Cryptographic Protocol Analysis on Real C Code

Jean Goubault-Larrecq (LSV)
Fabrice Parrennes (LSV, RATP)

VMCAI — January 19, 2005

Outline

- Verifying cryptographic protocols through logic. . .

Outline

- Verifying cryptographic protocols through logic
- Or rather C code implementing crypto protocols…

Outline

- Verifying cryptographic protocols through logic
- Or rather C code implementing crypto protocols
- Unifying (simple) shape analysis with security analysis through logic.

Outline

- Verifying cryptographic protocols through logic
- Or rather C code implementing crypto protocols
- Unifying (simple) shape analysis with security analysis through logic.
- Demo, conclusion.

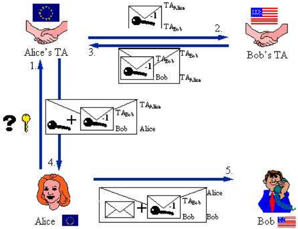
Cryptographic Protocols

Cryptography:

Cryptographic Protocols

Cryptography:

Protocols:



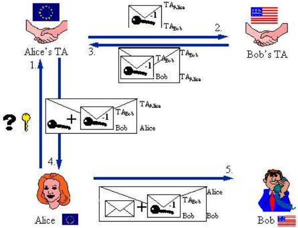
Used to ensure:

[sample]

Cryptographic Protocols

Cryptography:

Protocols:

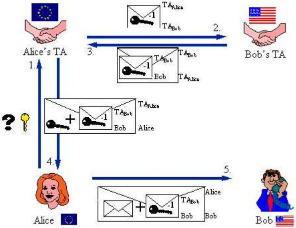


[sample]

Used to ensure:
secrecy: *M* is secret if no intruder can emit *M*;
authenticity: the only process that can build *M* is *A*;
etc.

Cryptographic Protocols

Cryptography: 

Protocols: 

Used to ensure: [sample]

secrecy: *M* is secret if no intruder can emit *M*;

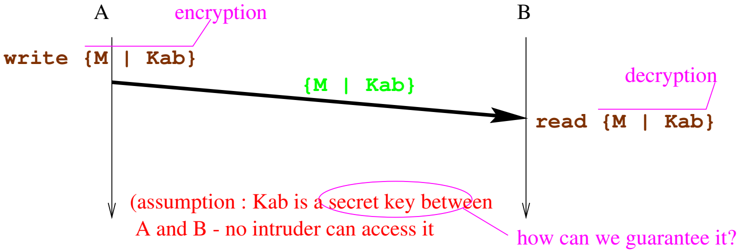
authenticity: the only process that can build *M* is *A*;

etc.

We shall concentrate on basic, **unreachability** properties, e.g., secrecy.

Cryptography Is Not Enough.

Even if you use perfect encryption algorithms (unbreakable), it is not easy to preserve secrecy or authenticity:

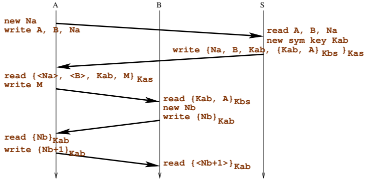


Cryptography Is Not Enough.

Even if you use perfect encryption algorithms (unbreakable), it is not easy to preserve secrecy or authenticity.

Needham-Schroeder's symmetric key protocol:

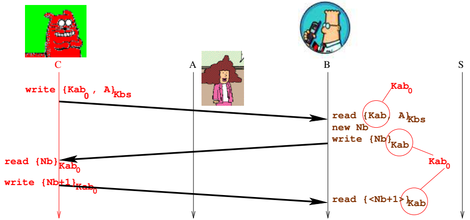
$$\begin{array}{ll}
1. & A \longrightarrow S : A, B, N_a \\
2. & S \longrightarrow A : \{N_a, B, K_{ab}, \{K_{ab}, A\}_{K_{bs}}\}_{K_{as}} \\
3. & A \longrightarrow B : \{K_{ab}, A\}_{K_{bs}} \\
4. & B \longrightarrow A : \{N_b\}_{K_{ab}} \\
5. & A \longrightarrow B : \{N_b + 1\}_{K_{ab}}
\end{array}$$



Cryptography Is Not Enough.

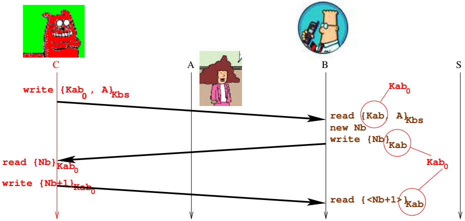
Even if you use perfect encryption algorithms (unbreakable), it is not easy to preserve secrecy or authenticity.

Needham-Schroeder's symmetric key protocol… and its attack:



Cryptography Is Not Enough.

Even if you use perfect encryption algorithms (unbreakable), it is not easy to preserve secrecy or authenticity



Purely logical attack!

Related Work

A fashionable domain. Many papers: formal methods, process calculi, strand spaces, abstract interpretation, tree automata, Horn clauses, etc.

Related Work

A fashionable domain.
Particularly relevant, but not exhaustive:

Related Work

A fashionable domain.
Particularly relevant, but not exhaustive:

- B. Blanchet 2001–2004 (sometimes with coauthors: M. Abadi, A. Podelski): encode (slightly idealized) reachability in protocols as sets of Horn clauses.
  Security = unreachability = satisfiability

Related Work

A fashionable domain.
Particularly relevant, but not exhaustive:

- B. Blanchet 2001–2004 (sometimes with coauthors: M. Abadi, A. Podelski): encode (slightly idealized) reachability in protocols as sets of Horn clauses.
  Security = unreachability = satisfiability
- F. Nielson, H.R. Nielson, H. Seidl 2002: encode (slightly idealized) reachability semantics of spi-calculus as decidable class $\mathcal{H}_1$ of Horn clauses.

