Computational Verification of C Protocol Implementations by Symbolic Execution

Mihhail Aizatulin[1] Andrew Gordon[23] Jan Jürjens[4]

[1]The Open University

[2]Microsoft Research Cambridge

[3]University of Edinburgh

[4]Dortmund University

October 2012

The Goal

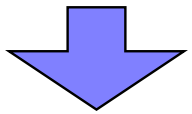
What we ultimately care about is the security of deployed systems. Therefore C programs are our objects of study.

- Things to verify: OpenSSL, IPSec, TPM, PKCS11 reference implementations, ...
- Method: extract a model using symbolic execution, then translate it to CryptoVerif.
- We check trace properties such as authentication and weak secrecy, aiming to be automated and sound.
- Assume correctness of cryptographic primitives.
- Main limitation so far: model extracted from a single program path.

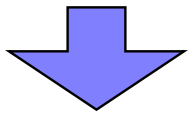
Example Flaw

In a Microsoft Research implementation of a smart metering protocol (1000 LOC):

```
unsigned char session_key[256 / 8];
...
encrypted_reading =
      ((unsigned int) *session_key) ^ *reading;
```



let msg3 = (hash2{0, 1} castTo "unsigned_int") $\oplus$ reading1 **in** ...



let msg3 = parse1(hash2) $\oplus$ reading1 **in** ...

Background

Types of properties and languages.

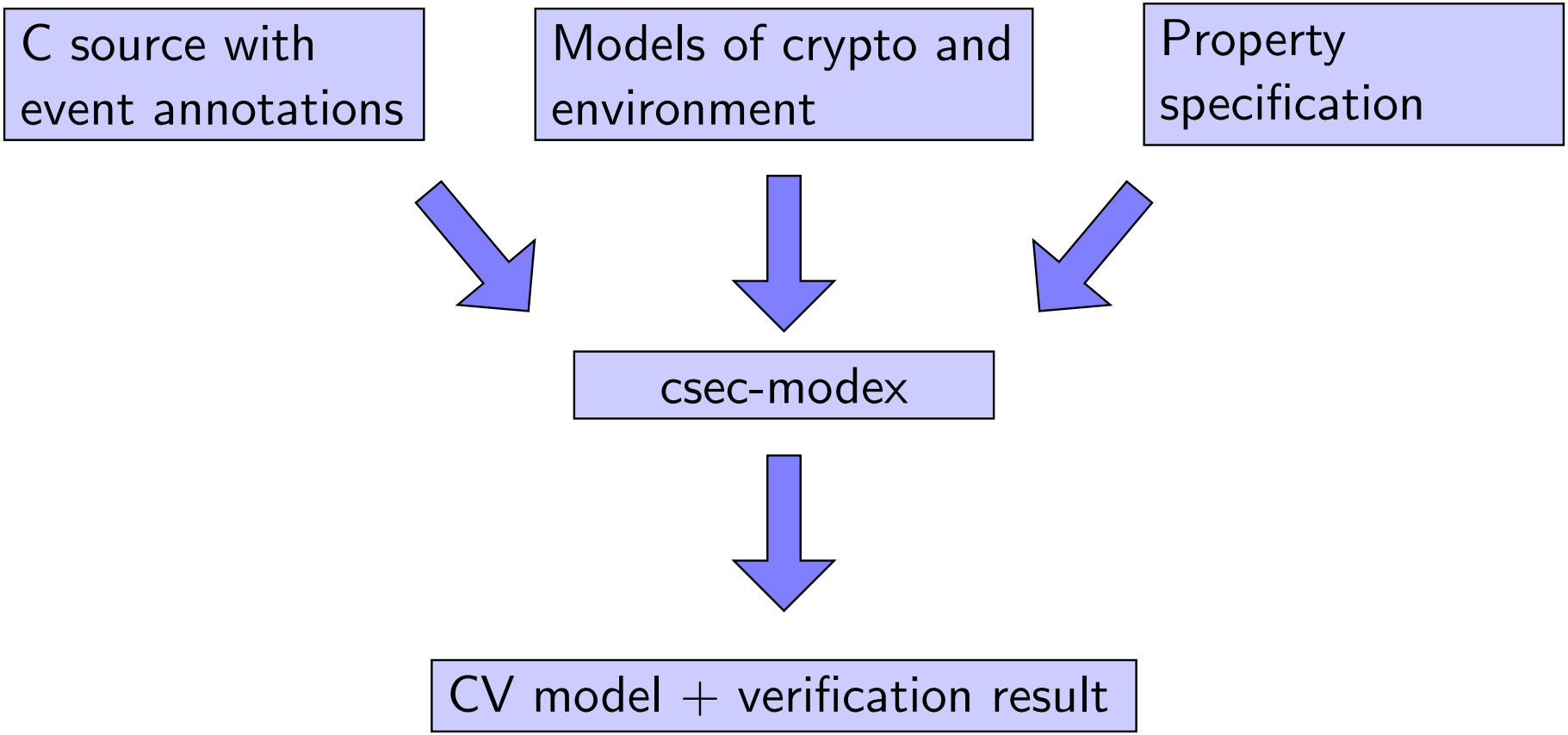
	Low-Level (C, Java)	High-Level (F#)	Formal (π, LySa)
low-level (NULL dereference, division by zero)	• VCC • Frama-C • ESC/Java • SLAM	N/A	N/A
high-level (secrecy, authentication)	• CSur • ASPIER • VCC-sec • csec-modex	• F7/F* • fs2pv/fs2cv	• ProVerif • CryptoVerif • EasyCrypt • AVISPA

