote registers from an infinite set. Translating C code to such control-flow graphs is tantamount to writing a compiler to a form of 3-address code. This is classical. We only illustrate a few points.

First, C variables must be allocated. E.g., declaring a variable `mesg1` by writing `msg1_t mesg1;` (Figure 2, l.52) is compiled by first creating a fresh local variable $@mesg1 \in \mathcal{V}_{\text{loc}}$ denoting the address of `mesg1`, by emitting an instruction $@mesg1 :=$ `new`$_s$ zt, for some fresh allocation site s, and where zt is the zone type associated with the type `msg1_t`. For example, assuming that `msg1_t` is declared as

```
#define SIZENONCE 16
typedef struct msg1_s msg1_t;
struct msg1_s {
```

$i \in Instr$	$::=$	$x := y$	variable copy
	$\mid$	$x := c$	storing constant c into x
	$\mid$	$x := f$	storing the address of function f into x
	$\mid$	$x := *y$	reading from a pointer
	$\mid$	$*x := y$	storing into a pointer
	$\mid$	$x := \&y[z]$	taking address of subarray at index z
	$\mid$	$x := \&\Diamond y$	address of first element of array y
	$\mid$	$x := \&y \xrightarrow{j} a$	taking the address of field a in struct y
	$\mid$	$x := op(x_1, \ldots, x_n)$	primitive call
	$\mid$	$x := \texttt{new}_s\ zt$	memory allocation
	$\mid$	$?x == 0$	zero test
	$\mid$	$?x\,!= 0$	non zero test
	$\mid$	$x := \texttt{call}\ g(y)$	calling function g
	$\mid$	$x := \texttt{call}*\ z(y)$	indirect call
	$\mid$	$\texttt{trust}\ A \Leftarrow A_1, \ldots, A_n$	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

```
  int id_1[4]; /* Alice's user id. */
  int dest_1[4]; /* The user id of whoever Alice wants to talk to. */
  char nonce_msg1[SIZENONCE]; /* random nonce Na */
};
```

zt would be

$$\begin{aligned}\texttt{struct}\,\{\texttt{id\_1} &= \texttt{array}(\underbrace{\texttt{int}, \ldots, \texttt{int}}_{\text{4 times}}),\\ \texttt{dest\_1} &= \texttt{array}(\underbrace{\texttt{int}, \ldots, \texttt{int}}_{\text{4 times}}),\\ \texttt{nonce\_msg1} &= \texttt{array}(\underbrace{\texttt{int}, \ldots, \texttt{int}}_{\text{16 times}})\}\end{aligned}$$

The reason why we create local variables such as $@mesg1$ is to simplify the handling of expressions such as `&mesg1` in C. With this scheme, `&mesg1` is compiled

merely as the value of the local variable $@mesg1$. On the other hand, a C assignment `x=y` is compiled to the sequence of two instructions $z := *@y$, $*@x := z$, where z is a fresh local variable in $\mathcal{V}_{\text{loc}}$.

Contrary to C variables, (local, global) program variables in $\mathcal{V}$ cannot be aliased. Also, we may create as many program variables as needed to translate C code to our format. In particular, complex instructions can be broken down to sequences of instructions as above. For example, assuming `mesg1` was declared as above, the expression `mesg1.id_1[0] = 192;` would be compiled as the sequence of instructions $z := 0$, $y := 192$, $x_1 := \&@mesg1 \xrightarrow{1} \texttt{id\_1}$, $x_2 := \&x_1[z]$, $x_3 = \&\Diamond x_2$, $*x_3 := y$.

The instructions $x := \&y \xrightarrow{j} a$ and $x := \&y[z]$ implement pointer arithmetic. The first adds the offset of field a to the pointer y, where a is the jth field in the structure pointed to by y. (If we know the structure type of y, yes, j and a are redundant information.) The second adds the integer z to the address y, yielding x. Precisely, if y is the address of some array a, then x will be the address of the subarray starting at the zth element of a. We choose to distinguish this address from that of the zth element itself, which will be $\&\Diamond x$. (This is a typical example of the subtle differences between arrays and pointers in C; a similar puzzling point is that, in C, if `a` is declared as `int a[10][15]`, then `a[0]` is actually exactly the same address as `a`, except it is of type `int *` instead of `int **`, and of allocation class `int [15]`. We insert instructions $x := \&\Diamond y$ at specific points, e.g., reading $y[z]$ will be done by outputting $x_1 := \&y[z]$, $x_2 := \&\Diamond x_1$, $x_3 := *x_2$, while writing x into $y[z]$ will be done by outputting $x_1 := \&y[z]$, $x_2 := \&\Diamond x_1$, $*x_2 := x$.)

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 place where it is most apparent that the language of Figure 3 is closer to an assembly language than to C is when it comes to function calls: `call` g is essentially the assembly language call-to-subroutine instruction. We associate each function g with two global variables in_g and out_g (in $\mathcal{V}_{\text{glob}}$), meant to transmit the actual parameters to the entry of g, and to get back the return value on exit from g. The effect of $x :=$ `call` $g(y)$ is to copy y to in_g, call g as a subroutine, then copy back the result out_g into x. Any actual C function call to g is translated as a sequence of instructions that first allocates space for the actual parameters in a `struct`, say y, fills it with the actual parameters, then executes $x :=$ `call` $g(y)$. The `struct` y is usually called the *call frame*. For example, the call at line 79 of Figure 2 would be compiled as the following sequence of instructions: $y := \texttt{new}_{s_1}\texttt{struct}\ \{\texttt{msg} = \texttt{ptr}, \texttt{n1} = \texttt{ptr}, \texttt{id} = \texttt{ptr}, \texttt{dest} = \texttt{ptr}\}$ (allocate frame), $i_1 := \&y \xrightarrow{1} \texttt{msg}$, $*i_1 := @mesg1$ (store first argument), $i_2 := \&y \xrightarrow{2} \texttt{n1}$, $*i_2 := @nonce$ (store second argument), $i_3 := \&y \xrightarrow{3} \texttt{id}$, $*i_3 := @ip\_id$ (compute third argument `ip_id`; since `ip_id` is a C array, the value of `ip_id` is $@ip\_id$ itself, not $*@ip\_id$—another standard subtlety of C semantics), $i_4 := \&y \xrightarrow{4} \texttt{dest}$,

$*i_4 := @ip\_dest, x :=$ `call create_mesg1`(y) (finally, call `create_mesg1`).

Dually, functions g are compiled to pieces of code that first read their arguments off the in_g variable, then compute the body of g; `return` instructions are translated as assignments to out_g, on transitions leading to the exit node $O(g)$. (Instructions such as `return;`, without any returning value, are translated to an assignment of the empty `struct` to out_g, including the implicit `return;` statement at the end of every function. Note in particular that out_g will always have received a value when returning from g.) For example, `create_nonce` (lines 1–6 of Figure 2, and ignoring the comment at line 4 for now) would be compiled as the sequence of instructions: $arg := in_{\texttt{create\_nonce}}$; $@nce := \&arg \xrightarrow{1} \texttt{nce}$, $y_1 := \&@nce \xrightarrow{1} \texttt{nonce}$ (this is the value of `nce->nonce`, an array of 16 `int`s), $y_2 := 16$, $y :=$ `new`$_{s_2}$`struct` $\{\texttt{buf} = zt_{\texttt{nonce\_t}}, \texttt{buf\_siz} = \texttt{int}\}$, $j_1 := \&y \xrightarrow{1} \texttt{buf}$, $*j_1 := y_1$, $j_2 := \&y \xrightarrow{2} \texttt{buf\_siz}$, $*j_2 := y_2$, $x :=$ `call RAND_bytes`(y), $out_{\texttt{create\_nonce}} := 0$.

Compiling C code to instructions presumes a given scheduling of elementary instructions. E.g., we might compile the C code `i ^= ++i` as, say, $z := 1$, $i := *@i$, $j := +(i, z)$ (where $+$ is the addition primitive), $*@i := j$, $i' := *@i$, $j' := \hat{}(i', j)$, $*@i := j'$, or as $z := 1$, $i' := *@i$, $i := *@i$, $j := +(i, z)$, $*@i := j$, $j' := \hat{}(i', j)$, $*@i := j'$. Assuming `i` is initially 1, the first sequence would end up storing 0 into `i`, while the second sequence would instead compute $1\hat{}2 = 3$. It is tempting to just forbid C code such as `i ^= ++i`, which is semantically illegal. However, we should in general assume that even illegal code is sometimes compiled in actual applications. A remedy to this is, in order to verify output from a given C compiler, to use the same scheduling as the compiler.

The only non-standard instruction in Figure 3 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 `trust` x `rec` $t \Leftarrow x_1$ `rec` $t_1, \ldots, x_n$ `rec` t_n relates the value of program variables (x, x_1, …, x_n) to terms (messages; $t, t_1, \ldots, t_n$) 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 `rec` t state that the value of x denotes t, other atomic formulae $P(t)$ (e.g., `knows`(t), see Section 2) will typically be defined by the external trust model (see Section 4.2).

Formally, the syntax of trust assertions is that of Horn clauses on atoms, which may either be of the form x `rec` t, where $x \in \mathcal{V}$ and t is a first-order term, or of the form $P(t)$, where P is a predicate symbol distinct from `rec` and t is a first-order term. In each case, we assume that t is built on a signature that does not contain any program variable x, using logical variables X, Y, Z, which are implicitly universally quantified at the level of clauses, and are assumed distinct from all program variables.

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 `trust` $cipher$ `rec` `crypt`$(M, K) \Leftarrow msg$ `rec` M, key_pub `rec` K, preceded by instructions that compute the values of local variables $cipher$, msg, key_pub: $cipher := *@cipher$, $msg := *@msg$, $key\_pub := *@key\_pub$. This trust statement 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 `crypt`(M, K); in other words, `encrypt_mesg` computes the encryption of `msg` using key `key_pub` and stores it into `cipher`. (We restrict comments enclosed between `/* %` and `% */`, syntactically, to ensure that no embedded C expression has any side effect or function call.)

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*, split into three disjoint denumerable sets, $Addr_{val}$ of *valid addresses*, $Addr_{txt}$ of *code addresses*, and $Addr_{inval}$ of *invalid addresses*. Invalid addresses include pointers such as `NULL`, or odd addresses on word-aligned systems. We also assume that, for every function name f, there is a distinct address fun $f \in Addr_{txt}$. Finally, we let $Addr_{val}$ be the disjoint union of the finitely many denumerable sets $Addr_{val}[s]$, where s ranges over all allocation sites of the program. We mean valid addresses to denote those memory addresses returned by the $x :=$ `new`$_s$ tz in-

structions: this accounts for memory allocation functions, e.g., `malloc`, but also allocation of C variables on the stack, as we have seen. The result of $\texttt{new}_s\ tz$ is an address in $Addr_{val}[s]$. The trick of using sets $Addr_{val}[s]$ of addresses for each allocation site s allows us to record which addresses may have been allocated at each site, and will ease correctness proofs. In general, we may imagine that $\texttt{new}_s\ zt$ just allocates addresses in $Addr_{val}$, and that $Addr_{val}[s]$ is the set of addresses that were allocated at site s during any given run of the program. A *store* $\mu \in Store$ is any partial function from valid adresses to zones.