Related Work

A fashionable domain.
Particularly relevant, but not exhaustive:

- B. Blanchet 2001–2004 (sometimes with coauthors: M. Abadi, A. Podelski): encode (slightly idealized) reachability in protocols as sets of Horn clauses.
  Security = unreachability = satisfiability
- F. Nielson, H.R. Nielson, H. Seidl 2002: encode (slightly idealized) reachability semantics of spi-calculus as decidable class $\mathcal{H}_1$ of Horn clauses.
- Related to D. Monniaux 1999, Goubault-Larrecq 2000, based on finite tree automata: one may see $\mathcal{H}_1$ as an elegant way of describing tree regular languages.

A Horn Clause Model 1. Intruder Abilities.

$$\begin{array}{rcll}
\texttt{knows}(\{M\}_K) & \Leftarrow & \texttt{knows}(M), \texttt{knows}(K) & (C \text{ can encrypt}) \\
\texttt{knows}(M) & \Leftarrow & \texttt{knows}(\{M\}_{\texttt{k}(\texttt{sym},X)}), & \\
 & & \texttt{knows}(\texttt{k}(\texttt{sym}, X)) & \ldots \text{and decrypt [symmetric keys])} \\
\texttt{knows}([]) & & & (C \text{ can build} \\
\texttt{knows}(M_1 :: M_2) & \Leftarrow & \texttt{knows}(M_1), \texttt{knows}(M_2) & \text{any list of known messages)} \\
\texttt{knows}(M_1) & \Leftarrow & \texttt{knows}(M_1 :: M_2) & (C \text{ can read heads}) \\
\texttt{knows}(M_2) & \Leftarrow & \texttt{knows}(M_1 :: M_2) & (C \text{ can read tails}) \\
\texttt{knows}(\texttt{suc}(M)) & \Leftarrow & \texttt{knows}(M) & (C \text{ can add} \\
\texttt{knows}(M) & \Leftarrow & \texttt{knows}(\texttt{suc}(M)) & \text{and subtract one)}
\end{array}$$

2. Protocol clauses—current sessions (à la Blanchet)

$$1.\ A \longrightarrow S : A, B, N_a \quad \mathtt{knows}([\mathtt{a}, \mathtt{b}, \mathtt{na}([\mathtt{a}, \mathtt{b}])])$$

$$\begin{array}{l} 1.\ A \rightarrow S :\ A, B, N_a \\ 2.\ S \longrightarrow A : \{N_a, B, K_{ab}, \\ \qquad \{K_{ab}, A\}_{K_{bs}} \\ \qquad \}_{K_{as}} \end{array} \quad \mathtt{knows}\left(\begin{array}{l} \{[N_a, B, k_{ab}, \\ \quad \{[k_{ab}, A]\}_{\mathtt{k}(\mathtt{sym},[B,\mathtt{s}])} \\ \quad]\}_{\mathtt{k}(\mathtt{sym},[A,\mathtt{s}])}) \end{array} \right) \Leftarrow \mathtt{knows}([A, B, N_a])$$

$$(k_{ab} \equiv \mathtt{k}(\mathtt{sym}, \mathtt{cur}(A, B, N_a)))$$

$$\begin{array}{l} 2.\ S \longrightarrow A : \{N_a, B, K_{ab}, \\ \qquad \{K_{ab}, A\}_{K_{bs}} \\ \qquad \}_{K_{as}} \\ 3.\ A \longrightarrow B : \{K_{ab}, A\}_{K_{bs}} \end{array} \quad \begin{array}{rcl} \mathtt{knows}(M) & \Leftarrow & \mathtt{knows}(\{[\mathtt{na}([\mathtt{a}, \mathtt{b}]), \mathtt{b}, K_{ab}, M]\}_{\mathtt{k}(\mathtt{sym},[\mathtt{a},\mathtt{s}])}) \\ \mathtt{a\_key}(K_{ab}) & \Leftarrow & \mathtt{knows}(\{[\mathtt{na}([\mathtt{a}, \mathtt{b}]), \mathtt{b}, K_{ab}, M]\}_{\mathtt{k}(\mathtt{sym},[\mathtt{a},\mathtt{s}])}) \end{array}$$

$$\begin{array}{l} 3.\ A \longrightarrow B : \{K_{ab}, A\}_{K_{bs}} \\ 4.\ B \longrightarrow A : \{N_b\}_{K_{ab}} \end{array} \quad \mathtt{knows}(\{\mathtt{nb}(K_{ab}, A, B)\}_{K_{ab}}) \Leftarrow \mathtt{knows}(\{[K_{ab}, A]\}_{\mathtt{k}(\mathtt{sym},[B,\mathtt{s}])})$$

---

4. $B \longrightarrow A : \{N_b\}_{K_{ab}}$
5. $A \longrightarrow B : \{N_b + 1\}_{K_{ab}}$

$$\mathtt{knows}(\{\mathtt{suc}(N_b)\}_{K_{ab}}) \Leftarrow \mathtt{knows}(\{N_b\}_{K_{ab}})$$

3. Protocol clauses—old sessions

$$\begin{array}{l} 1.\ A \rightarrow S :\ A, B, N_a \\ 2.\ S \longrightarrow A : \{N_a, B, K_{ab}, \\ \qquad\qquad \{K_{ab}, A\}_{K_{bs}} \\ \qquad\quad \}_{K_{as}} \end{array} \quad \mathtt{knows}\left(\begin{array}{l}\{[N_a, B, k_{ab}, \\ \quad \{[k_{ab}, A]\}_{\mathtt{k}(\mathtt{sym},[B,\mathtt{s}])} \\ \quad]\}_{\mathtt{k}(\mathtt{sym},[A,\mathtt{s}])}\end{array}\right) \Leftarrow \mathtt{knows}([A, B, N_a])$$

$$(k_{ab} \equiv \mathtt{k}(\mathtt{sym}, \mathtt{prev}(A, B, N_a)))$$

4. Initial intruder knowledge

$$\begin{array}{rcl}
\texttt{agent}(\texttt{a}) & & \texttt{agent}(\texttt{b}) \\
\texttt{agent}(\texttt{s}) & & \texttt{agent}(\texttt{i}) \\
\texttt{knows}(X) & \Leftarrow & \texttt{agent}(X) \\
\texttt{knows}(\texttt{k}(\texttt{pub}, X)) & & \\
\texttt{knows}(\texttt{k}(\texttt{prv}, \texttt{i})) & & \\
\texttt{knows}(\texttt{k}(\texttt{sym}, \texttt{prev}(A, B, N_a))) & & \text{(old session keys} \\
 & & \text{are compromised)}
\end{array}$$