Results

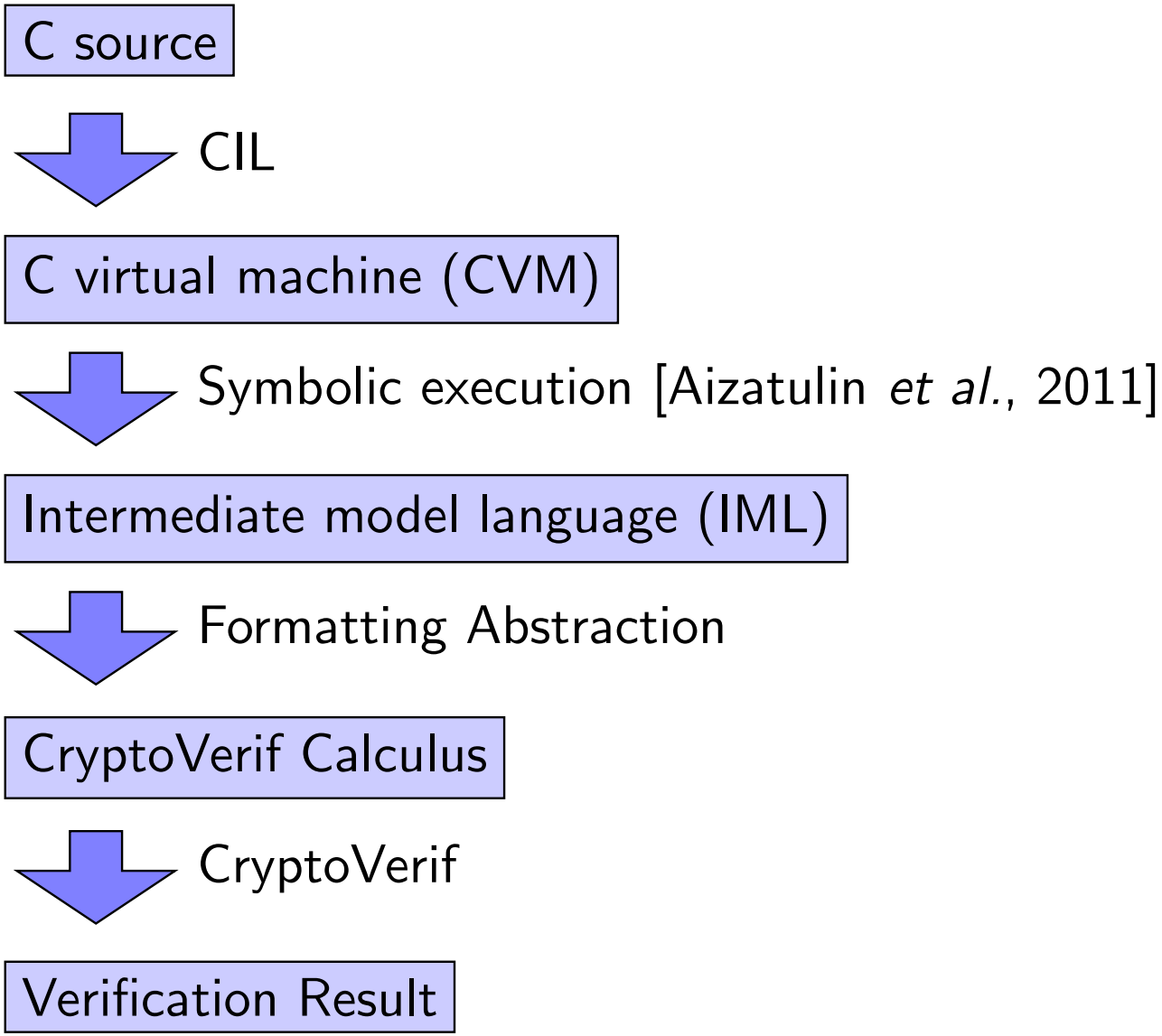
	C LOC	CV LOC	Time	Primitives
Simple MAC	~ 250	107	4s	UF-CMA MAC
Simple XOR	~ 100	90	4s	XOR
NSL	~ 450	241	26s	IND-CCA2 PKE
RPC	~ 600	126	5s	UF-CMA MAC
RPC-enc	~ 700	200	6s	IND-CPA INT-CTXT AE
Metering	~ 1000	266	21s	UF-CMA sig, CR/PRF hash