Zones describe the layout of data stored at given addresses, and are described by the following grammar:

$$
\begin{array}{lll}
z ::= \texttt{int}\ n & & \text{integer } n \\
\quad \mid \texttt{float}\ r & & \text{floating-point value } x \\
\quad \mid \texttt{ptr}\ \ell & & \text{pointer, pointing to location } \ell \\
\quad \mid \texttt{struct}\ \{lab_1 = z_1, \ldots, lab_n = z_n\} & & \text{structure, with pairwise distinct} \\
 & & \qquad \text{labels } lab_i,\ 1 \leq i \leq n \\
\quad \mid \texttt{array}(z_1, \ldots, z_n) & & \text{array of } n \text{ sub-zones}
\end{array}
$$

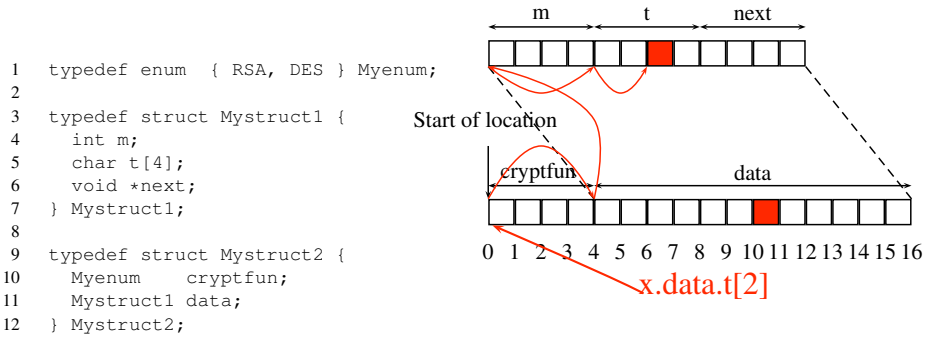

Fig. 4. Sample memory zone

We define the relation $:\prec$ relating zones and zone types as the smallest such that

$$\texttt{int}\ n :\prec \texttt{int} \qquad \texttt{float}\ r :\prec \texttt{float} \qquad \texttt{ptr}\ a :\prec \texttt{ptr} \text{ for every } a \in Addr_{inval}$$

$$\frac{z_1 :\prec zt_1 \quad \ldots \quad z_n :\prec zt_n}{\texttt{struct}\ \{lab_1 = z_1, \ldots, lab_n = z_n\} :\prec \texttt{struct}\ \{lab_1 = zt_1, \ldots, lab_n = zt_n\}}$$

$$\frac{z_1 :\prec zt \quad \ldots \quad z_n :\prec zt}{\texttt{array}(z_1, \ldots, z_n) :\prec \texttt{array}^*(zt)} \qquad \frac{z_1 :\prec zt_1 \quad \ldots \quad z_n :\prec zt_n}{\texttt{array}(z_1, \ldots, z_n) :\prec \texttt{array}(zt_1, \ldots, zt_n)}$$

(We shall explain later why a is restricted to be in $Addr_{inval}$ in the axiom $\texttt{ptr}\ a :\prec \texttt{ptr}$.)

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*. Let Lab be a set of so-called *labels* lab (i.e., field names in `structs`); selectors are either pairs $\langle lab, i\rangle$ where $lab \in Lab$ and $i \geq 1$ (i denotes the position of the field lab in a given `struct`), or integers $n \geq 1$ (array indices), or the constant $\Diamond$ (read first element of array). For example, in Figure 4, if a is the address of `x` of type `Mystruct2`, $a.\langle\texttt{data}, 2\rangle.\langle\texttt{t}, 2\rangle.3.\Diamond$ is the location of the cell `x.data.t[2]`, while $a.\langle\texttt{data}, 2\rangle.\langle\texttt{t}, 2\rangle.3$ is the location of the subarray consisting of the elements `x.data.t[2]` and `x.data.t[3]`.

Let Loc be the set of all locations. Let $Store$ be the set of all stores, i.e., of all partial functions from $Addr_{val}$ to $Zone$. Any store μ extends in a unique way to a partial map $\hat{\mu}$ from *locations* to zones: if $a \in Addr_{val}$ and $\mu(a)$ is defined, then $\hat{\mu}(a) = \mu(a)$; $\hat{\mu}(\ell.\langle lab, i\rangle) = z_i$ provided $\hat{\mu}(\ell)$ is defined and of the form $\texttt{struct}\ \{lab_1 = z_1, \ldots, lab_n = z_n\}$, $1 \leq i \leq n$, with $lab = lab_i$; $\hat{\mu}(\ell.j) = \texttt{array}(z_{j+1}, \ldots, z_n)$ provided $\hat{\mu}(\ell)$ is defined and of the form $\texttt{array}(z_1, \ldots, z_n)$, $0 \leq j \leq n$; and $\hat{\mu}(\ell.\Diamond) = z_1$ provided $\hat{\mu}(\ell)$ is defined and of the form $\texttt{array}(z_1, \ldots, z_n)$, $n \geq 1$. E.g., `x.data` has a location, namely $a.\langle\texttt{data}, 2\rangle$, mapped by $\hat{\mu}$ to the zone shown in Figure 4, top right.

Given a C program mapping each function f to G_f (Section 3), we define its semantics as a transition system. *Transitions* (inside G_f) are defined by judgments $q, \rho, \mu, \mathcal{R}, \mathcal{B} \xrightarrow{i} q', \rho', \mu', \mathcal{R}', \mathcal{B}'$, one for each edge $q \xrightarrow{i} q'$ in G_f, where ρ and ρ' are *environments* mapping variables to their values. (On first reading, please just ignore the $\mathcal{R}$ and $\mathcal{B}$ components; these will be dealt with in Section 4.1.) *Variable values* are those zones of the form $\texttt{int}\ n$, $\texttt{float}\ r$, and $\texttt{ptr}\ \ell$. Let $\mathcal{V}al$ be the set of variable values.

The concrete semantics is shown in Figure 5, which describes a deduction system with 12 axioms (top) and 5 inference rules (bottom). The additional axiom describing the effet of trust assertions will be given separately, in Section 4.1. The relation $\longrightarrow^*$ is the relation of intraprocedural reachability, and is used to give a semantics to function calls.

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 $a \in Addr_{val}$ and $z \in Zone$. To model writing into arbitrary locations ℓ, not just addresses, we let $\mu[\ell := z]$ be partially defined by: $\mu[a := z] = \mu[a \mapsto z]$ if $a \in Addr_{val}$ is such that $\mu(a)$ is defined; $\mu[\ell.\langle lab, i\rangle := z] = \mu[\ell := \texttt{struct}\ \{lab_1 = z_1, \ldots, lab_i = z, \ldots, lab_n = z_n\}]$ whenever $\hat{\mu}(\ell) = \texttt{struct}\ \{lab_1 = z_1, \ldots, lab_n = z_n\}$, $1 \leq i \leq n$, and $lab = lab_i$; $\mu[\ell.i := z] = \mu[\ell := \texttt{array}(z_1, \ldots, z_i, z_{i+1}, \ldots, z_n)]$ where $z = \texttt{array}(z_{i+1}, \ldots, z_n)$, whenever $\hat{\mu}(\ell) = \texttt{array}(z_1, \ldots, z_i, z'_{i+1}, \ldots, z'_n)$ for some

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{x:=y} q',\rho[x\mapsto\rho(y)],\mu,\mathcal{R},\mathcal{B}$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{x:=c} q',\rho[x\mapsto c],\mu,\mathcal{R},\mathcal{B}$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{x:=f} q',\rho[x\mapsto \texttt{ptr}\ (\texttt{fun}\ f)],\mu,\mathcal{R},\mathcal{B}$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{x:=*y} q',\rho[x\mapsto\hat{\mu}(\ell)],\mu,\mathcal{R},\mathcal{B} \quad \text{if } \rho(y)=\texttt{ptr}\ \ell, \hat{\mu}(\ell) \text{ is defined and in } \mathcal{V}val$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{*x:=y} q',\rho,\mu[\ell:=\rho(y)],\mathcal{R},\mathcal{B} \quad \text{if } \rho(x)=\texttt{ptr}\ \ell, \mu[\ell:=\rho(y)] \text{ is defined}$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{x:=\&y[z]} q',\rho[x\mapsto \texttt{ptr}(\ell.j)],\mu,\mathcal{R},\mathcal{B}$$
$$\text{if } \rho(y)=\texttt{ptr}\ \ell, \hat{\mu}(\ell)=\texttt{array}\ (z_1,\ldots,z_n),$$
$$\text{and } \rho(z)=\texttt{int}\ j, 0\le j\le n$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{x:=\&\Diamond y} q',\rho[x\mapsto \texttt{ptr}(\ell.\Diamond)],\mu,\mathcal{R},\mathcal{B}$$
$$\text{if } \rho(y)=\texttt{ptr}\ \ell, \hat{\mu}(\ell)=\texttt{array}\ (z_1,\ldots,z_n), n\ge 1$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{x:=\&y\overset{j}{\rightarrow}a} q',\rho[x\mapsto \texttt{ptr}(\ell.\langle a,j\rangle)],\mu,\mathcal{R},\mathcal{B}$$
$$\text{if } \rho(y)=\texttt{ptr}\ \ell, \hat{\mu}(\ell)=\texttt{struct}\ \{lab_1=z_1,\ldots,lab_n=z_n\},$$
$$1\le j\le n, a=lab_j$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{x=op(x_1,\ldots,x_n)} q',\rho[x\mapsto \widehat{op}(\rho(x_1),\ldots,\rho(x_n))],\mu,\mathcal{R},\mathcal{B}$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{x:=\texttt{new}_s\ zt} q',\rho[x\mapsto \texttt{ptr}\ a],\mu[a\mapsto z],\mathcal{R},\mathcal{B}$$
$$\text{if } a\in Addr_{val}[s]\setminus \mathrm{dom}\ \mu, z\in Zone, z:\prec zt$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{?x==0} q',\rho,\mu,\mathcal{R},\mathcal{B} \text{ if } \rho(x)=\texttt{int}\ 0$$

$$q,\rho,\mu,\mathcal{R},\mathcal{B} \xrightarrow{?x\models 0} q',\rho,\mu,\mathcal{R},\mathcal{B} \text{ if } \rho(x)\neq\texttt{int}\ 0$$

$$\frac{I(G_g),\rho_{|\mathcal{V}_{\mathrm{glob}}}[in_g\mapsto\rho(y)],\mu,\mathcal{R},\mathcal{B}\longrightarrow^* O(G_g),\rho',\mu',\mathcal{R}',\mathcal{B}'}{q,\rho,\mu,\mathcal{R},\mathcal{B}\xrightarrow{x:=\texttt{call}\ g(y)} q',(\rho\oplus\rho'_{|\mathcal{V}_{\mathrm{glob}}})[x\mapsto\rho'(out_g)],\mu',\mathcal{R}',\mathcal{B}'}$$

$$\frac{\rho(z)=\texttt{ptr}\ (\texttt{fun}\ g)\quad I(G_g),\rho_{|\mathcal{V}_{\mathrm{glob}}}[in_g\mapsto\rho(y)],\mu,\mathcal{R},\mathcal{B}\longrightarrow^* O(G_g),\rho',\mu',\mathcal{R}',\mathcal{B}'}{q,\rho,\mu,\mathcal{R},\mathcal{B}\xrightarrow{x:=\texttt{call*}\ z(y)} q',(\rho\oplus\rho'_{|\mathcal{V}_{\mathrm{glob}}})[x\mapsto\rho'(out_g)],\mu',\mathcal{R}',\mathcal{B}'}$$

$$\frac{}{q,\rho,\mu,\mathcal{R},\mathcal{B}\longrightarrow^* q,\rho,\mu,\mathcal{R},\mathcal{B}} \qquad \frac{q,\rho,\mu,\mathcal{R},\mathcal{B}\xrightarrow{i} q',\rho',\mu',\mathcal{R}',\mathcal{B}'}{q,\rho,\mu,\mathcal{R},\mathcal{B}\longrightarrow^* q',\rho',\mu',\mathcal{R}',\mathcal{B}'}$$

$$\frac{q,\rho,\mu,\mathcal{R},\mathcal{B}\longrightarrow^* q',\rho',\mu',\mathcal{R}',\mathcal{B}' \quad q',\rho',\mu',\mathcal{R}',\mathcal{B}'\longrightarrow^* q'',\rho'',\mu'',\mathcal{R}'',\mathcal{B}''}{q,\rho,\mu,\mathcal{R},\mathcal{B}\longrightarrow^* q'',\rho'',\mu'',\mathcal{R}'',\mathcal{B}''}$$

Fig. 5. Concrete semantics

$z'_{i+1}, \ldots, z'_n$; and $\mu[\ell.\Diamond := z] = \mu[\ell := \texttt{array}(z, z_2, \ldots, z_n)]$ whenever $\hat{\mu}(\ell) = \texttt{array}(z_1, z_2, \ldots, z_n)$ for some z_1, $n \geq 1$. (In particular, $\mu[\ell.j.\Diamond := z]$ writes z at position j of the array at ℓ.) The notation $\mu[\ell := z]$ is undefined in all other cases. This extension is used in the semantics of $*x := y$. We write $\rho_{|\mathcal{V}_{\text{glob}}}$ the restriction of ρ to the set $\mathcal{V}_{\text{glob}}$ of global variables (including the in_g and out_g variables), and $\rho \oplus \rho'$ the environment mapping every $x \in \text{dom}\rho \cup \text{dom}\rho'$ to $\rho(x)$ if $x \notin \text{dom}\rho'$, to $\rho'(x)$ otherwise.