5. Security queries

$$\perp \quad \Leftarrow \quad \mathtt{knows}(\mathtt{k}(\mathtt{sym}, \mathtt{cur}(\mathtt{a}, \mathtt{b}, N_a)))$$

can C build K_{ab}

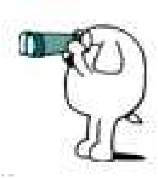
as created by S?

$$\perp \quad \Leftarrow \quad \mathtt{knows}(K_{ab}), \mathtt{a\_key}(K_{ab})$$

... as received by A?

$$\perp \quad \Leftarrow \quad \mathtt{knows}(\{\mathtt{suc}(\mathtt{nb}(K_{ab}, A, B))\}_{K_{ab}}), \mathtt{knows}(K_{ab})$$

... as received by B?



If you see this slide,
ask me to run `h1`
to find attacks
and security guarantees on
the Needham-Schroeder symmetric
key protocol!

In case I forget:

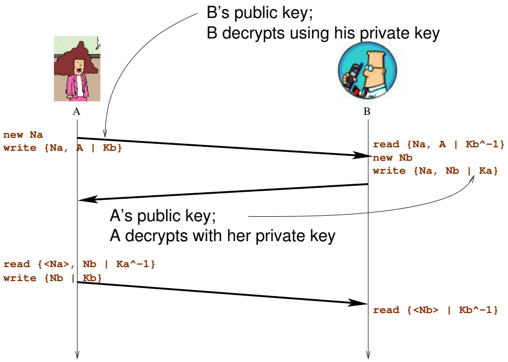
```
cd ˜H1.1/; h1 -all nspriv.p
```

Finds attack on Bob, but, less trivially: no attack on Alice or server.

Actual Code vs. Cryptographic Protocols

The Needham-Schroeder public key protocol.

. . . the cream pie of cryptographic slapstick!



B's public key;
B decrypts using his private key

A

B

`new Na`
`write {Na, A | Kb}`

`read {Na, A | Kb^-1}`
`new Nb`
`write {Na, Nb | Ka}`

A's public key;
A decrypts with her private key

`read {<Na>, Nb | Ka^-1}`
`write {Nb | Kb}`

`read {<Nb> | Kb^-1}`

1. $\mathtt{a} \rightarrow \mathtt{b}\colon \{N_{\mathtt{a}}, \mathtt{a}\}_{\mathtt{pub}(\mathtt{b})}$
2. $\mathtt{b} \rightarrow \mathtt{a}\colon \{N_{\mathtt{a}}, N_{\mathtt{b}}\}_{\mathtt{pub}(\mathtt{a})}$
3. $\mathtt{a} \rightarrow \mathtt{b}\colon \{N_{\mathtt{b}}\}_{\mathtt{pub}(\mathtt{b})}$

Actual Code vs. Cryptographic Protocols

The Needham-Schroeder public key protocol. In C.

```c
int create_nonce (nonce_t *nce)
{
  RAND_bytes(nce->nonce,SIZENONCE);
  return(0);
}

int encrypt_mesg(msg1_t *msg, BIGNUM *key_pub,
                 BIGNUM *key_mod, BIGNUM *cipher)
{
  BIGNUM *plain;
  int msg_len;
  BN_CTX *ctx;
  ctx = BN_CTX_new();
  msg_len = sizeof (msg1_t);
  plain = BN_bin2bn((const unsigned char *)msg, msg_len, NULL);
  BN_CTX_init(ctx);
  BN_mod_exp(cipher, plain, key_pub, key_mod, ctx);
  return (0);
}

int write(int fd, const void *buf, int count)
{
  write (fd, buf, count);
  return(0);
}

int create_mesg1(msg1_t *mesg, nonce_t *n1, int *id, int *dest)
{
  /* First copy nonce. */
  memcpy (&msg->nonce_msg1, n1, sizeof(nonce_t));

  /* copy id... */
  msg->id_1[0] = id[0]; msg->id_1[1] = id[1];
  msg->id_1[2] = id[2]; msg->id_1[3] = id[3];
  /* ... and dest. */
  msg->dest_1[0] = dest[0]; msg->dest_1[1] = dest[1];
```

```c
int main(int argc, char *argv[])
{
  int conn_fd;  // The communication socket.
  msg1_t mesg1; // Message
  nonce_t nonce;
  BIGNUM * cipher1; // Cipher Message
  BIGNUM * pubkey;  // Keys
  BIGNUM * prvkey;  // Keys
  BIGNUM * modkey;  // Keys
  unsigned int ip_id[4];   // A's name
  unsigned int ip_dest[4]; // B's name as seen from A.

  /* Init ip_id and ip_dest. */
  ip_id[0]   = 192; ip_id[1]   = 100;
  ip_id[2]   = 200; ip_id[3]   = 100;
  ip_dest[0] = 192; ip_dest[1] = 100;
  ip_dest[2] = 200; ip_dest[3] = 101;
  // Open connection to B
  conn_fd = connect_socket(ip_dest, 522);

  init_keys(&pubkey, &prvkey, &modkey, PUBALICESERV,
            MODALICESERV, PRIVALICESERV);

  /*** 1. A -> B : {Na, A}_pub(B) ***/
  create_nonce (&nonce);
  create_mesg1(&mesg1, &nonce, ip_id, ip_dest);
  cipher1 = BN_new();
  encrypt_mesg(&mesg1, pubkey, modkey, cipher1);
  write(conn_fd, cipher1, 128);

  /** ...Remaining code omitted... **/
}
```

Actual Code vs. Cryptographic Protocols

The Needham-Schroeder public key protocol. In C.