- Six protocols verified in the computational model (as opposed to just one in the ProVerif approach).
- Found flaws in an NSL implementation and the implementation of a smart metering protocol.
- NSL implementation verified before, but computational soundness places unrealistic assumptions!

Overview: What



Overview: How



One-time Pad: Symbolic Execution

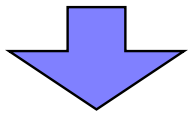
```
unsigned char * payload = malloc(PAYLOAD_LEN);
RAND_bytes(payload, PAYLOAD_LEN);

ulong msg_len = PAYLOAD_LEN + 1;
unsigned char * msg = malloc(msg_len);

* msg = 0x01;                                    // add tag
memcpy(msg + 1, payload, PAYLOAD_LEN); // add payload

unsigned char * pad = otp(msg_len);      // one-time pad
xor(msg, pad, msg_len);

send(msg, msg_len);
```



new nonce1: fixed_20; **out**(XOR(01|nonce1, pad))

One-time Pad: CV Model

```
type fixed_20 [fixed, large].
type fixed_21 [fixed, large].

expand Xor(fixed_21, XOR).

query secret nonce1.

fun conc1(fixed_20): fixed_21 [compos].

let A =
  in(c_in, ());
  new nonce1: fixed_20;
  out(c_out, XOR(conc1(nonce1), pad)); 0 .

process
! N (
  in(c_in, ());
  (* one-time pad *)
  new pad: fixed_21;
  out(c_out, ());
  A)
```

RPC-enc

$$
\begin{array}{lll}
A & : & \textbf{event}\ client\_begin(A, B, request) \\
A \rightarrow B: & & A, \{request, k_S\}_{k_{AB}} \\
B & : & \textbf{event}\ server\_reply(A, B, request, response) \\
B \rightarrow A: & & \{response\}_{k_S} \\
A & : & \textbf{event}\ client\_accept(A, B, request, response)
\end{array}
$$

RPC-enc: IML

```
let A = ...
let B =
  in(c, msg1);
  if 'p' = msg1{0, 1} then
  if len(msg1) ≥ 5 + msg1{1, 4} then
  if msg1{1, 4} ≤ 100 then
  if len(msg1) − (5 + msg1{1, 4}) ≤ 200 then
  let client1 = msg1{5, msg1{1, 4}} in
  let cipher1 = msg1{5 + msg1{1, 4}, len(msg1) − (5 + msg1{1, 4})} in
  if client1 = xClient then
  let msg2 = injbot^−1(D(cipher1, lookup(client1, serverID, db))) in
  if 'p' = msg2{0, 1} then
  if len(msg2) ≥ 5 + msg2{1, 4} then
  if msg2{1, 4} = 100 then
  let var2 = msg2{5, msg2{1, 4}} in
  event server_reply(client1, serverID, var2, response);
  if len(msg2) − (5 + msg2{1, 4}) = 16 then
  let kS = msg2{5 + msg2{1, 4}, len(msg2) − (5 + msg2{1, 4})} in
  new nonce1: seed;
  let cipher2 = E(response, kS, nonce1) in
  out(c, cipher2); 0 .
```