The $x := \texttt{new}_s\ zt$ instruction allocates a fresh address a, stores it into x, and extends the store so that we now find at a a zone whose type is zt, and where `ptr` fields are instantiated to invalid addresses (remember that $\texttt{ptr}\ a :\prec \texttt{ptr}$ if and only if a is in $Addr_{inval}$). To be fair, we assume this so as to be able to keep an efficient static analysis scheme that does not lose too much precision. In a security scenario, it is moreover reasonable to assume that either all pointers will be zeroed out (using `calloc()` instead of `malloc()`), or that memory will be filled with random values prior to allocation (e.g., as in ExecShield [19]), making the probability that a given random bit pattern corresponds to anything but an invalid address extremely low.

While instances of `new` meant to allocate local variables pose no problem, some trickery is needed to translate C `malloc` calls to an $x := \texttt{new}_s\ zt$ instruction, and in particular to find the zone type zt. In most cases, where `malloc` is called as `x = malloc (...);`, zt can be inferred from the type of `x`. Because a pointer to type τ can also point to an array of elements of type τ, we must e.g., translate a `malloc` call for an `int` variable as a $\texttt{new}_{\texttt{array*(int)}}$ instruction. Special care must also be taken with variable-length arrays: e.g., a `struct` with a final field, say, `int a[1]`, must be translated through a type zone $\texttt{struct}\ \{\ldots, \texttt{a} = \texttt{array*(int)}\}$ to allow for the `a` array to contain more (or less) than one entry.

Most other entries of Figure 5 are self-explanatory. In the semantics of primitive calls $x = op(x_1, \ldots, x_n)$, we assume that the semantics $\widehat{op}$ of op is given separately. We also assume that all considered primitives return either ints or floats, but no pointer. We let τ_{op} be the *return type* of op, either `int`, if $\widehat{op}$ always returns integers, or `float`, if $\widehat{op}$ always returns floats.

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 asser-

tions 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 $Val \times Term_0$, 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 `ptr` ℓ, 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.

In a previous version of this paper, we claimed that only the contents of the zone pointed to by `cipher1` should be relevant, not the location ℓ. The irrelevance of ℓ was handled by equating bisimilar locations. Briefly, two pointers were equated provided they pointed to equal zones, and zones were equated provided they had the same structure and corresponding pointers inside them were equated, taking greatest fixpoints in case of infinite cycles. However, the abstract semantics that we proposed, just like the one of this paper (Section 5), is sensitive to address changes, and is therefore not invariant under bisimilarity. In other words, our abstract semantics is not sound with respect to a concrete semantics that takes C values to be zones up to bisimilarity.

Let's take an example to illustrate what a C value should be: look at line 47 of Figure 2, stating the trust assertion `trust knows`$(B) \Leftarrow @buf$ `rec` B. What will be sent to an intruder is the contents of the zone pointed to by the pointer $*@buf$. If this contains pointers, they will be sent in clear, too. So we must assume that C values are just plain zones, including the actual values of embedded pointers. However the actual location ℓ of the zone itself should be disregarded, that is, $@buf$ `rec` B should mean that $@buf$ is a pointer to some location ℓ such that $\hat{\mu}(\ell)$ is related to B by $\mathcal{R}$. Formally, define $\rho, \mu, \mathcal{R}, \mathcal{B} \models x$ `rec` t, where t is a ground term, if and only if $\rho(x) =$ `ptr` ℓ for some location ℓ, $\hat{\mu}(\ell)$ is defined and $(\hat{\mu}(\ell), t) \in \mathcal{R}$.

Define the semantics of other ground atoms A by $\rho, \mu, \mathcal{R}, \mathcal{B} \models A$ if and only if $A \in \mathcal{B}$.

Trust assertions `trust` $A \Leftarrow A_1, \ldots, A_n$ are meant to add new relations to $\mathcal{R}$ and new facts to $\mathcal{B}$. To make this explicit, 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 clause $A \Leftarrow A_1, \ldots, A_n$ and saturating under repeated application of the clauses in $\mathcal{M}$.

Formally, 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 $\subseteq^2$ 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 $T^{\rho,\mu}_{\mathcal{M}}(\mathcal{R}, \mathcal{B})$ be the union over all $C \in \mathcal{M}$ of $T^{\rho,\mu}_{C}(\mathcal{R}, \mathcal{B})$. (This is the familiar T_P operator of Prolog semantics.) Let lfp $T^{\rho,\mu}_{\mathcal{M}}(\mathcal{R}, \mathcal{B})$ be the least fixpoint of $T^{\rho,\mu}_{\mathcal{M}}$ above $(\mathcal{R}, \mathcal{B})$. Write $\cup^2$ for componentwise union of pairs of sets. The semantics of trust assertions is given in Figure 6.

$$q, \rho, \mu, \mathcal{R}, \mathcal{B} \xrightarrow{\texttt{trust}\ A \Leftarrow A_1, \ldots, A_n} q', \rho, \mu, \mathcal{R}', \mathcal{B}'$$
$$\text{where} \qquad (\mathcal{R}', \mathcal{B}') = \text{lfp}\ T^{\rho,\mu}_{\mathcal{M}}((\mathcal{R}, \mathcal{B}) \cup^2 T^{\rho,\mu}_{A \Leftarrow A_1, \ldots, A_n}(\mathcal{R}, \mathcal{B}))$$

Fig. 6. Concrete semantics of trust assertions

To simplify things a bit, imagine that $\mathcal{M}$ is empty. So $(\mathcal{R}', \mathcal{B}') = (\mathcal{R}, \mathcal{B}) \cup^2 T^{\rho,\mu}_{A \Leftarrow A_1, \ldots, A_n}(\mathcal{R}, \mathcal{B})$. In particular, if i is `trust` x `rec` $t \Leftarrow x_1$ `rec` $t_1, \ldots, x_n$ `rec` t_n, and $\rho(x) =$ `ptr` ℓ, $\rho(x_i) =$ `ptr` ℓ_i $(1 \leq i \leq n)$, then $\mathcal{B}' = \mathcal{B}$ and

$$\mathcal{R}' = \mathcal{R} \cup \{(\hat{\mu}(\ell), t\sigma) \mid (\hat{\mu}(\ell_1), t_1\sigma) \in \mathcal{R} \text{ and } \ldots \text{ and } (\hat{\mu}(\ell_n), t_n\sigma)\}$$

where σ ranges over all substitutions such that $t\sigma$, $t_1\sigma$, ..., $t_n\sigma$ are ground terms. In other words, 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 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 communicate, 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.)

One may naturally compare this approach to the rely-guarantee method [20]: our

trust assertions form the rely part, and our static analysis will compute the guarantee part.

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). We also include clauses 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 [12] or Nielson et al.'s [21] approaches. (We tend to prefer the latter for pragmatic reasons: the output clauses are always in the decidable class $\mathcal{H}_1$; more detail in Section 5.1.)

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 Figure 6. The effect of applying $\mathrm{lfp}\ T_{\mathcal{M}}^{\rho,\mu}$ is to close the set of 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 address* with each C variable and each memory allocation site, in the form of a fresh constant, taken from a finite set. This is particularly simple in the language of Figure 3: we just take abstract addresses to be the allocation sites s given as subscripts in the finitely many instructions $x := \mathtt{new}_s\ zt$ that occur in the program.

We shall describe our abstract semantics through Horn clauses over some arbitrary first-order signature. Given Horn clauses $C_1, \ldots, C_k$ such that at most one of them is definite, and which have first been renamed so as to share no free variable, write their disjunction $C_1 \vee \ldots \vee C_k$; this is again a Horn clause. The `knows` unary predicate is conventionally used to represent what messages a Dolev-Yao intruder may infer. We reserve the binary predicate p to denote a form of points-to relation: $\mathsf{p}(t, t')$ states that t is a location that may point to zone t'. This is the abstract counterpart of $\hat{\mu}$. We reserve unary predicates $\mathtt{init}_{zt}$ for each zone type zt to denote all zones z such that $z :\prec zt$; the constant `inval` denotes any invalid address, while the constant fun_f, for each function name f, denote the address of f. We also reserve a binary predicate `rec` to denote the concrete predicate `rec`, and a binary predicate val such that $\mathsf{val}(\mathsf{c}_x, t)$ means that variable x may have t as value. The term c_x,

here, is a fresh constant for each program variable $x \in \mathcal{V}$ except for variables of the form in_g or out_g. If x is of the form $x = in_g$ (resp. $x = out_g$), let c_x denote the term $\mathtt{in}(\mathsf{fun}_g)$ (resp. $\mathtt{out}(\mathsf{fun}_g)$), where `in`, `out` are unary function symbols. We also use the following function symbols: the constants `int` and `float` (denoting integers and floats; we disregard actual values); the unary functions `ptr` (denoting addresses), asel (denoting access to the subarray starting at the second element of an array), $\Diamond$ (denoting access to the first element of an array), $\mathsf{ssel}_{a,j}$ (access to field a, position j, in a `struct`); acons and anil are used to build arrays, while for each zone type `struct` $\{a_1 = zt_1, \ldots, a_n = zt_n\}$ we use an n-ary function symbol `struct` $\{a_1 = \_, \ldots, a_n = \_\}$; finally, each abstract zone s is itself a constant of our language. We write $\mathsf{asel}^k(t)$ for the term $\underbrace{\mathsf{asel}(\ldots(\mathsf{asel}(}_{k}\, t)\ldots)$.

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 $[\![i]\!]^{\#}$ of instruction i, mapping variable names to abstract zones, a.k.a., constants, as sets of Horn clauses, see Figure 7. The semantics of a function, resp. a whole program, is the union of the semantics of all instructions in the given control-flow graphs, plus the clauses given in Figure 8 and the auxiliary clauses of Figure 9 for each zone type zt that occurs in the given program, and finally the clauses in the external trust model $\mathcal{M}$. This yields finitely many clauses, as only finitely many zone types can be used in any given program.

Horn clauses alone would not be expressive enough to record the actual values that integer indices z may take in pointer arithmetic instructions $x := \&y[z]$. In fact, we disregard integer values entirely in the $[\![\_]\!]^\sharp$ semantics (see the definition of $[\![x := \mathtt{int}\ n]\!]^\sharp$). However, we run an auxiliary analysis, based on a given integral abstract domain, yielding a set of possible integer values $[\![z]\!]^{\#}_{int}$ for each integer variable z: in Figure 7, we assume that $[\![z]\!]^{\sharp}_{int}$ either returns a finite set of natural numbers containing all possible values of z, or returns a special symbol $\top$. Our original idea was to handle array accesses as in [22,23], distinguishing *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). Thanks to the use of Horn clauses, we may assume only one kind of array, i.e., expanded arrays represented as lists built using acons and anil. We only make a difference in selecting elements from arrays, depending on whether the index is known or not.