Unanalyzable functions

```c
int create_nonce (nonce_t *nce)
{
  RAND_bytes(nce->nonce,SIZENONCE);
  return(0);
}
```

On the crypto level,
this is nonce creation.

```c
int encrypt_mesg(msg1_t *msg, BIGNUM *key_pub,
                 BIGNUM *key_mod, BIGNUM *cipher)
{
  BIGNUM *plain;
  int msg_len;
  BN_CTX *ctx;
  ctx = BN_CTX_new();
  msg_len = sizeof (msg1_t);
  plain = BN_bin2bn((const unsigned char *)msg, msg_len, NULL);
  BN_CTX_init(ctx);
  BN_mod_exp(cipher, plain, key_pub, key_mod, ctx);
  return (0);
}
```

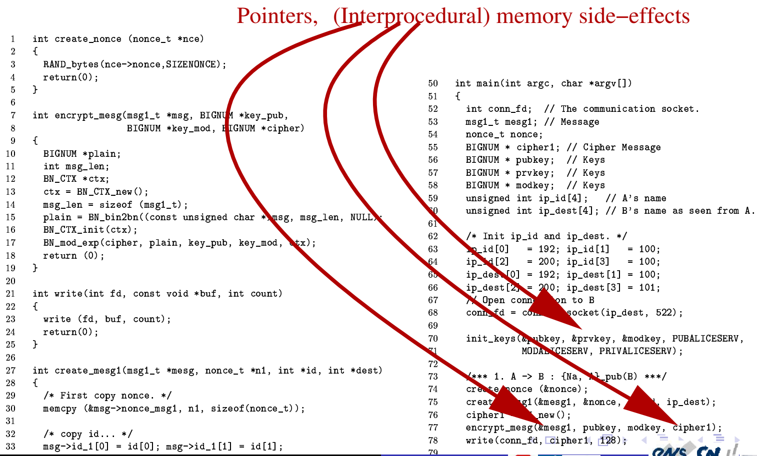
On the crypto level,
this is just encryption.

```c
int write(int fd, const void *buf, int count)
{
  write (fd, buf, count);
  return(0);
}

int create_mesg1(msg1_t *mesg, nonce_t *n1, int *id, int *dest)
{
  /* First copy nonce. */
  memcpy (&msg->nonce_msg1, n1, sizeof(nonce_t));

  /* copy id... */
```

```c
int main(int argc, char *argv[])
{
  int conn_fd;  // The communication socket.
  msg1_t mesg1; // Message
  nonce_t nonce;
  BIGNUM * cipher1; // Cipher Message
  BIGNUM * pubkey;  // Keys
  BIGNUM * prvkey;  // Keys
  BIGNUM * modkey;  // Keys
  unsigned int ip_id[4];   // A's name
  unsigned int ip_dest[4]; // B's name as seen from A.

  /* Init ip_id and ip_dest. */
  ip_id[0]   = 192; ip_id[1]   = 100;
  ip_id[2]   = 200; ip_id[3]   = 100;
  ip_dest[0] = 192; ip_dest[1] = 100;
  ip_dest[2] = 200; ip_dest[3] = 101;
  // Open connection to B
  conn_fd = connect_socket(ip_dest, 522);

  init_keys(&pubkey, &prvkey, &modkey, PUBALICESERV,
            MODALICESERV, PRIVALICESERV);

  /*** 1. A -> B : {Na, A}_pub(B) ***/
  create_nonce (&nonce);
  create_mesg1(&mesg1, &nonce, ip_id, ip_dest);
  cipher1 = BN_new();
  encrypt_mesg(&mesg1, pubkey, modkey, cipher1);
  write(conn_fd, cipher1, 128)
```

Actual Code vs. Cryptographic Protocols

The Needham-Schroeder public key protocol. In C.



Pointers, (Interprocedural) memory side–effects

```
int create_nonce (nonce_t *nce)
{
  RAND_bytes(nce->nonce,SIZENONCE);
  return(0);
}

int encrypt_mesg(msg1_t *msg, BIGNUM *key_pub,
                 BIGNUM *key_mod, BIGNUM *cipher)
{
  BIGNUM *plain;
  int msg_len;
  BN_CTX *ctx;
  ctx = BN_CTX_new();
  msg_len = sizeof (msg1_t);
  plain = BN_bin2bn((const unsigned char *)msg, msg_len, NULL);
  BN_CTX_init(ctx);
  BN_mod_exp(cipher, plain, key_pub, key_mod, ctx);
  return (0);
}

int write(int fd, const void *buf, int count)
{
  write (fd, buf, count);
  return(0);
}

int create_mesg1(msg1_t *mesg, nonce_t *n1, int *id, int *dest)
{
  /* First copy nonce. */
  memcpy (&msg->nonce_msg1, n1, sizeof(nonce_t));

  /* copy id... */
  msg->id_1[0] = id[0]; msg->id_1[1] = id[1];
```

```
int main(int argc, char *argv[])
{
  int conn_fd;  // The communication socket.
  msg1_t mesg1; // Message
  nonce_t nonce;
  BIGNUM * cipher1; // Cipher Message
  BIGNUM * pubkey;  // Keys
  BIGNUM * prvkey;  // Keys
  BIGNUM * modkey;  // Keys
  unsigned int ip_id[4];   // A's name
  unsigned int ip_dest[4]; // B's name as seen from A.

  /* Init ip_id and ip_dest. */
  ip_id[0]   = 192; ip_id[1]   = 100;
  ip_id[2]   = 200; ip_id[3]   = 100;
  ip_dest[0] = 192; ip_dest[1] = 100;
  ip_dest[2] = 200; ip_dest[3] = 101;
  // Open connection to B
  conn_fd = con[illegible]socket(ip_dest, 522);

  init_keys(&pubkey, &prvkey, &modkey, PUBALICESERV,
            MODALICESERV, PRIVALICESERV);

  /*** 1. A -> B : {Na, A}_pub(B) ***/
  create_nonce (&nonce);
  creat[illegible]g1(&mesg1, &nonce, [illegible]d, ip_dest);
  cipher1 [illegible]_new();
  encrypt_mesg(&mesg1, pubkey, modkey, cipher1);
  write(conn_fd, cipher1, 128);
```

Actual Code vs. Cryptographic Protocols

The Needham-Schroeder public key protocol. In C.