RPC-enc: Introduce Formatting Functions 1

From A:

$$conc1(x, y) := \text{'p'}|len(x)|x|y$$
$$conc2(x, y) := \text{'p'}|len(x)|x|y$$

From B:

$$parse1(x) := x\{5, x\{1, 4\}\}$$
$$parse2(x) := x\{5 + x\{1, 4\}, \text{len}(x) - (5 + x\{1, 4\})\}$$
$$parse3(x) := x\{5, x\{1, 4\}\}$$
$$parse4(x) := x\{5 + x\{1, 4\}, \text{len}(x) - (5 + x\{1, 4\})\}$$

RPC-enc: Introduce Formatting Functions 2

```
let A = ...
let B =
  in(c, msg1);
  if 'p' = msg1{0, 1} then
  if len(msg1) ≥ 5 + msg1{1, 4} then
  if msg1{1, 4} ≤ 100 then
  if len(msg1) − (5 + msg1{1, 4}) ≤ 200 then
  let client1 = msg1{5, msg1{1, 4}} in
  let cipher1 = msg1{5 + msg1{1, 4}, len(msg1) − (5 + msg1{1, 4})} in
  if client1 = xClient then
  let msg2 = injbot^−1(D(cipher1, lookup(client1, serverID, db))) in
  if 'p' = msg2{0, 1} then
  if len(msg2) ≥ 5 + msg2{1, 4} then
  if msg2{1, 4} = 100 then
  let var2 = msg2{5, msg2{1, 4}} in
  event server_reply(client1, serverID, var2, response);
  if len(msg2) − (5 + msg2{1, 4}) = 16 then
  let kS = msg2{5 + msg2{1, 4}, len(msg2) − (5 + msg2{1, 4})} in
  new nonce1: seed;
  let cipher2 = E(response, kS, nonce1) in
  out(c, cipher2); 0 .
```

RPC-enc: Introduce Formatting Functions 3

```
let A = ...
let B =
  in(c, msg1);
  if 'p' = msg1{0, 1} then
  if len(msg1) ≥ 5 + msg1{1, 4} then
  if msg1{1, 4} ≤ 100 then
  if len(msg1) − (5 + msg1{1, 4}) ≤ 200 then
  let client1 = parse1(msg1) in
  let cipher1 = parse2(msg1) in
  if client1 = xClient then
  let msg2 = injbot^−1(D(cipher1, lookup(client1, serverID, db))) in
  if 'p' = msg2{0, 1} then
  if len(msg2) ≥ 5 + msg2{1, 4} then
  if msg2{1, 4} = 100 then
  if len(msg2) − (5 + msg2{1, 4}) = 16 then
  let var2 = parse3(msg2) in
  event server_reply(client1, serverID, var2, response);
  let kS = parse4(msg2) in
  new nonce1: seed;
  let cipher2 = E(response, kS, nonce1) in
  out(c, cipher2); 0 .
```

RPC-enc: Type Inference

From user template:

$$injbot^{-1} : \texttt{bitstringbot} \to \texttt{bounded}_{200}$$
$$E : \texttt{bounded}_{200} \times \texttt{fixed}_{16} \times \texttt{seed} \to \texttt{bounded}_{200}$$

Therefore

$$parse_4 : \texttt{bounded}_{200} \to \texttt{fixed}_{16}$$

Similarly

$$conc_1 : \texttt{bounded}_{100} \times \texttt{bounded}_{200} \to \texttt{bitstring}$$
$$conc_2 : \texttt{fixed}_{100} \times \texttt{fixed}_{16} \to \texttt{bounded}_{200}$$
$$parse_1 : \texttt{bitstring} \to \texttt{bounded}_{200}$$
$$parse_2 : \texttt{bitstring} \to \texttt{bounded}_{200}$$
$$parse_3 : \texttt{bounded}_{200} \to \texttt{fixed}_{100}$$