The abstract semantics for function calls is implemented roughly as in [17]. In fact, we have designed our core C language and its semantics so that its abstract version matches the approach of [17], where calling the known function g copies the actual parameters, using run-of-the-mill assignments, into global locations, then jumps to g's entry node (compare with our semantics of function calls, Section 4). For simplicity, we have refrained from introducing additional standard optimizations in Figure 7, e.g., keeping track of effective call sites when returning from functions in

$$
\begin{aligned}
[\![x := y]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, X) \Leftarrow \mathsf{val}(\mathsf{c}_y, X)\} \\
[\![x := \texttt{int}\ n]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, \texttt{int})\} \\
[\![x := \texttt{float}\ r]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, \texttt{float})\} \\
[\![x := f]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, \texttt{ptr}(\mathsf{fun}_f))\} \\
[\![x := *y]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, X) \Leftarrow \mathsf{val}(\mathsf{c}_y, \texttt{ptr}(Y)), \mathsf{p}(Y, X)\} \\
[\![*x := y]\!]^\sharp &= \{\mathsf{p}(X, Y) \Leftarrow \mathsf{val}(\mathsf{c}_x, \texttt{ptr}(X)), \mathsf{val}(\mathsf{c}_y, Y)\} \\
[\![x := \&y[z]]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, \texttt{ptr}(\mathsf{asel}^j(Y))) \Leftarrow \mathsf{val}(\mathsf{c}_y, \texttt{ptr}(Y)) | j \in [\![z]\!]^\sharp_{int}\} \\
&\quad \text{if } [\![z]\!]^\sharp_{int} \neq \top \\
[\![x := \&y[z]]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, \texttt{ptr}(Y)) \Leftarrow \mathsf{val}(\mathsf{c}_y, \texttt{ptr}(Y)), \\
&\quad \mathsf{val}(\mathsf{c}_x, \texttt{ptr}(\mathsf{asel}(Y))) \Leftarrow \mathsf{val}(\mathsf{c}_x, \texttt{ptr}(Y))\} \text{ otherwise} \\
[\![x := \&\Diamond y]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, \texttt{ptr}(\Diamond(Y))) \Leftarrow \mathsf{val}(\mathsf{c}_y, \texttt{ptr}(Y))\} \\
[\![x := \&y \xrightarrow{j} a]\!]^\# &= \{\mathsf{val}(\mathsf{c}_x, \texttt{ptr}(\mathsf{ssel}_{a,j}(Y))) \Leftarrow \mathsf{val}(\mathsf{c}_y, \texttt{ptr}(Y))\} \\
[\![x := op(x_1, \ldots, x_n)]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, \tau_{op})\} \quad \text{where } \tau_{op} \text{ is the return type of } op \\
[\![x := \texttt{new}_s\ zt]\!]^\sharp &= \{\mathsf{val}(\mathsf{c}_x, \texttt{ptr}(s)), \\
&\quad \mathsf{p}(s, X) \Leftarrow \texttt{init}_{zt}(X)\} \\
[\![x := \texttt{call}\ g(y)]\!]^\sharp &= \{\mathsf{val}(\texttt{in}(\mathsf{fun}_g), Y) \Leftarrow \mathsf{val}(\mathsf{c}_y, Y), \\
&\quad \mathsf{val}(\mathsf{c}_x, X) \Leftarrow \mathsf{val}(\texttt{out}(\mathsf{fun}_g), X)\} \\
[\![x := \texttt{call}* z(y)]\!]^\sharp &= \{\mathsf{val}(\texttt{in}(Z), Y) \Leftarrow \mathsf{val}(\mathsf{c}_z, \texttt{ptr}(Z)), \mathsf{val}(\mathsf{c}_y, Y), \\
&\quad \mathsf{val}(\mathsf{c}_x, X) \Leftarrow \mathsf{val}(\mathsf{c}_z, \texttt{ptr}(Z)), \mathsf{val}(\texttt{out}(Z), X)\} \\
[\![?x == 0]\!]^\sharp &= \emptyset \\
[\![?x\,!= 0]\!]^\sharp &= \emptyset \\
[\![\texttt{trust}\ A \Leftarrow A_1, \ldots, A_n]\!]^\# &= \{A^+ \vee A_1^- \vee \ldots \vee A_n^-\} \\
\text{where } (x\ \texttt{rec}\ t)^+ &= (\texttt{rec}(Z, t) \Leftarrow \mathsf{val}(\mathsf{c}_x, \texttt{ptr}(X)), \mathsf{p}(X, Z)) \\
(x\ \texttt{rec}\ t)^- &= (\bot \Leftarrow \mathsf{val}(\mathsf{c}_x, \texttt{ptr}(X)), \mathsf{p}(X, Z), \texttt{rec}(Z, t)) \\
\text{where} \quad & X, Y, Z \text{ are fresh} \\
A^+ = A, \quad & A^- = (\bot \Leftarrow A) \text{ for other atoms } A
\end{aligned}
$$

Fig. 7. Abstract semantic equations

$$
\begin{aligned}
&\mathtt{init}_{\mathtt{int}}(\mathtt{int}) \quad \mathtt{init}_{\mathtt{float}}(\mathtt{float})\\
&\mathtt{init}_{\mathtt{ptr}}(\mathtt{ptr}(\mathtt{inval}))\\
&\mathtt{init}_{zt}(\mathtt{struct}\ \{\ a_1 = X_1, \ldots, a_n = X_n\}) \\
&\qquad\qquad \Leftarrow \mathtt{init}_{zt_1}(X_1), \ldots, \mathtt{init}_{zt_n}(X_n)\\
&\qquad\qquad \text{where} \quad zt = \mathtt{struct}\ \{lab_1 = zt_1, \ldots, lab_n = zt_n\}\\
&\mathtt{init}_{zt}(\mathsf{acons}(X_1, Z)) \Leftarrow \mathtt{init}_{zt_1}(X_1), \mathtt{init}_{\mathtt{array}(zt_2,\ldots,zt_n)}(Z)\\
&\qquad\qquad \text{where} \quad zt = \mathtt{array}(zt_1, zt_2, \ldots, zt_n), n \geq 1\\
&\mathtt{init}_{\mathtt{array}()}(\mathsf{anil})\\
&\mathtt{init}_{\mathtt{array}^*(zt)}(\mathsf{acons}(X, Z)) \Leftarrow \mathtt{init}_{zt}(X), \mathtt{init}_{\mathtt{array}^*(zt)}(Z)\\
&\mathtt{init}_{\mathtt{array}^*(zt)}(\mathsf{anil})
\end{aligned}
$$

Fig. 8. Generic instances of zone types

$$
\begin{aligned}
&\mathsf{p}(X, \mathtt{struct}\ \{\ a_1 = X_1, \ldots, a_i = Y, \ldots, a_n = X_n\}) && (12)\\
&\qquad \Leftarrow \mathsf{p}(\mathsf{ssel}_{lab,i}(X), Y),\\
&\qquad\quad \mathsf{p}(X, \mathtt{struct}\ \{a_1 = X_1, \ldots, a_i = X_i, \ldots, a_n = X_n\})\\
&\qquad \text{for each} \quad \text{zone type } \mathtt{struct}\ \{a_1 = zt_1, \ldots, a_n = zt_n\}\\
&\qquad \text{where} \quad 1 \leq i \leq n, lab = a_i\\
&\mathsf{p}(X, \mathsf{acons}(Z, Y')) \Leftarrow \mathsf{p}(\mathsf{asel}(X), Y'), \mathsf{p}(X, \mathsf{acons}(Z, Y)) && (13)\\
&\mathsf{p}(\mathsf{asel}(X), Y) \Leftarrow \mathsf{p}(X, \mathsf{acons}(Z, Y)) && (14)\\
&\mathsf{p}(X, \mathsf{acons}(Y', Z)) \Leftarrow \mathsf{p}(\Diamond(X), Y'), \mathsf{p}(X, \mathsf{acons}(Y, Z)) && (15)\\
& && (16)
\end{aligned}
$$

Fig. 9. Auxiliary clauses

order to avoid spurious, fake control flow.

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 the buffer pointed to by x is always secret. This can be checked by verifying that S plus the goal clause

$$\bot \Leftarrow \mathsf{val}(\mathsf{c}_{\mathtt{x}}, \mathtt{ptr}(X)), \mathsf{p}(X, Z), \mathtt{rec}(Z, Y), \mathtt{knows}(Y) \tag{17}$$

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 [24].) 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 an closed term t denoting an address such that $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(t))$ holds, there will be another closed term t' denoting the zone pointed to by t (i.e., the C value of `x`, in the sense of Section 4.1) such that $\mathsf{p}(t, t')$ holds, and a closed term u denoting the message that we think is one possible reading of the value t' such that t' `rec` u, and which the intruder can discover, namely $\mathtt{knows}(u)$.

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

$$\bot \Leftarrow \mathsf{val}(\mathsf{c}_{\mathtt{x}}, \mathtt{ptr}(X)), \mathsf{p}(X, Z), \mathtt{rec}(Z, t) \tag{18}$$

is unsatisfiable, where X and Z are 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 most of the clauses provided in the abstract semantics are in the decidable class $\mathcal{H}_1$, and in fact in the polynomial-time decidable (cubic) subclass $\mathcal{H}_3$ [21]. $\mathcal{H}_3$ is the class of Horn clauses C where all atoms are linear, there is at most one free variable common to any two distinct atoms of C, and the graph whose vertices are atoms of C, and whose edges link atoms having one free variable in common, is acyclic. We refer the interested reader to op.cit. and to [25], where it is shown that while the intermediate class $\mathcal{H}_2$ is DEXPTIME-complete, just like its super-class $\mathcal{H}_1$, a certain sub-class $\mathsf{H}_2^{i,a}$ of $\mathcal{H}_2$ (where i is a bound on the number of occurrences of variables in the body, and a is a bound on the arity of function and predicate symbols) is polynomial-time decidable for every $i, a \in \mathbb{N}$. Now it can be observed that all the clauses in Figure 7 are in $\mathcal{H}_3$, except possibly for trust assertions (we shall return to these clauses when we consider the external trust model below); the i and a bounds can be taken to be 2. The clauses in Figure 8 are also in $\mathcal{H}_3$, with an i bound of 1, and an a bound of 3.

None of the clauses in Figure 9 is in $\mathcal{H}_3$, or even in $\mathcal{H}_1$. However, we may use a similar trick as in [21, Section 6], where some non-$\mathcal{H}_1$ clauses are instantiated to a finite number of $\mathcal{H}_3$ instances. The trick consists in observing that, if the set of abstract locations (terms built on allocation sites s, using asel, $\mathsf{ssel}_{a,i}$, $\Diamond$) corresponding to actual concrete locations is finite, then we may replace the clauses of Figure 9 by instances where X is replaced by finitely many ground terms instead. The resulting clauses are then in $\mathcal{H}_3$. Since a given C program contains at most finitely many allocation sites s and zone types zt as arguments to `new`, it would

suffice to observe that the number of locations inside any given zone type is finite. Unfortunately, while this would indeed be the case without pointer arithmetic, the possibility of doing pointer arithmetic on arrays of unknown length ruins this hope. Indeed, the set of abstract locations inside a zone type of the form $\texttt{array}^*(zt)$, itself at location ℓ, contains at least $\Diamond(\ell)$ (read first element), $\mathsf{asel}(\ell)$ (move to next entry in array), but also $\mathsf{asel}^j(\ell)$ and $\Diamond(\mathsf{asel}^j(\ell))$ for each $j \in \mathbb{N}$. Instead, we may approximate the clauses of Figure 9 in two steps, as follows.

First, we identify a finite set of locations $locs_\ell(zt)$ through the zone type zt, starting from ℓ (not necessarily all possible locations):

$$locs_\ell(\texttt{int}) = \{\ell\} \qquad locs_\ell(\texttt{float}) = \{\ell\} \qquad locs_\ell(\texttt{ptr}) = \{\ell\}$$

$$locs_\ell(\texttt{struct}\ \{lab_1 = zt_1, \ldots, lab_n = zt_n\}) = \{\ell\} \cup \textstyle\bigcup_{i=1}^n locs_{\mathsf{ssel}_{lab_i,i}(\ell)}(zt_i)$$

$$locs_\ell(\texttt{array}(zt_1, \ldots, zt_n)) = \{\ell\} \cup locs_{\Diamond(\ell)}(zt_1) \cup locs_{\mathsf{asel}(\ell)}(\texttt{array}(zt_2, \ldots, zt_n))$$

$$locs_\ell(\texttt{array}^*(zt)) = \{\ell\}$$

Since there are only finitely many allocation sites s and finitely many zone types zt per program, there are only finitely many abstract locations in $L = \bigcup_{s,zt} locs_s(zt)$. Produce all instances of clauses in Figure 9 obtained by replacing X by each ground term of L in turn, yielding $\mathcal{H}_3$ clauses. To account for the instances of clauses in Figure 9 where X takes a value outside of L, notice that the complement of L is a regular tree language; a deterministic tree automaton can be built for it in polynomial time. As such, we can write, in polynomial time, a set $S_{\overline{L}}$ of definite clauses defining a fresh predicate $p_{\overline{L}}$ such that $p_{\overline{L}}(t)$ is true in the least Herbrand model if and only if t is an abstract location outside of L. This is because deterministic, or even non-deterministic tree automata are easily encoded as sets of definite clauses [26,21,25], in fact as sets of clauses in $\mathcal{H}_3$. Now we can give an account of those instances of clauses in Figure 9 where X takes a value outside of L by adding $p_{\overline{L}}(X)$ to the body of each; e.g., by writing $\mathsf{p}(X, \mathsf{acons}(Y', Z)) \Leftarrow \mathsf{p}(\Diamond(X), Y'), \mathsf{p}(X, \mathsf{acons}(Y, Z)), p_{\overline{L}}(X)$ instead of $\mathsf{p}(X, \mathsf{acons}(Y', Z)) \Leftarrow \mathsf{p}(\Diamond(X), Y'), \mathsf{p}(X, \mathsf{acons}(Y, Z))$.