Interfacing C (pointer) semantics
with crypto semantics

```
int create_nonce (nonce_t *nce)
{
  RAND_bytes(nce->nonce,SIZENONCE);
  return(0);
}

int encrypt_mesg(msg1_t *msg, BIGNUM *key_pub,
```

Writing on a socket is:
– doing some side–effect on some bit sequence cipher1, viewed from C;
– sending a properly formed message $\{Na, A\}_{pub(B)}$ in the crypto world.

```
                                                 len, NULL);
}

int create_mesg1(msg1_t *mesg, nonce_t *n1, int *id, int *dest)
{
  /* First copy nonce. */
  memcpy (&msg->nonce_msg1, n1, sizeof(nonce_t));

  /* copy id... */
  msg->id_1[0] = id[0]; msg->id_1[1] = id[1];
```

```
int main(int argc, char *argv[])
{
  int conn_fd;  // The communication socket.
  msg1_t mesg1; // Message
  nonce_t nonce;
  BIGNUM * cipher1; // Cipher Message
  BIGNUM * pubkey;  // Keys
  BIGNUM * prvkey;  // Keys
  BIGNUM * modkey;  // Keys
  unsigned int ip_id[4];   // A's name
  unsigned int ip_dest[4]; // B's name as seen from A.

  /* Init ip_id and ip_dest. */
  ip_id[0]   = 192; ip_id[1]   = 100;
  ip_id[2]   = 200; ip_id[3]   = 100;
  ip_dest[0] = 192; ip_dest[1] = 100;
  ip_dest[2] = 200; ip_dest[3] = 101;
  // Open connection to B
  conn_fd = connect_socket(ip_dest, 522);

  init_keys(&pubkey, &prvkey, &modkey, PUBALICESERV,
            MODALICESERV, PRIVALICESERV);

  /*** 1. A -> B : {Na, A}_pub(B) ***/
  create_nonce (&nonce);
  create_mesg1(&mesg1, &nonce, ip_id, ip_dest);
  cipher1 = BN_new();
  encrypt_mesg(&mesg1, pubkey, modkey, cipher1);
  write(conn_fd, cipher1, 128);
```

If you see this slide,
ask me to run this example!

In case I forget:

```
cd ˜csur/examples/Protocols/Needham_Schroeder_public_keys
```

Same thing in second window.

`make clean; make; ./bob.exe` in one window,

`./alice.exe` in the other.

What Can We Do?

- Can we rebuild cryptographic protocol from the code?
  - Usual protocol description languages not expressive enough
    (except e.g., spi-calculus... or Horn clause sets);
  - Missing roles (e.g., the example was just Alice's role);
  - Seems as difficult as analyzing code directly anyway.

Trust Assertions

. . . Or: relating C semantics with crypto semantics.

- We trust, e.g., `encrypt_msg` to encrypt;

  at the crypto level; we still analyze it for side-effects!

Trust Assertions

. . . Or: relating C semantics with crypto semantics.

- We trust, e.g., `encrypt_msg` to encrypt;

  at the crypto level; we still analyze it for side-effects!

- We trust the environment (intruder mainly) to obey certain rules, expressed as Horn clauses;

  e.g., rules on `knows`: 1. Intruder Abilities + possibly others.

Trust Assertions

… Or: relating C semantics with crypto semantics.

- We trust, e.g., `encrypt_msg` to encrypt;

  at the crypto level; we still analyze it for side-effects!

- We trust the environment (intruder mainly) to obey certain rules, expressed as Horn clauses;

  e.g., rules on `knows`: 1. Intruder Abilities + possibly others.

- We use a special binary predicate `rec`: *e* `rec` *M* means we trust the C expression *e* to denote the crypto message *M*.

  requires annotations of library functions (or stubs).

  Avoid annotating user code as much as we can!

Trust Assertions on an Example

/* trust rec(*nce, nonce(CTX)) */

```
int create_nonce (nonce_t *nce)
{
  RAND_bytes(nce->nonce,SIZENONCE);
  return(0);
}

int encrypt_mesg(msg1_t *msg, BIGNUM *key_pub,
                 BIGNUM *key_mod, BIGNUM *cipher)
{
  BIGNUM *plain;
  int msg_len;
  BN_CTX *ctx;
  ctx = BN_CTX_new();
  msg_len = sizeof (msg1_t);
  plain = BN_bin2bn((const unsigned char *)msg, msg_len, NULL);
  BN_CTX_init(ctx);
  BN_mod_exp(cipher, plain, key_pub, key_mod, ctx);
  return (0);
}

int write(int fd, const void *buf, int count)
{
  write (fd, buf, count);
  return(0);
}

int create_mesg1(msg1_t *mesg, nonce_t *n1, int *id, int *dest)
{
  /* First copy nonce. */
  memcpy (&msg->nonce_msg1, n1, sizeof(nonce_t));

```

```
int main(int argc, char *argv[])
{
  int conn_fd;  // The communication socket.
  msg1_t mesg1; // Message
  nonce_t nonce;
  BIGNUM * cipher1; // Cipher Message
  BIGNUM * pubkey;  // Keys
  BIGNUM * prvkey;  // Keys
  BIGNUM * modkey;  // Keys
  unsigned int ip_id[4];   // A's name
  unsigned int ip_dest[4]; // B's name as seen from A.

  /* Init ip_id and ip_dest. */
  ip_id[0]   = 192; ip_id[1]   = 100;
  ip_id[2]   = 200; ip_id[3]   = 100;
  ip_dest[0] = 192; ip_dest[1] = 100;
  ip_dest[2] = 200; ip_dest[3] = 101;
  // Open connection to B
  conn_fd = connect_socket(ip_dest, 522);

  init_keys(&pubkey, &prvkey, &modkey, PUBALICESERV,
            MODALICESERV, PRIVALICESERV);

  /*** 1. A -> B : {Na, A}_pub(B) ***/
  create_nonce (&nonce);
  create_mesg1(&mesg1, &nonce, ip_id, ip_dest);
  cipher1 = BN_new();
```