RPC-enc: Type Checking 1

Context for $parse_4$:

$$\begin{aligned}\Phi_{parse_4} =&(\text{len}(msg_2) \leq 200) \wedge (\text{'p'} = msg_2\{0,1\}) \\ &\wedge (\text{len}(msg_2) \geq 5 + msg_2\{1,4\}) \\ &\wedge (msg_2\{1,4\} \leq 100) \\ &\wedge (\text{len}(msg_2) - (5 + msg_2\{1,4\}) = 16)\end{aligned}$$

The last clause implies that

$$\begin{aligned}&\text{len}(parse_4(msg_2)) \\ &\quad = \text{len}(msg_2\{5 + msg_2\{1,4\}, \text{len}(msg_2) - (5 + msg_2\{1,4\})\}) \\ &\quad = \text{len}(msg_2) - (5 + msg_2\{1,4\})\} = 16,\end{aligned}$$

thus the type of $parse_4$ is correct.

Example: Type Checking 2

let A = ...
let B =
  in(c, msg1);
  if 'p' = msg1{0, 1} **then**
  if len(msg1) ≥ 5 + msg1{1, 4} **then**
  if msg1{1, 4} ≤ 100 **then**
  if len(msg1) − (5 + msg1{1, 4}) ≤ 200 **then**
  let client1 = parse1(msg1) **in**
  let cipher1 = parse2(msg1) **in**
  if client1 = xClient **then**
  let msg2 = injbot^{-1}(D(cipher1, lookup(client1, serverID, db))) **in**
  if 'p' = msg2{0, 1} **then**
  if len(msg2) ≥ 5 + msg2{1, 4} **then**
  if msg2{1, 4} = 100 **then**
  if len(msg2) − (5 + msg2{1, 4}) = 16 **then**
  let var2 = parse3(msg2) **in**
  event server_reply(client1, serverID, var2, response);
  let kS = parse4(msg2) **in**
  new nonce1: seed;
  let cipher2 = E($\text{cast}_{\texttt{fixed}_{100} \to \texttt{bounded}_{200}}$(response), kS, nonce1) **in**
  out(c, cipher2); 0 .

RPC-enc: Parsing Safety 1

$$\Phi_{parse4} \implies \exists x : \mathtt{bounded}_{100}, y : \mathtt{bounded}_{200} : msg_2 = conc_2(x, y)$$

Additionally, $conc_2$ is injective and for all $x : \mathtt{bounded}_{100}$, $y : \mathtt{bounded}_{200}$

$$parse_3(conc_2(x, y)) = x,$$
$$parse_4(conc_2(x, y)) = y.$$

Example: Parsing Safety 2

let A = ...
let B =
  in(c, msg1);
  if 'p' = msg1{0, 1} **then**
  if len(msg1) ≥ 5 + msg1{1, 4} **then**
  if msg1{1, 4} ≤ 100 **then**
  if len(msg1) − (5 + msg1{1, 4}) ≤ 200 **then**
  let client1 = parse1(msg1) **in**
  let cipher1 = parse2(msg1) **in**
  if client1 = xClient **then**
  let msg2 = injbot^{-1}(D(cipher1, lookup(client1, serverID, db))) **in**
  if 'p' = msg2{0, 1} **then**
  if len(msg2) ≥ 5 + msg2{1, 4} **then**
  if msg2{1, 4} = 100 **then**
  if len(msg2) − (5 + msg2{1, 4}) = 16 **then**
  let var2 = parse3(msg2) **in**
  event server_reply(client1, serverID, var2, response);
  let kS = parse4(msg2) **in**
  new nonce1: seed;
  let cipher2 = E(cast$_{\texttt{fixed}_{100} \rightarrow \texttt{bounded}_{200}}$(response), kS, nonce1) **in**
  out(c, cipher2); 0 .

RPC-enc: Parsing Safety 3

let A = ...
let B =
  in(c, msg1);
  if 'p' = msg1{0, 1} **then**
  if len(msg1) ≥ 5 + msg1{1, 4} **then**
  if msg1{1, 4} ≤ 100 **then**
  if len(msg1) − (5 + msg1{1, 4}) ≤ 200 **then**
  let conc1(client1, cipher1) = msg1 **in**
  if client1 = xClient **then**
  let msg2 = injbot^{-1}(D(cipher1, lookup(client1, serverID, db))) **in**
  if 'p' = msg2{0, 1} **then**
  if len(msg2) ≥ 5 + msg2{1, 4} **then**
  if msg2{1, 4} = 100 **then**
  if len(msg2) − (5 + msg2{1, 4}) = 16 **then**
  let conc2(var2, kS) = msg2 **in**
  event server_reply(client1, serverID, var2, response);
  new nonce1: seed;
  let cipher2 = E(cast$_{\mathtt{fixed}_{100} \to \mathtt{bounded}_{200}}$(response), kS, nonce1) **in**
  out(c, cipher2); 0 .