The latter clauses are still outside $\mathcal{H}_1$, but as a second step we may apply the standard approximation procedure of [25, Proposition 3, Proposition 4], which terminates in polynomial time. E.g., the latter clause $\mathsf{p}(X, \mathsf{acons}(Y', Z)) \Leftarrow \mathsf{p}(\Diamond(X), Y'), \mathsf{p}(X, \mathsf{acons}(Y, Z)), p_{\overline{L}}(X)$ will be approximated by the clauses

$$q(\mathsf{acons}(Y', Z)) \Leftarrow \mathsf{p}(\Diamond(X), Y'), \mathsf{p}(X, \mathsf{acons}(Y, Z)), p_{\overline{L}}(X)$$
$$\mathsf{p}(X, U) \Leftarrow \mathsf{p}(\Diamond(X), Y'), \mathsf{p}(X, \mathsf{acons}(Y, Z)), p_{\overline{L}}(X), q(U)$$

where q is a fresh predicate. These clauses are in $\mathcal{H}_1$, as guaranteed by [25].

We might have applied the standard approximation procedure of [25] directly on the clauses of Figure 9, however this would incur a loss of precision even on zone types where all sizes are known statically. The point in the abstraction above is that, in case no $\texttt{array}^*(zt)$ zone type appears in the program, then the standard approximation procedure is not needed, and no loss of precision is required.

Dealing with clauses in Figure 9 was the hardest case. The argument above shows that our analysis can be made decidable, and in exponential time in the worst case. The fact that most clauses can be made to inhabit $\mathcal{H}_3$ or $\mathsf{H}_2^{i,a}$ suggests that the resolution procedure of [25] will run in polynomial time on large parts of the clause set. In practice, running a resolution-based prover such as SPASS [27], although it may fail to terminate, is also a reasonable alternative.

Finally, observe that clause (17) is in $\mathcal{H}_3$, too, and that (18) is in $\mathcal{H}_3$ as soon as t is linear (i.e., no variable occurs more than once in t), and in $\mathsf{H}_2^{2,\max(a,2)}$ where a is the maximal arity of function symbols occurring in t.

Now look at the external trust model $\mathcal{M}$ and the trust assertions. First, note that any trust assertion $A \Leftarrow A_1, \ldots, A_n$ that is itself in $\mathcal{H}_3$ (considering program variables x as constants) yields clauses in $[\![A \Leftarrow A_1, \ldots, A_n]\!]^\sharp$ that are again in $\mathcal{H}_3$, and similarly for $\mathcal{H}_2$ and $\mathcal{H}_1$. Therefore at least all the clauses arising from trust assertions in the example of Figure 2 are in $\mathcal{H}_3$, with an i bound of 1 and an a bound of 2. As far as the external trust model is concerned, clauses (1) through (9), which are typical examples of clauses that we shall include in $\mathcal{M}$, are in $\mathcal{H}_3$ with an i bound of 2 and an a bound of 2, except for (8) with is in $\mathsf{H}_2^{2,2}$.

So, assuming that the size of every array is known, that t is linear in (18), and that all clauses in the external trust model $\mathcal{M}$ and arising from trust assertions are in $\mathcal{H}_3$ or in $\mathsf{H}_2^{i,a}$ (for some fixed i, a), we observe that we can check security and conformance of C programs in polynomial time.

In general, some clauses in the external trust model will not be in $\mathcal{H}_1$. Typical cases are clauses describing the behavior of trusted, honest agents sharing secrets with the analyzed program, such as clauses (10) or (11). It turns out that such non-$\mathcal{H}_1$ clauses can be safely approximated, in polynomial time, by $\mathcal{H}_1$ clauses [25, Proposition 3, Proposition 4], up to some loss of precision. Since this loss occurs in well-circumscribed portions of our Horn clause sets, we believe that it should not be too damageable in practice. In fact, past experience in the verification of cryptographic protocols suggests that this usually does not throw away any essential information [28].

6 Correctness

Let us show that our abstract semantics is correct with respect to our concrete semantics, using the framework of abstract interpretation.

The set $\mathcal{C}$ of concrete configurations of programs is $\mathcal{Q} \times Env \times Store \times \mathbb{P}(Zone \times Term_0) \times \mathbb{P}(Atom_0)$, where $\mathcal{Q}$ is the set of vertices of control-flow graphs in the program, $Env = \mathcal{V} \to \mathcal{V}al$, $Store = Addr_{val} \to Zone$, i.e., the set of tuples $q, \rho, \mu, \mathcal{R}, \mathcal{B}$ considered in Figure 5. The *concrete* lattice $L^\flat$ is defined as the powerset $\mathbb{P}(\mathcal{C})$, ordered by inclusion. This is, as usual, the lattice of the collecting semantics of our language.

On the abstract side, consider the set $L_0^\sharp$ of all finite or infinite sets of definite clauses on the language of Section 5. $L_0^\sharp$ is quasi-ordered by reverse logical implication: $S_1^\sharp \preceq S_2^\sharp$ if and only if $S_1^\sharp$ is implied by $S_2^\sharp$, i.e., every Herbrand model of $S_2^\sharp$ is a Herbrand model of $S_1^\sharp$. Note in particular that if $S_1^\sharp \subseteq S_2^\sharp$ then $S_1^\sharp \preceq S_2^\sharp$. Let $\approx$ be the equivalence generated by $\preceq$, i.e., $S_1^\sharp \approx S_2^\sharp$ if and only if $S_1^\sharp \preceq S_2^\sharp$ and $S_2^\sharp \preceq S_1^\sharp$. Then $L^\sharp = L_0^\sharp / \approx$ is a partial order, and in fact a complete lattice. Indeed, $L^\sharp$ has all joins: taking any family of sets $S_i^\sharp \in L_0^\sharp$, $i \in I$, $\bigvee_{i \in I} S_i^\sharp = \bigcup_{i \in I} S_i^\sharp$, and this construction is obviously compatible with $\approx$. (We need $L_0^\sharp$ to contain infinite clause sets precisely to enforce completeness.) Since $L^\sharp$ has all joins, it has all meets, too, as is well-known: $\bigwedge_{i \in I} S_i^\sharp = \bigvee_{S^\sharp / \forall i \in I . S^\sharp \preceq S_i^\sharp} S^\sharp$.

Let therefore the *abstract lattice* be $L^\sharp$. We shall silently equate elements of $L^\sharp$ with chosen representatives in $L_0^\sharp$, and reason up to $\approx$. Let the *definable elements* of $L^\sharp$ be the (equivalence classes of) *finite* clause sets. It is clear that our abstract semantics only produces definable elements of $L^\sharp$.

Let a *fact* be either a ground atom in $Atom_0$ (possible elements of $\mathcal{B}$) or a pair $\texttt{rec}(z, t)$, $z \in Zone$, $t \in Term_0$ (possible elements of $\mathcal{R}$). Given any set $S^\sharp$ of definite clauses (definable or not), $S^\sharp$ has a *least Herbrand model* $lhm(S^\sharp)$, which is a set of facts, the set of all facts that must be true in every model of $S^\sharp$. As a set of facts, $lhm(S^\sharp)$ is the set of all facts A that are deducible from the clauses in $S^\sharp$ by unit resolution and instantiation. It is clear that $lhm(S^\sharp)$ is in $L^\sharp$, and that $S_1^\sharp \approx S_2^\sharp$ if and only if $lhm(S_1^\sharp) = lhm(S_2^\sharp)$. In other words, $lhm(S^\sharp)$ is a *canonical representative*—usually infinite—of the equivalence class of $S^\sharp$.

It is more intuitive to reason with definable elements, i.e., finite clause sets, in an implementation. To establish a correctness result, it is more practical to reason on Herbrand models instead. Then, $S_1^\sharp \preceq S_2^\sharp$ if and only if the least Herbrand model of $S_2^\sharp$ is a Herbrand model of $S_1^\sharp$, i.e., if and only if $lhm(S_1^\sharp) \subseteq lhm(S_2^\sharp)$. In other words, up to isomorphism, $L^\sharp$ is just the set $\mathbb{P}(Fact_0)$, where $Fact_0$ is the set of facts, ordered by inclusion $\subseteq$. Observe that any set of facts can be split into a pair R, B, where R is the set of facts of the form $\texttt{rec}(z, t)$, and B is the set of ground

atoms in it. So $Fact_0$ is the disjoint union $(Zone \times Term_0) \uplus Atom_0$, whence $\mathbb{P}(Fact_0)$ is isomorphic to $\mathbb{P}(Zone \times Term_0) \times \mathbb{P}(Atom_0)$. So, up to isomorphism again, $L^\sharp$ is the set $\mathbb{P}(Zone \times Term_0) \times \mathbb{P}(Atom_0)$ of pairs R, B, ordered by pairwise inclusion $\subseteq^2$.

$$\frac{\alpha : L^\flat \to L^\sharp}{\alpha(C^\flat) = \bigcup_{(q,\rho,\mu,\mathcal{R},\mathcal{B}) \in C^\flat} \alpha_{cfg}(q,\rho,\mu,\mathcal{R},\mathcal{B})}$$

$$\frac{\alpha_{cfg} : \mathcal{C} \to L^\sharp}{\alpha_{cfg}(q,\rho,\mu,\mathcal{R},\mathcal{B}) = (\alpha_{rec}(\mathcal{R}), \mathcal{B} \cup \alpha_{env}(\rho) \cup \alpha_{store}(\mu))}$$

$$\frac{\alpha_{env} : Env \to L^\sharp}{\alpha_{env}(\rho) = \{\mathsf{val}(\mathsf{c}_x, t) | x \in \mathrm{dom}\ \rho, t \in \alpha_{val}(\rho(x))\}}$$

$$\frac{\alpha_{store} : Store \to L^\sharp}{\alpha_{store}(\mu) = \{\mathsf{p}(t,t') | \ell \in Loc, \hat{\mu}(\ell) \text{ is defined}, t \in \alpha_{loc}(\ell), t' \in \alpha_{zone}(\hat{\mu}(\ell))\}}$$

$$\frac{\alpha_{val} : \mathcal{V}val \to \mathbb{P}(Term_0)}{\begin{array}{l}\alpha_{val}(\texttt{int}\ n) = \{\texttt{int}\} \\ \alpha_{val}(\texttt{float}\ r) = \{\texttt{float}\} \\ \alpha_{val}(\texttt{ptr}\ \ell) = \{\texttt{ptr}(t) \\ \qquad\qquad |t \in \alpha_{loc}(\ell)\}\end{array}}$$

$$\frac{\alpha_{rec} : \mathbb{P}(Zone \times Term_0) \to L^\sharp}{\alpha_{rec}(\mathcal{R}) = \{\texttt{rec}(t,t') | (z,t') \in \mathcal{R}, t \in \alpha_{zone}(z)\}}$$

$$\frac{\alpha_{zone} : Zone \to \mathbb{P}(Term_0)}{\begin{array}{l}\alpha_{zone}(z) = \alpha_{val}(z) \text{ if } z \in \mathcal{V}al \\ \alpha_{zone}(\texttt{struct}\ \{a_1 = z_1, \ldots, a_n = z_n\}) = \\ \quad \{\texttt{struct}\ \{a_1 = t_1, \ldots, a_n = t_n\} \\ \qquad |t_1 \in \alpha_{zone}(z_1), \ldots, t_n \in \alpha_{zone}(z_n)\} \\ \alpha_{zone}(\texttt{array}(z_1, \ldots, z_n)) = \\ \quad \{\mathsf{acons}(t_1, \ldots \mathsf{acons}(t_n, \mathsf{anil}) \ldots) \\ \qquad |t_1 \in \alpha_{zone}(z_1), \ldots, t_n \in \alpha_{zone}(z_n)\}\end{array}}$$

$$\frac{\alpha_{loc} : Loc \to \mathbb{P}(Term_0)}{\begin{array}{l}\alpha_{loc}(a) = \{s \in \mathcal{S} | a \in Addr_{val}[s]\} \\ \qquad \text{if } a \in Addr_{val} \\ \alpha_{loc}(a) = \{\texttt{inval}\} \text{ if } a \in Addr_{inval} \\ \alpha_{loc}(\mathsf{fun}\ f) = \{\mathsf{fun}_f\} \\ \alpha_{loc}(\ell.\langle lab, i\rangle) = \\ \qquad\qquad \{\mathsf{ssel}_{lab,i}(t) | t \in \alpha_{loc}(\ell)\} \\ \alpha_{loc}(\ell.n) = \{\mathsf{asel}^n(t) | t \in \alpha_{loc}(\ell)\} \\ \alpha_{loc}(\ell.\Diamond) = \{\Diamond(t) | t \in \alpha_{loc}(\ell)\}\end{array}}$$

Fig. 10. The abstraction functions

Define a Galois connection $\alpha \dashv \gamma$ between $\alpha : L^\flat \to L^\sharp$ and $\gamma : L^\sharp \to L^\flat$ as follows. Remember that it is equivalent to define γ and check that γ preserves meets, or to define α and check that α preserves joins. In each case the other component is defined uniquely so that $\alpha \dashv \gamma$. We define α, see Figure 10. It is clear that α preserves joins, because $\alpha(C^\flat)$ is defined as a union of sets, one for each element of $C^\flat$. (In particular, α is monotonic.)