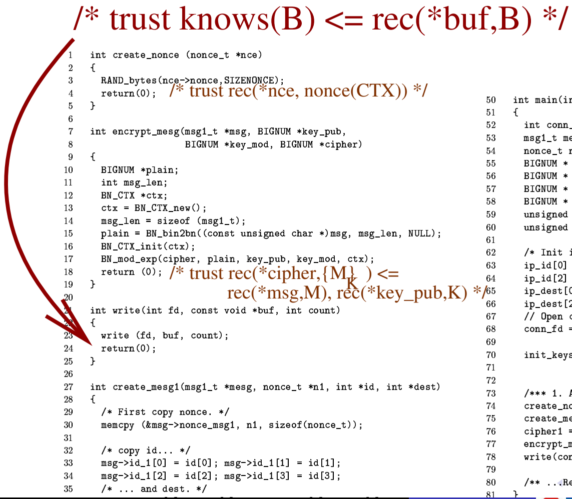
Trust Assertions on an Example

/* trust rec(*cipher,$\{M\}_K$) <=
rec(*msg,M), rec(*key_pub,K) */

```
int create_nonce (nonce_t *nce)
{
  RAND_bytes(nce->nonce,SIZENONCE);
  return(0);  /* trust rec(*nce, nonce(CTX)) */
}

int encrypt_mesg(msg1_t *msg, BIGNUM *key_pub,
                 BIGNUM *key_mod, BIGNUM *cipher)
{
  BIGNUM *plain;
  int msg_len;
  BN_CTX *ctx;
  ctx = BN_CTX_new();
  msg_len = sizeof (msg1_t);
  plain = BN_bin2bn((const unsigned char *)msg, msg_len, NULL);
  BN_CTX_init(ctx);
  BN_mod_exp(cipher, plain, key_pub, key_mod, ctx);
  return (0);
}

int write(int fd, const void *buf, int count)
{
  write (fd, buf, count);
  return(0);
}

int create_mesg1(msg1_t *mesg, nonce_t *n1, int *id, int *dest)
{
  /* First copy nonce. */
  memcpy (&msg->nonce_msg1, n1, sizeof(nonce_t));

  /* copy id... */
  msg->id_1[0] = id[0]; msg->id_1[1] = id[1];
  msg->id_1[2] = id[2]; msg->id_1[3] = id[3];
```

```
int main(int argc, char *argv[])
{
  int conn_fd;  // The communication socket.
  msg1_t mesg1; // Message
  nonce_t nonce;
  BIGNUM * cipher1; // Cipher Message
  BIGNUM * pubkey;  // Keys
  BIGNUM * prvkey;  // Keys
  BIGNUM * modkey;  // Keys
  unsigned int ip_id[4];   // A's name
  unsigned int ip_dest[4]; // B's name as seen from A.

  /* Init ip_id and ip_dest. */
  ip_id[0]   = 192; ip_id[1]   = 100;
  ip_id[2]   = 200; ip_id[3]   = 100;
  ip_dest[0] = 192; ip_dest[1] = 100;
  ip_dest[2] = 200; ip_dest[3] = 101;
  // Open connection to B
  conn_fd = connect_socket(ip_dest, 522);

  init_keys(&pubkey, &prvkey, &modkey, PUBALICESERV,
            MODALICESERV, PRIVALICESERV);

  /*** 1. A -> B : {Na, A}_pub(B) ***/
  create_nonce (&nonce);
  create_mesg1(&mesg1, &nonce, ip_id, ip_dest);
  cipher1 = BN_new();
  encrypt_mesg(&mesg1, pubkey, modkey, cipher1);
  write(conn_fd, cipher1, 128);
```

Trust Assertions on an Example



/* trust knows(B) <= rec(*buf,B) */

```
int create_nonce (nonce_t *nce)
{
  RAND_bytes(nce->nonce,SIZENONCE);
  return(0);
}

int encrypt_mesg(msg1_t *msg, BIGNUM *key_pub,
                 BIGNUM *key_mod, BIGNUM *cipher)
{
  BIGNUM *plain;
  int msg_len;
  BN_CTX *ctx;
  ctx = BN_CTX_new();
  msg_len = sizeof (msg1_t);
  plain = BN_bin2bn((const unsigned char *)msg, msg_len, NULL);
  BN_CTX_init(ctx);
  BN_mod_exp(cipher, plain, key_pub, key_mod, ctx);
  return (0);
}

int write(int fd, const void *buf, int count)
{
  write (fd, buf, count);
  return(0);
}

int create_mesg1(msg1_t *mesg, nonce_t *n1, int *id, int *dest)
{
  /* First copy nonce. */
  memcpy (&msg->nonce_msg1, n1, sizeof(nonce_t));

  /* copy id... */
  msg->id_1[0] = id[0]; msg->id_1[1] = id[1];
  msg->id_1[2] = id[2]; msg->id_1[3] = id[3];
  /* ... and dest. */
```