RPC-enc: Cleaning Up 1

let A = ...
let B =
  in(c, msg1);
  if 'p' = msg1{0, 1} **then**
  if len(msg1) ≥ 5 + msg1{1, 4} **then**
  if msg1{1, 4} ≤ 100 **then**
  if len(msg1) − (5 + msg1{1, 4}) ≤ 200 **then**
  let conc1(client1, cipher1) = msg1 **in**
  if client1 = xClient **then**
  let msg2 = injbot^{-1}(D(cipher1, lookup(client1, serverID, db))) **in**
  if 'p' = msg2{0, 1} **then**
  if len(msg2) ≥ 5 + msg2{1, 4} **then**
  if msg2{1, 4} = 100 **then**
  if len(msg2) − (5 + msg2{1, 4}) = 16 **then**
  let conc2(var2, kS) = msg2 **in**
  event server_reply(client1, serverID, var2, response);
  new nonce1: seed;
  let cipher2 = E(cast$_{\mathtt{fixed}_{100} \rightarrow \mathtt{bounded}_{200}}$(response), kS, nonce1) **in**
  out(c, cipher2); 0 .

RPC-enc: Cleaning Up 2

let A = ...
let B =
  in(c, msg1);
  let conc1(client1, cipher1) = msg1 **in**
  if client1 = xClient **then**
  let msg2 = injbot^{-1}(D(cipher1, lookup(client1, serverID, db))) **in**
  let conc2(var2, kS) = msg2 **in**
  event server_reply(client1, serverID, var2, response);
  new nonce1: seed;
  let cipher2 = E(cast$_{\mathtt{bounded}_{100} \to \mathtt{bounded}_{200}}$(response), kS, nonce1) **in**
  out(c, cipher2); 0 .

Formatting Abstraction: Summary

To simplify IML to CV:

- Replace formatting expressions by function symbols.
- Prove that formatting functions are type-safe *in context*. Introduce type casts when necessary and justified.
- Recognize places where parsing is *safe* and introduce pattern matching there.
- Remove arithmetic conditionals.

Soundness

CryptoVerif

- Special value $\perp$.
- Arbitrary security parameter k.
- Must be type-safe.
- $\mathrm{cvinsec}(Q, \rho, k)$

C/IML

- No special value, failure stops execution.
- Fixed security parameter k_0.
- No types for bitstrings.
- $\mathrm{insec}(Q, \rho)$

Theorem (CryptoVerif Translation is Sound)

Let Q be an IML process that successfully translates to a CryptoVerif process $\tilde{Q}$. Then for any evaluation context C and any correspondence property ρ

$$\mathrm{insec}(C[Q], \rho) \leq \mathrm{cvinsec}(C[\tilde{Q}], \rho, k_0).$$

Tool Support in Cryptography

There is a trend of developing tool support for cryptographic tools: ProVerif, CryptoVerif, EasyCrypt, F7. Personal experience: CryptoVerif is great!

- Makes proofs easier. Halevi, 2005:
  "The problem is that as a community, we generate more proofs than we carefully verify (and as a consequence some of our published proofs are incorrect)."
- Have a common language for security specifications, modelling protocols, and doing proofs.

Is cryptography mature enough, so that instead of writing proofs we can start writing tools to do proofs for us?

Current Status

Implementation available from
`https://github.com/tari3x/csec-modex`

Csec-challenge:
`http://research.microsoft.com/csec-challenge`

Working on verifying PolarSSL.

Open questions:

- Observational equivalence/simulation.
- Support arbitrary control flow.

Thank you!

Bibliography


Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens.
Extracting and verifying cryptographic models from C protocol code by symbolic execution.
In *18th ACM Conference on Computer and Communications Security (CCS 2011)*, 2011.
Full version available at `http://arxiv.org/abs/1107.1017`.