Given the least Herbrand model $lhm(S) \in L^\sharp$ of the set S of clauses given by the abstract semantics of Section 5 on a given program π, saying that a configuration $q, \rho, \mu, \mathcal{R}, \mathcal{B}$ is correct w.r.t. $lhm(S)$ means that $q, \rho, \mu, \mathcal{R}, \mathcal{B}$ is in the semantics $\gamma(lhm(S))$, or equivalently, that $\alpha(\{(q, \rho, \mu, \mathcal{R}, \mathcal{B})\}) \subseteq^2 lhm(S)$, by standard properties of Galois connections. By the definition of α, it is equivalent to state that $\alpha_{cfg}(q, \rho, \mu, \mathcal{R}, \mathcal{B}) \subseteq^2 lhm(S)$, i.e., to the fact that S implies all facts in $\alpha_{cfg}(q, \rho, \mu, \mathcal{R}, \mathcal{B})$. Correctness states that this property is preserved under any execution of the program:

Theorem 1 (Correctness) *Let π be a program, S be the set of clauses given by the abstract semantics of π.*

If $q, \rho, \mu, \mathcal{R}, \mathcal{B} \xrightarrow{i} q', \rho', \mu', \mathcal{R}', \mathcal{B}'$ (resp. $q, \rho, \mu, \mathcal{R}, \mathcal{B} \longrightarrow^* q', \rho', \mu', \mathcal{R}', \mathcal{M}'$) and S implies $\alpha_{cfg}(q, \rho, \mu, \mathcal{R}, \mathcal{B})$ then S implies $\alpha_{cfg}(q', \rho', \mu', \mathcal{R}', \mathcal{B}')$.

PROOF. By induction over derivations, as given by the axioms and rules of Figure 5.

If i is $x := y$, then $\rho' = \rho[x \mapsto \rho(y)]$, $\mu' = \mu$, $\mathcal{R}' = \mathcal{R}$, $\mathcal{B}' = \mathcal{B}$. So $\alpha_{cfg}(q', \rho', \mu', \mathcal{R}', \mathcal{B}') \subseteq \alpha_{cfg}(q, \rho, \mu, \mathcal{R}, \mathcal{B}) \cup^2 (\emptyset, \{\mathsf{val}(\mathsf{c}_x, t) | t \in \alpha_{val}(\rho(y))\})$. We have to check that S implies the extra atoms $\mathsf{val}(\mathsf{c}_x, t)$ for every $t \in \alpha_{val}(\rho(y))$. Since S implies $\alpha_{cfg}(q, \rho, \mu, \mathcal{R}, \mathcal{B})$, S implies $\mathsf{val}(\mathsf{c}_y, t)$ for every $t \in \alpha_{val}(\rho(y))$, in particular. Since S contains the clause $\mathsf{val}(\mathsf{c}_x, X) \Leftarrow \mathsf{val}(\mathsf{c}_y, X)$ by the definition of $[\![x := y]\!]^\sharp$, S indeed implies $\mathsf{val}(\mathsf{c}_x, t)$ for every $t \in \alpha_{val}(\rho(y))$.

The cases $x :=$ `int` n, $x :=$ `float` r, $x := f$ are equally easy. When i is $x := *y$, we need to check that S implies the extra atoms $\mathsf{val}(\mathsf{c}_x, t')$ for every $t' \in \alpha_{val}(\hat{\mu}(\ell))$, where $\rho(y) =$ `ptr` ℓ. Since $\rho(y) =$ `ptr` ℓ, we obtain that S implies $\mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(t))$ for every $t \in \alpha_{loc}(\ell)$. Since $\alpha_{loc}(\ell)$ is not empty (an easy check), fix some $t \in \alpha_{loc}(\ell)$. We obtain (a): S implies $\mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(t))$. On the other hand, by definition, $\alpha_{store}(\mu)$ contains all atoms $\mathsf{p}(t, t')$ where $t \in \alpha_{loc}(\ell)$ and $t' \in \alpha_{zone}(\hat{\mu}(\ell))$. Since $\hat{\mu}(\ell) \in \mathcal{V}val$, $\alpha_{zone}(\hat{\mu}(\ell)) = \alpha_{val}(\hat{\mu}(\ell))$. Since S implies $\alpha_{store}(\mu)$, we obtain (b): S also implies all atoms $\mathsf{p}(t, t')$, $t' \in \alpha_{val}(\hat{\mu}(\ell))$. By definition of $[\![x := *y]\!]^\sharp$, (a), and (b), S must also imply $\mathsf{val}(\mathsf{c}_x, t')$ for every $t' \in \alpha_{val}(\hat{\mu}(\ell))$, whence the claim.

When i is $*x := y$, let $\rho(x)$ be `ptr` ℓ, μ' be $\mu[\ell := \rho(y)]$. Using the definition of α_{store}, we must check that S implies all atoms $\mathsf{p}(t, t')$, where $t \in \alpha_{loc}(\ell_1)$, ℓ_1 ranges over locations such that $\widehat{\mu'}(\ell_1)$ is defined, and $t' \in \alpha_{zone}(\widehat{\mu'}(\ell_1))$. Let $\sqsubseteq$ be the prefix ordering on locations, and $\sqsubset$ be its strict part. There are three cases to consider: $\ell_1 \sqsubseteq \ell$, $\ell \sqsubset \ell_1$, and ℓ and ℓ_1 incomparable. The case $\ell \sqsubset \ell_1$ is impossible, because $\widehat{\mu'}(\ell_1)$ is a variable value (in $\mathcal{V}val$), i.e., of the form `int` n, `float` r, or `ptr` ℓ', and there is simply no case where either $\widehat{\mu'}(\ell.\langle lab, i\rangle)$, $\widehat{\mu'}(\ell.j)$, or $\widehat{\mu'}(\ell.\Diamond)$ would be defined. In the case where ℓ and ℓ_1 are incomparable, then $\widehat{\mu'}(\ell_1) = \hat{\mu}(\ell_1)$, so by assumption S implies all atoms $\mathsf{p}(t, t')$, $t \in \alpha_{loc}(\ell_1)$, $t' \in \alpha_{zone}(\hat{\mu}(\ell_1)) = \alpha_{zone}(\widehat{\mu'}(\ell_1))$, and we are done. Finally, if $\ell_1 \sqsubseteq \ell$, we show that S implies all atoms $\mathsf{p}(t, t')$ where

$t \in \alpha_{loc}(\ell_1)$, $t' \in \alpha_{zone}(\widehat{\mu'}(\ell_1))$, by induction over $|\ell| - |\ell_1|$, where $|\ell|$ denotes the length of ℓ:

- Base case: $\ell_1 = \ell$. Since $\rho(x) = \mathtt{ptr}\ \ell$, S implies $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(t))$ for every $t \in \alpha_{loc}(\ell)$. By assumption, S implies every atom $\mathsf{val}(\mathsf{c}_y, t')$ for every $t' \in \alpha_{val}(\rho(y))$. Using the definition of $[\![*x := y]\!]^\sharp$, we infer that S also implies $\mathsf{p}(t, t')$ for every $t \in \alpha_{loc}(\ell) = \alpha_{loc}(\ell_1)$, and for every $t' \in \alpha_{val}(\rho(y))$. An easy check shows that, since $\mu' = \mu[\ell := \rho(y)]$, $\widehat{\mu'}(\ell) = \rho(y)$. So $\alpha_{zone}(\widehat{\mu'}(\ell_1)) = \alpha_{zone}(\widehat{\mu'}(\ell)) = \alpha_{zone}(\rho(y)) = \alpha_{val}(\rho(y))$, because $\rho(y) \in \mathcal{V}al$. Whence S also implies $\mathsf{p}(t, t')$ for every $t \in \alpha_{loc}(\ell_1)$, and for every $t' \in \alpha_{zone}(\widehat{\mu'}(\ell_1))$.
- First induction case: there is a prefix $\ell_2 \sqsubseteq \ell$ such that $\ell_2 = \ell_1.\langle lab, i\rangle$. By induction hypothesis, S implies every atom $\mathsf{p}(u, u')$, $u \in \alpha_{loc}(\ell_2)$, $u' \in \alpha_{zone}(\widehat{\mu'}(\ell_2))$. Expanding definitions, S must imply every atom of the form $\mathsf{p}(\mathsf{ssel}_{lab,i}(t), u')$, $t \in \alpha_{loc}(\ell_1)$, $u' \in \alpha_{zone}(\widehat{\mu'}(\ell_2))$. Since $\widehat{\mu'}$ is defined, in particular $\hat{\mu}(\ell_2)$ is defined, which entails that $\hat{\mu}(\ell_1)$ is of the form $\mathtt{struct}\ \{a_1 = z_1, \ldots, a_n = z_n\}$, for some zone type $\mathtt{struct}\ \{a_1 = zt_1, \ldots, a_n = zt_n\}$, with $1 \leq i \leq n$ and $lab = a_i$. By definition of α_{store}, S must therefore also imply all atoms of the form $\mathsf{p}(t, \mathtt{struct}\ \{a_1 = t'_1, \ldots, a_n = t'_n\})$, with $t \in \alpha_{loc}(\ell_1)$, and where $t'_1 \in \alpha_{zone}(z_1)$, …, $t'_n \in \alpha_{zone}(z_n)$. Using (12) (Figure 9), S must also imply $\mathsf{p}(t, \{a_1 = t'_1, \ldots, a_i = u', \ldots, a_n = t'_n\})$, for any $t \in \alpha_{loc}(\ell_1)$, $u' \in \alpha_{zone}(\widehat{\mu'}(\ell_2))$, $t'_1 \in \alpha_{zone}(z_1)$, …, $t'_n \in \alpha_{zone}(z_n)$. It follows that S implies $\mathsf{p}(t, t')$ for every $t \in \alpha_{loc}(\ell_1)$ and $t' \in \alpha_{zone}(\widehat{\mu'}(\ell_1))$.
- Second induction case: for some prefix $\ell_2 \sqsubseteq \ell$, $\ell_2 = \ell_1.j$. The argument is similar, but needs to be explained in full. By induction hypothesis, S implies every atom $\mathsf{p}(u, u')$, $u \in \alpha_{loc}(\ell_2)$, $u' \in \alpha_{zone}(\widehat{\mu'}(\ell_2))$. Expanding definitions, S must imply every atom of the form $\mathsf{p}(\mathsf{asel}^j(t), u')$, $t \in \alpha_{loc}(\ell_1)$, $u' \in \alpha_{zone}(\widehat{\mu'}(\ell_2))$. Since $\widehat{\mu'}$ is defined, in particular $\hat{\mu}(\ell_2)$ is defined, which entails that $\hat{\mu}(\ell_1)$ is of the form $\mathtt{array}(z_1, \ldots, z_n)$, with $0 \leq j \leq n$. By definition of α_{store}, S must therefore also imply all atoms of the form $\mathsf{p}(t, \mathsf{acons}(t'_1, \mathsf{acons}(\ldots, \mathsf{acons}(t'_n, \mathsf{anil}))))$, with $t \in \alpha_{loc}(\ell_1)$, and where $t'_1 \in \alpha_{zone}(z_1)$, …, $t'_n \in \alpha_{zone}(z_n)$. Using (14) as many times as needed, we obtain (a): S implies $\mathsf{p}(\mathsf{asel}^k(t), \mathsf{acons}(t'_{k+1}, \mathsf{acons}(\ldots \mathsf{acons}(t'_n, \mathsf{anil}))))$ for every k, $0 \leq k \leq n$. Now S also implies $\mathsf{p}(\mathsf{asel}^j(t), u')$, so using (a) with $k = j - 1$ and (13), S implies $\mathsf{p}(\mathsf{asel}^{j-1}(t), \mathsf{acons}(t'_j, u'))$. Using (a) again with $k = j - 2$ and (13) again, S implies $\mathsf{p}(\mathsf{asel}^{j-2}(t), \mathsf{acons}(t'_{j-1}, \mathsf{acons}(t'_j, u')))$. Iterating this construction for $k = j - 3$, …, $k = 0$, we eventually obtain that S implies $\mathsf{p}(t, \mathsf{acons}(t'_1, \mathsf{acons}(\ldots \mathsf{acons}(t'_j, u'))))$. The claim follows.
- Third induction case: for some prefix $\ell_2 \sqsubseteq \ell$, $\ell_2 = \ell_1.\Diamond$. By induction hypothesis, S implies every atom $\mathsf{p}(u, u')$, $u \in \alpha_{loc}(\ell_2)$, $u' \in \alpha_{zone}(\widehat{\mu'}(\ell_2))$. Expanding definitions, S must imply every atom of the form $\mathsf{p}(\Diamond(t), u')$, $t \in \alpha_{loc}(\ell_1)$, $u' \in \alpha_{zone}(\widehat{\mu'}(\ell_2))$. Since $\widehat{\mu'}$ is defined, in particular $\hat{\mu}(\ell_2)$ is defined, which entails that $\hat{\mu}(\ell_1)$ is of the form $\mathtt{array}(z_1, \ldots, z_n)$, for some $n \geq 1$. By definition of α_{store}, S must therefore also imply all atoms of the form $\mathsf{p}(t, \mathsf{acons}(t'_1, \mathsf{acons}(\ldots \mathsf{acons}(t'_n, \mathsf{anil}))))$, with $t \in \alpha_{loc}(\ell_1)$, and where $t'_1 \in \alpha_{zone}(z_1)$, …, $t'_n \in \alpha_{zone}(z_n)$. Using (15), S must imply $\mathsf{p}(t, \mathsf{acons}(u', \mathsf{acons}(\ldots \mathsf{acons}(t'_n, \mathsf{anil}))))$, whence the claim.