/* trust rec(*nce, nonce(CTX)) */

/* trust rec(*cipher,$\{M\}_K$) <=
rec(*msg,M), rec(*key_pub,K) */

```
int main(int argc, char *argv[])
{
  int conn_fd;  // The communication socket.
  msg1_t mesg1; // Message
  nonce_t nonce;
  BIGNUM * cipher1; // Cipher Message
  BIGNUM * pubkey;  // Keys
  BIGNUM * prvkey;  // Keys
  BIGNUM * modkey;  // Keys
  unsigned int ip_id[4];   // A's name
  unsigned int ip_dest[4]; // B's name as seen from A.

  /* Init ip_id and ip_dest. */
  ip_id[0]   = 192; ip_id[1]   = 100;
  ip_id[2]   = 200; ip_id[3]   = 100;
  ip_dest[0] = 192; ip_dest[1] = 100;
  ip_dest[2] = 200; ip_dest[3] = 101;
  // Open connection to B
  conn_fd = connect_socket(ip_dest, 522);

  init_keys(&pubkey, &prvkey, &modkey, PUBALICESERV,
            MODALICESERV, PRIVALICESERV);

  /*** 1. A -> B : {Na, A}_pub(B) ***/
  create_nonce (&nonce);
  create_mesg1(&mesg1, &nonce, ip_id, ip_dest);
  cipher1 = BN_new();
  encrypt_mesg(&mesg1, pubkey, modkey, cipher1);
  write(conn_fd, cipher1, 128);

  /** ...Remaining code omitted     **/
}
```

Related Work

- N. El Kadhi 2001, P. Boury and N. El Kadhi 2001: similar problem, for JavaCard applets; some simplifying assumptions; generates constraints and uses dedicated solver (StuPa).

Related Work

- N. El Kadhi 2001, P. Boury and N. El Kadhi 2001: similar problem, for JavaCard applets; some simplifying assumptions; generates constraints and uses dedicated solver (StuPa).
- I should say the most important problem in analyzing security of code is dealing with buffer overflows.
    - See A. Simon, A. King 2002 for a nice static analysis approach to this problem.
    - Or we can use intrusion detection/prevention (run-time) approaches to deal with this in practice (reference monitors, ORCHIDS).

Related Work

- N. El Kadhi 2001, P. Boury and N. El Kadhi 2001: similar problem, for JavaCard applets; some simplifying assumptions; generates constraints and uses dedicated solver (StuPa).
- I should say the most important problem in analyzing security of code is dealing with buffer overflows.
  - See A. Simon, A. King 2002 for a nice static analysis approach to this problem.
  - Or we can use intrusion detection/prevention (run-time) approaches to deal with this in practice (reference monitors, ORCHIDS).
  - In any case, this is orthogonal to our concern: we assume no overflow in access to arrays, structs.

Concrete Semantics, Without Trust Assertions

You know the stuff:

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

$q, \rho, \mu \xrightarrow{x=c} q', \rho, \mu[\rho(x) \mapsto c]$

$q, \rho, \mu \xrightarrow{x=f} q', \rho, \mu[\rho(x) \mapsto a]$ if $\mu(a) = \mathtt{code}\ f$ for sor

$q, \rho, \mu \xrightarrow{x=\&y} q', \rho, \mu[\rho(x) \mapsto \mathtt{ptr}(\rho(y))]$

$q, \rho, \mu \xrightarrow{x=*y} q', \rho, \mu[\rho(x) \mapsto \hat{\mu}(\ell)]$ if $\mu(\rho(y)) = \mathtt{ptr}\ \ell$ fo

$q, \rho, \mu \xrightarrow{*x=y} q', \rho, \mu[\ell \mapsto \mu(\rho(y))]$ if $\mu(\rho(x)) = \mathtt{ptr}\ \ell$ fo

$q, \rho, \mu \xrightarrow{x=\&y[z]} q', \rho, \mu[\rho(x) \mapsto \mathtt{ptr}\ (\ell.(j+1))]$ if $\rho(y) =$ p
$\mu(\ell) = \mathtt{array}\ (z_1, \ldots, z_n)$, and $\mu(\rho(z)) =$ i

$q, \rho, \mu \xrightarrow{x=\&y\rightarrow a} q', \rho, \mu[\rho(x) \mapsto \mathtt{ptr}\ (\ell.a)]$ if $\rho(y) = \mathtt{ptr}\ \ell,$
and $\mu(\ell) = \mathtt{struct}\ \{$

Concrete Semantics, With Trust Assertions

New components:

- $\mathcal{R}$: binary relation between C values and crypto messages.
  Values are (possibly infinite) tree unfoldings of memory graph μ
  reachable from given address ℓ.
  … formally, pairs (μ, ℓ) up to bisimulation
- $\mathcal{B}$: set of facts (e.g., $\texttt{knows}(N_a)$).

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

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

$$q, \rho, \mu, \mathcal{R}, \mathcal{B} \xrightarrow{x=f} q', \rho, \mu[\rho(x) \mapsto a], \mathcal{R}, \mathcal{B} \quad \text{if } \mu(a) = \texttt{code}\ f \text{ for so}$$

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

$$\vdots$$

Concrete Semantics, With Trust Assertions

New components:

- $\mathcal{R}$: binary relation between C values and crypto messages.
- $\mathcal{B}$: set of facts.

$$q, \rho, \mu, \mathcal{R}, \mathcal{B} \xrightarrow{\texttt{trust } A \Leftarrow A_1, \ldots, A_n} q', \rho, \mu, \mathcal{R}', \mathcal{B}'$$

where, informally, $\mathcal{R}'$, $\mathcal{B}'$ are obtained by:

- laying out the contents of $\mathcal{R}$, $\mathcal{B}$ as (infinitely many) facts;
- adding the clause $A \Leftarrow A_1, \ldots, A_n$;
- deducing every fact from all this;

  i.e., computing the least Herbrand model;
- and distributing them into $\mathcal{R}'$ and $\mathcal{B}'$.

Abstract Semantics, C Semantics

Do some shape analysis.
In practice, a simple one inspired from points-to analysis (Andersen 1994, Steensgaard 1996), keeping shapes of values.

- Create constants c_ℓ for each syntactically recognizable allocation site;

  `malloc`, of course

  . . . also `&x` for each variable `x`.

- Model "points-to" relation by binary predicate p, semantics expressed through Horn clauses.

  mixes well with crypto semantics.

- Loses lots of information! (Notably order of execution.) Seems to be enough on preliminary experiments, though.

Abstract Semantics, C Semantics

Do some shape analysis.
In practice, a simple one inspired from points-to analysis (Andersen 1994, Steensgaard 1996), keeping shapes of values.

$$
\begin{array}{rcl}
\llbracket x = y \rrbracket^{\#} \rho^{\#} & = & \{\mathrm{p}(\mathrm{c}_x, X) \Leftarrow \mathrm{p}(\mathrm{c}_y, X)\} \text{ where } \mathrm{c}_x = \rho^{\#}(x), \mathrm{c}_y = \rho^{\#} \\
\llbracket x = c \rrbracket^{\#} \rho^{\#} & = & \{\mathrm{p}(\mathrm{c}_x, c)\} \\
\llbracket x = f \rrbracket^{\#} \rho^{\#} & = & \{\mathrm{p}(\mathrm{c}_x, \mathtt{code}(f))\} \\
\llbracket x = \&y \rrbracket^{\#} \rho^{\#} & = & \{\mathrm{p}(\mathrm{c}_x, \mathtt{ptr}(\mathrm{c}_y))\} \\
\llbracket x = *y \rrbracket^{\#} \rho^{\#} & = & \{\mathrm{p}(\mathrm{c}_x, X) \Leftarrow \mathrm{p}(\mathrm{c}_y, \mathtt{ptr}\ Y), \mathrm{p}(Y, X)\} \\
\llbracket *x = y \rrbracket^{\#} \rho^{\#} & = & \{\mathrm{p}(X, Y) \Leftarrow \mathrm{p}(\mathrm{c}_x, \mathtt{ptr}\ X), \mathrm{p}(\mathrm{c}_y, Y)\} \\
 & \vdots &
\end{array}
$$

Abstract Semantics, Crypto Semantics

Do some shape analysis.
In practice, a simple one inspired from points-to analysis (Andersen 1994, Steensgaard 1996), keeping shapes of values.
Nice (and easy) integration with trust assertion semantics.

$$[\![\texttt{trust } A \Leftarrow A_1, \ldots, A_n]\!]^{\#} \rho^{\#} \quad = \quad \{(A \Leftarrow A_1, \ldots, A_n)\rho^{\sharp}\}$$

Comments

- A logical view of points-to analyses (or related ones);
- Efficiency:
  - Well, Horn clause satisfiability is undecidable.

Comments

- A logical view of points-to analyses (or related ones);
- Efficiency:
  - Well, Horn clause satisfiability is undecidable.
  - Even then, current theorem provers can only handle up to a few hundred clauses at best.

Comments

- A logical view of points-to analyses (or related ones);
- Efficiency:
  - Well, Horn clause satisfiability is undecidable.
  - Even then, current theorem provers can only handle up to a few hundred clauses at best.
  - But most clauses are in Nielson, Nielson, and Seidl (2002)'s decidable class $\mathcal{H}_1$

    . . . and the remaining ones can be abstracted à la Frühwirth, Shapiro, Vardi, Yardeni (1991), without losing much important information.

    . . . keeping relationships between brothers, a strong point of $\mathcal{H}_1$.

Comments

- A logical view of points-to analyses (or related ones);
- Efficiency:
  - Well, Horn clause satisfiability is undecidable.
  - Even then, current theorem provers can only handle up to a few hundred clauses at best.
  - But most clauses are in Nielson, Nielson, and Seidl (2002)'s decidable class $\mathcal{H}_1$
    … and the remaining ones can be abstracted à la Frühwirth, Shapiro, Vardi, Yardeni (1991), without losing much important information.
    … keeping relationships between brothers, a strong point of $\mathcal{H}_1$.
  - Yes, $\mathcal{H}_1$ is DEXPTIME-complete…

Comments

- A logical view of points-to analyses (or related ones);
- Efficiency:
  - Well, Horn clause satisfiability is undecidable.
  - Even then, current theorem provers can only handle up to a few hundred clauses at best.
  - But most clauses are in Nielson, Nielson, and Seidl (2002)'s decidable class $\mathcal{H}_1$

    … and the remaining ones can be abstracted à la Frühwirth, Shapiro, Vardi, Yardeni (1991), without losing much important information.

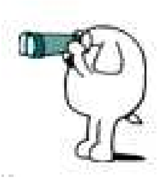
    … keeping relationships between brothers, a strong point of $\mathcal{H}_1$.
  - Yes, $\mathcal{H}_1$ is DEXPTIME-complete…
  - But most clauses are in fact in $\mathcal{H}_2$ (polynomial), or even $\mathcal{H}_3$ (cubic). (See points-to clauses.)

Comments

- A logical view of points-to analyses (or related ones);
- Efficiency:
  - Well, Horn clause satisfiability is undecidable.
  - Even then, current theorem provers can only handle up to a few hundred clauses at best.
  - But most clauses are in Nielson, Nielson, and Seidl (2002)'s decidable class $\mathcal{H}_1$

    … and the remaining ones can be abstracted à la Frühwirth, Shapiro, Vardi, Yardeni (1991), without losing much important information.

    … keeping relationships between brothers, a strong point of $\mathcal{H}_1$.
  - Yes, $\mathcal{H}_1$ is DEXPTIME-complete…
  - But most clauses are in fact in $\mathcal{H}_2$ (polynomial), or even $\mathcal{H}_3$ (cubic). (See points-to clauses.)
  - Yes, cubic is still too much in practice. We use additional ad hoc optimizations.



If you see this slide,
ask me to run the `csur` tool!

In case I forget:

```
cd ˜csur/examples/Protocols/Needham_Schroeder_public_keys
```

`make clean; make analysis`; note lots of warnings
(has to parse all of `stdlib` + some bugs remain. . .)

`h1 -all tptp_1.p`.

Note finds attack on Bob again, not on Alice.
Note here we analyze Alice as code + Bob as code
+ environment as clauses.

Conclusion

- A logical view of points-to analyses, through Horn clauses (and $\mathcal{H}_{1,2,3}$).
- Logic allows us to integrate pointer semantics with crypto semantics seamlessly.
- Working prototype: the `Csur` tool. Written by Fabrice Parrennes, atop a front-end by Jean Goubault-Larrecq. Available at `http://www.lsv.ens-cachan.fr/csur/`.