When i is $x := \&y[z]$, we need to check that S implies $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(\mathsf{asel}^j(t)))$ for every $t \in \alpha_{loc}(\ell)$, where $\rho(y) = \mathtt{ptr}\ \ell$. In particular, we already know that S implies $\mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(t))$ for every $t \in \alpha_{loc}(\ell)$. If $[\![z]\!]^\sharp_{int} \neq \top$, so that it is a finite set of natural numbers, by our assumption that the $[\![\_]\!]^\sharp_{int}$ abstraction is correct, necessarily $j \in [\![z]\!]^\sharp_{int}$. So, by definition of $[\![x := \&y[z]]\!]^\sharp$, we use the clause $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(\mathsf{asel}^j(Y))) \Leftarrow \mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(Y))$ to conclude that S must also imply $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(\mathsf{asel}^j(t)))$ for every $t \in \alpha_{loc}(\ell)$. If $[\![z]\!]^\sharp_{int} = \top$, then by using the two clauses in the definition of $[\![x := \&y[z]]\!]^\sharp$, we infer $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(\mathsf{asel}^j(Y))) \Leftarrow \mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(Y))$ for every $j \in \mathbb{N}$, whence the claim again.

When i is $x := \&\Diamond y$, we need to check that S implies $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(\Diamond(t)))$ for every $t \in \alpha_{loc}(\ell)$, where $\rho(y) = \mathtt{ptr}(\ell)$. Since $\rho(y) = \mathtt{ptr}(\ell)$, S implies $\mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(t))$ for every $t \in \alpha_{loc}(\ell)$. The claim follows easily by the definition of $[\![x := \&\Diamond y]\!]^\sharp$.

When i is $x := \&y \xrightarrow{j} a$, we need to check that S implies $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(\mathsf{ssel}_{a,j}(t)))$ for every $t \in \alpha_{loc}(\ell)$, where $\rho(y) = \mathtt{ptr}\ \ell$. As S implies $\mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(t))$ for every $t \in \alpha_{loc}(\ell)$, the claim again follows easily.

When i is of the form $x := op(x_1, \ldots, x_n)$, we need to check that S implies $\mathsf{val}(\mathsf{c}_x, t)$ for every $t \in \alpha_{val}(\widehat{op}(\rho(x_1), \ldots, \rho(x_n)))$. Recall that each primitive op has a unique return type τ_{op}, either `int` or `float`. In the first case, t is necessarily `int` and the definition of $[\![x := op(x_1, \ldots, x_n)]\!]^\sharp$ contains the clause $\mathsf{val}(\mathsf{c}_x, \mathtt{int})$. Similarly for `float`.

When i is of the form $x := \mathtt{new}_s\ zt$, we need to check that S implies both $\mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(t))$, for every $t \in \alpha_{loc}(a)$, where $a \in Addr_{val}[s]$, and $\mathsf{p}(t, t')$ for every $t \in \alpha_{loc}(a)$ and $t' \in \alpha_{zone}(z)$, where z is any zone such that $z :\prec zt$. Now the only element of $\alpha_{loc}(a)$ is s, so $\mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(t))$ is just $\mathsf{val}(\mathsf{c}_y, \mathtt{ptr}(s))$ for every $t \in \alpha_{loc}(a)$, and this is one of the elements of $[\![x := \mathtt{new}_s\ zt]\!]^\sharp$. On the other hand, it is easy to check, by structural induction on zt', that for every zone z' such that $z' :\prec zt'$, for every $t' \in \alpha_{zone}(z')$, $\mathtt{init}_{zt'}(t')$ is deducible from the clauses of Figure 8. For $zt' = zt$ and $z' = z$, it follows that S implies $\mathtt{init}_{zt}(t')$ for every $t' \in \alpha_{zone}(z)$. Using the second clause of the definition of $[\![x := \mathtt{new}_s\ zt]\!]^\sharp$, it follows that S also implies $\mathsf{p}(s, t')$. Since the only element of $\alpha_{loc}(a)$ is s, S implies $\mathsf{p}(t, t')$ for every $t \in \alpha_{loc}(a)$ and $t' \in \alpha_{zone}(z)$.

When i is of the form $?x == 0$ or $?x! = 0$, the claim is obvious. When i is of the form $x := \mathtt{call}\ g(y)$, by induction hypothesis (refer to Figure 5 for notations), we obtain ρ', μ', $\mathcal{R}'$, and $\mathcal{B}'$ such that if (a): S implies $\alpha_{cfg}(I(G_g), \rho_{|\mathcal{V}_{\mathrm{glob}}}[in_g \mapsto \rho(y)], \mu, \mathcal{R}, \mathcal{B})$ then (b): S also implies $\alpha_{cfg}(O(G_g), \rho', \mu', \mathcal{R}', \mathcal{B}')$. Assume that (c): S implies $\alpha_{cfg}(q, \rho, \mu, \mathcal{R}, \mathcal{B})$. S contains the clause $\mathsf{val}(\mathtt{in}(\mathsf{fun}_g), Y) \Leftarrow \mathsf{val}(\mathsf{c}_y, Y)$ by definition of $[\![x := \mathtt{call}\ g(y)]\!]^\sharp$. Since clearly S implies $\mathsf{val}(\mathsf{c}_y, t)$ for every $t \in \alpha_{val}(\rho(y))$, S implies $\mathsf{val}(\mathtt{in}(\mathsf{fun}_g), t)$ for every $t \in \alpha_{val}(\rho(y))$. Recall that $\mathsf{c}_{in_g} = \mathtt{in}(\mathsf{fun}_g)$ by definition, so S implies $\mathsf{val}(\mathsf{c}_{in_g}, t)$ for every $t \in \alpha_{val}(\rho(y))$. Using (c), it follows that S implies $\alpha_{cfg}(I(G_g), \rho[in_g \mapsto \rho(y)], \mu, \mathcal{R}, \mathcal{B})$ by definition of α_{cfg},

and therefore that it implies any smaller subset of facts, in particular (a) holds. So (b) holds, too. By (b) and (c), S implies the union of $\alpha_{cfg}(O(G_g), \rho', \mu', \mathcal{R}', \mathcal{B}')$ and $\alpha_{cfg}(q, \rho, \mu, \mathcal{R}, \mathcal{B})$, hence also any subset; in particular, (d): S implies $\alpha_{cfg}(q', (\rho \oplus \rho'_{|\mathcal{V}_{\text{glob}}}), \mu', \mathcal{R}', \mathcal{B}')$. Since $out_g \in \mathcal{V}_{\text{glob}}$, and since we have assumed out_g will always have received a value when returning from g, so that $out_g \in \text{dom}\ \rho'$, it follows that $out_g \in \text{dom}\rho'_{|\mathcal{V}_{\text{glob}}}$. From (d), we therefore obtain (e): S must imply $\mathsf{val}(\mathsf{c}_{out_g}, t')$ for every $t' \in \alpha_{val}(\rho'(out_g))$. Because S contains the clause $\mathsf{val}(\mathsf{c}_x, X) \Leftarrow \mathsf{val}(\mathtt{out}(\mathsf{fun}_g), X)$ by definition of $[\![x := \mathtt{call}\ g(y)]\!]^\sharp$, since $\mathsf{c}_{out_g} = \mathtt{out}(\mathsf{fun}_g)$ and by (e), S must imply $\mathsf{val}(\mathsf{c}_x, t')$ for every $t' \in \alpha_{val}(\rho'(out_g))$. So, by definition of α_{cfg} and using (d), S implies $\alpha_{cfg}(q', (\rho \oplus \rho'_{|\mathcal{V}_{\text{glob}}})[x \mapsto \rho'(out_g)], \mu', \mathcal{R}', \mathcal{B}')$. The case when i is $x := \mathtt{call}{*}\ z(y)$ is entirely similar.

Finally, when i is a trust assertion $\mathtt{trust}\ A \Leftarrow A_1, \ldots, A_n$, we must show that S implies all facts in $\alpha_{rec}(\mathcal{R}')$ and $\mathcal{B}'$, where

$$(\mathcal{R}', \mathcal{B}') = \text{lfp}\ T^{\rho,\mu}_{\mathcal{M}}((\mathcal{R}, \mathcal{B}) \cup^2 T^{\rho,\mu}_{A \Leftarrow A_1,\ldots,A_n}(\mathcal{R}, \mathcal{B}))$$

In general, let $(\mathcal{R}_0, \mathcal{B}_0)$ be any set of facts such that S implies $(\alpha_{rec}(\mathcal{R}_0), \mathcal{B}_0)$, assume that S implies $\alpha_{env}(\rho)$ and $\alpha_{store}(\mu)$, and let $A^0 \Leftarrow A^0_1, \ldots, A^0_n$ be any clause in S.

- We claim that (a): if $\rho, \mu, \mathcal{R}_0, \mathcal{B}_0 \models A'\sigma$, for any atom A' such that $A'\sigma$ is ground, then S union any clause $C \vee A'^-$ implies C. If A' is of the form $x\ \mathtt{rec}\ t$, then $\rho, \mu, \mathcal{R}_0, \mathcal{B}_0 \models A'\sigma$ means that $\rho(x) = \mathtt{ptr}\ \ell$ for some location ℓ, $z = \hat{\mu}(\ell)$ is defined and $(z, t) \in \mathcal{R}_0$. It follows that for all $t' \in \alpha_{loc}(\ell)$, S implies (a_1): $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(t'))$; since $\alpha_{loc}(\ell)$ is not empty, we may fix such a $t' \in \alpha_{loc}(\ell)$. From $z = \hat{\mu}(\ell)$ and the definition of α_{store}, S must imply (a_2): $\mathsf{p}(t', t'')$ for any $t'' \in \alpha_{zone}(z)$; again, since $\alpha_{zone}(z)$ is never empty, fix such a t''. From $(z, t) \in \mathcal{R}_0$ and the definition of α_{rec}, S implies (a_3): $\mathtt{rec}(t'', t)$. Using (a_1), (a_2), (a_3) together with $C \vee A'^-$, namely $C \vee (\bot \Leftarrow \mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(X)), \mathsf{p}(X, Z), \mathtt{rec}(Z, t))$ instantiated by $X = t'$, $Z = t''$, we infer C. If A' is of any other form, then $\rho, \mu, \mathcal{R}_0, \mathcal{B}_0 \models A'\sigma$ means that $A'\sigma \in \mathcal{B}_0$, so $A'\sigma$ is implied by S. Using $C \vee A'^-$, namely $C \vee (\bot \Leftarrow A')$, we infer C.
- We then claim that (b): S implies all facts A' in $(\alpha_{rec}(\mathcal{R}_1), \mathcal{B}_1)$, where $(\mathcal{R}_1, \mathcal{B}_1) = T^{\rho,\mu}_{A^0 \Leftarrow A^0_1,\ldots,A^0_n}(\mathcal{R}_0, \mathcal{B}_0)$. Indeed, take any such fact A'.

  If A' is in $\alpha_{rec}(\mathcal{R}_1)$, then A' is of the form $\mathtt{rec}(t, u)$, where $t \in \alpha_{zone}(z)$ for some zone z such that $(z, u) \in \mathcal{R}_1$. By definition of $\mathcal{R}_1$, A^0 must be of the form $x\ \mathtt{rec}\ t'$, where $\rho(x) = \mathtt{ptr}\ \ell$ for some location ℓ, $\hat{\mu}(\ell)$ is defined, $z = \hat{\mu}(\ell)$, and $t' = u$. Also, $\rho, \mu, \mathcal{R}, \mathcal{B} \models A^0_i\sigma$ for every i, $1 \leq i \leq n$. By (a) and since S contains $A^{0+} \vee A^{0-}_1 \vee A^{0-}_2 \vee \ldots \vee A^{0-}_n$, S entails $A^{0+} \vee A^{0-}_2 \vee \ldots \vee A^{0-}_n$, so using (a) again and iterating, S entails A^{0+}. Now A^{0+} is $\mathtt{rec}(Z, u) \Leftarrow \mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(X)), \mathsf{p}(X, Z)$. Take any $t'' \in \alpha_{loc}(\ell)$ (which is not empty), since $\rho(x) = \mathtt{ptr}\ \ell$ we get that S implies $\mathsf{val}(\mathsf{c}_x, \mathtt{ptr}(t''))$. Since $z = \hat{\mu}(\ell)$, $t \in \alpha_{zone}(z)$, and $t'' \in \alpha_{loc}(\ell)$, S also implies $\mathsf{p}(t'', t)$. Plugging t'' for X and t for Z into A^{0+}, we obtain that S implies $\mathtt{rec}(t, u)$, i.e., A'.

If A' is in $\mathcal{B}_1$, then A' is a ground instance $A^0\sigma$ of A^0. By (a) and since S contains $A^{0+} \vee A_1^{0-} \vee A_2^{0-} \vee \ldots \vee A_n^{0-}$, S entails $A^{0+} \vee A_2^{0-} \vee \ldots \vee A_n^{0-}$, so using (a) again and iterating, S entails A^{0+}. Since $A^{0+} = A^0$, S also implies A^0, hence its ground instance A'.

Now take $\mathcal{R}_0 = \mathcal{R}$, $\mathcal{B}_0 = \mathcal{B}$. By (b) S implies all facts in $(\alpha_{rec}(\mathcal{R}_1), \mathcal{B}_1)$, where $(\mathcal{R}_1, \mathcal{B}_1) = T^{\rho,\mu}_{A \Leftarrow A_1,\ldots,A_n}(\mathcal{R}, \mathcal{B})$. Since S also implies all facts in $(\alpha_{rec}(\mathcal{R}), \mathcal{B})$ and abstractions preserve joins, S implies all facts in $(\alpha_{rec}(\mathcal{R}_2), \mathcal{B}_2)$, where $(\mathcal{R}_2, \mathcal{B}_2) = (\mathcal{R}, \mathcal{B}) \cup^2 T^{\rho,\mu}_{A \Leftarrow A_1,\ldots,A_n}(\mathcal{R}, \mathcal{B})$. By (b) again, S is stable under applications of the $T^{\rho,\mu}_{\mathcal{M}}$ operator, in the sense that for any set of facts $(\mathcal{R}_j, \mathcal{B}_j)$ such that S implies $(\alpha_{rec}(\mathcal{R}_j), \mathcal{B}_j)$, S also implies $(\alpha_{rec}(\mathcal{R}_{j+1}), \mathcal{B}_{j+1})$, where $(\mathcal{R}_{j+1}, \mathcal{B}_{j+1}) = (\mathcal{R}_j, \mathcal{B}_j) \cup^2 T^{\rho,\mu}_{\mathcal{M}}(\mathcal{R}_j, \mathcal{B}_j)$. Since abstractions preserve joins, S implies $(\alpha_{rec}(\mathcal{R}_\infty, \mathcal{B}_\infty))$, where $(\mathcal{R}_\infty, \mathcal{B}_\infty)$ is the union of the sets $(\mathcal{R}_j, \mathcal{B}_j)$, $j \geq 2$. We conclude: $(\mathcal{R}_\infty, \mathcal{B}_\infty)$ is precisely $\mathrm{lfp}\ T^{\rho,\mu}_{\mathcal{M}}((\mathcal{R}, \mathcal{B}) \cup^2 T^{\rho,\mu}_{A \Leftarrow A_1,\ldots,A_n}(\mathcal{R}, \mathcal{B}))$. □

7 Implementation

We have implemented a variant of this abstract semantics in the CSur project [29]. Although this variant does not work exactly as specified here, we wish to report on it, as an example as how this abstract semantics may be implemented.

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, as well as running a static analysis $[\![\_]\!]^\sharp_{int}$ over integer domains). This follows the Compile-Link-Analysis technique of Heintze and Tardieu [17].

For each function, a control-flow graph is generated and the compiler collects types for each variable of programs. For all types, corresponding zone types are also computed. Finally a linker merges all control-flow graphs and zone types into a unique table. In the same way a library manager `csur_ar` (used just like `ar`) is 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 [27,30] or the first author's prototype `h1` prover [31]. The fact that most clauses are in $\mathcal{H}_3$, a polynomial class, and some others are in $\mathsf{H}_2^{i,a}$, 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.

8 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. 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] J. Goubault-Larrecq (Ed.), Special Issue on Models and Methods for Cryptographic Protocol Verification, Vol. 4, Instytut Łącsności (Institute of Telecommunications), Warsaw, Poland, 2002.

[2] R. M. Needham, M. D. Schroeder, Using encryption for authentication in large networks of computers, Communications of the ACM 21 (12) (1978) 993–999.

[3] G. Lowe, An attack on the Needham-Schroeder public-key authentication protocol, Information Processing Letters 56 (3) (1995) 131–133.
URL `citeseer.nj.nec.com/lowe95attack.html`

[4] A. Freier, P. Karlton, P. Kocher, The SSL protocol. Version 3.0, `http://home.netscape.com/eng/ssl3/` (1996).

[5] O. Gay, Exploitation avancée de buffer overflows, Tech. rep., LASEC, Ecole Polytechnique Fédérale de Lausanne, `http://diwww.epfl.ch/~ogay/advbof/advbof.pdf` (Jun. 2002).

[6] D. Wagner, J. S. Foster, E. A. Brewer, A. Aiken, A first step towards automated detection of buffer overrun vulnerabilities, in: Network and Distributed System Security Symposium, San Diego, CA, 2000, pp. 3–17.
URL `citeseer.nj.nec.com/wagner00first.html`

[7] A. Simon, A. King, Analyzing string buffers in C, in: Intl. Conf. on Algebraic Methods and Software Methodology (AMAST'2002), 2002, pp. 365–379.

[8] N. El Kadhi, Automatic verification of confidentiality properties of cryptographic programs, Networking and Information Systems 3 (6).

[9] P. Boury, N. El Khadi, Static analysis of Java cryptographic applets, in: ECOOP'2001 Workshop on Formal Techniques for Java Programs, Fern Universität Hagen, 2001.

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

[11] H. Comon-Lundh, V. Cortier, Security properties: Two agents are sufficient, in: Proc. 12th European Symposium on Programming (ESOP'2003), Springer-Verlag LNCS 2618, Warsaw, Poland, 2003, pp. 99–113.

[12] B. Blanchet, An efficient cryptographic protocol verifier based on Prolog rules, in: 14th IEEE Computer Security Foundations Workshop (CSFW-14), IEEE Computer Society, Cape Breton, Nova Scotia, Canada, 2001, pp. 82–96.

[13] N. A. Durgin, P. D. Lincoln, J. C. Mitchell, A. Scedrov, Multiset rewriting and the complexity of bounded security protocols, Journal of Computer Security 12 (2) (2004) 247–311.

[14] R. M. Amadio, W. Charatonik, On name generation and set-based analysis in the Dolev-Yao model, in: 13th International Conference on Concurrency Theory (CONCUR'02), Springer-Verlag LNCS 2421, 2002, pp. 499–514.

[15] S. Sagiv, T. W. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, ACM Trans. Prog. Lang. Sys. 24 (3) (2002) 217–298.

[16] L. O. Andersen, Program analysis and specialization for the C programming language, Ph.D. thesis, DIKU, University of Copenhagen, (DIKU report 94/19) (1994).

[17] N. Heintze, 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 (PLDI'2001), ACM Press, 2001, pp. 254–263.

[18] 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, ACM Press, 2001, pp. 54–61.

[19] A. van de Ven, New security enhancements in Red Hat Enterprise Linux v.3, update 3, `http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf`, white paper, Red Hat, Inc. (Aug. 2004).

[20] C. B. Jones, Tentative steps toward a development method for interfering programs, ACM Transactions on Programming Languages and Systems 5 (4) (1983) 596–619.

[21] F. Nielson, H. R. Nielson, H. Seidl, Normalizable Horn clauses, strongly recognizable relations, and Spi, in: Proceedings of the 9th International Symposium on Static Analysis (SAS'2002), Springer-Verlag LNCS 2477, 2002, pp. 20–35.

[22] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, 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, I. H. Sudborough (Eds.), The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, Springer-Verlag LNCS 2566, 2002, pp. 85–108.

[23] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, A static analyzer for large safety-critical software, in: ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), ACM Press, San Diego, CA, 2003, pp. 196–207.

[24] P. Selinger, Models for an adversary-centric protocol logic, Electronic Notes in Theoretical Computer Science 55 (1) (2001) 73–87, proceedings of the 1st Workshop on Logical Aspects of Cryptographic Protocol Verification (LACPV'01), J. Goubault-Larrecq, ed.

[25] J. Goubault-Larrecq, Deciding $\mathcal{H}_1$ by resolution, Information Processing Letters. 95 (3) (2005) 401–408.

[26] T. Frühwirth, E. Shapiro, M. Y. Vardi, E. Yardeni, Logic programs as types for logic programs, in: Proceedings of the 6th Symposium Logic in Computer Science (LICS'91), IEEE Computer Society Press, 1991, pp. 300–309.

[27] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, D. Topic, SPASS version 2.0, in: Proc. 18th Conference on Automated Deduction (CADE-18), Springer-Verlag LNCS 2392, 2002, pp. 275–279.

[28] 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), INRIA, Sainte-Marie-de-Ré France, 2004, pp. 1–40.

[29] F. Parrennes, The CSur project, `http://www.lsv.ens-cachan.fr/csur/` (2004).

[30] C. Weidenbach, Towards an automatic analysis of security protocols, in: H. Ganzinger (Ed.), 16th International Conference on Automated Deduction (CADE-16), Springer-Verlag LNCS 1632, 1999, pp. 378–382.

[31] J. Goubault-Larrecq, The `h1` Tool Suite, LSV/UMR CNRS & ENS Cachan, INRIA Futurs projet SECSI (Jan. 2005).
URL
`http://www.lsv.ens-cachan.fr/~goubault/H1/h1index.html`