# Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution

**Mihhail Aizatulin**[1]
supervised by
Andrew Gordon[23], Jan Jürjens[4], Bashar Nuseibeh[1]

[1]The Open University

[2]Microsoft Research Cambridge

[3]University of Edinburgh

[4]Dortmund University

November 2011

# The Goal

Problem: we often verify formal models of cryptographic protocols, but what we rely on are their implementations.
We bridge the gap by extracting high-level (pi calculus) models straight from C code. Support following scenarios:

- Given a legacy implementation of a protocol, learn what the implementation really does and prove security.
- When implementing a new protocol make sure that you did so without mistakes.

We check trace properties such as authentication and weak secrecy, aiming to be automated and sound. We assume correctness of cryptographic primitives.

# Background

Types of properties and languages.

| | Low-Level (C, Java) | High-Level (F#) | Formal ($\pi$, LySa) |
|---|---|---|---|
| low-level (NULL dereference, division by zero) | • VCC<br>• Frama-C<br>• ESC/Java<br>• SLAM | N/A | N/A |
| high-level (secrecy, authentication) | • CSur<br>• JavaSec<br>• ASPIER<br>• csec-modex | • F7/F*<br>• fs2pv/fs2cv | • ProVerif<br>• CryptoVerif<br>• AVISPA<br>• LySatool |

# Results

- Three implementations (1300 LOC) verified in the symbolic model.
- One of them also verified in the computational model by application of a computational soundness result.
- Found 3 flaws in a Microsoft Research implementation of a smart metering protocol (1000 LOC) (all fixed now).

Metering flaw:

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

Extracted model:

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

# Demo

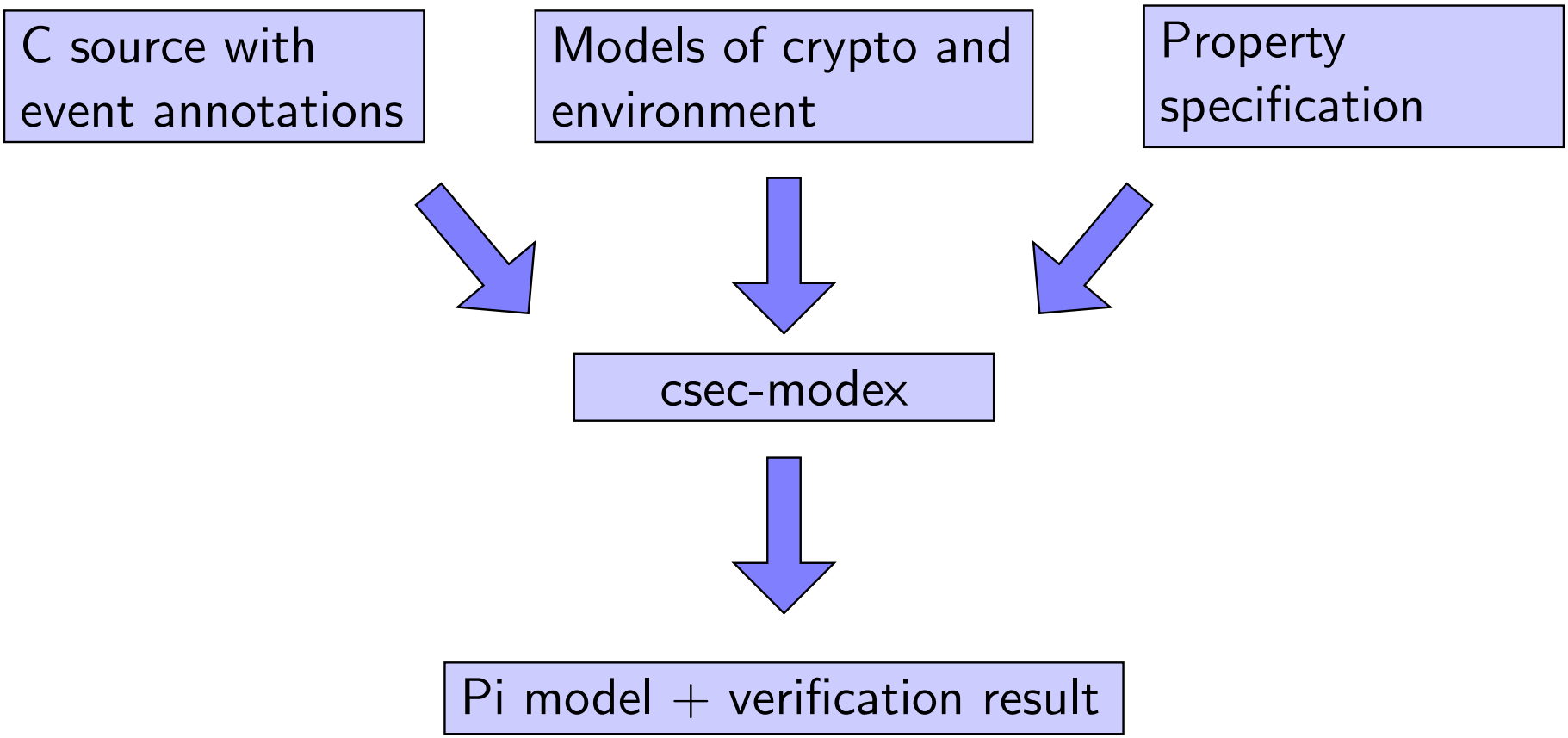
Abstract protocol:

$$A \xrightarrow{m, hmac(m, k_{AB})} B.$$

Concrete protocol:

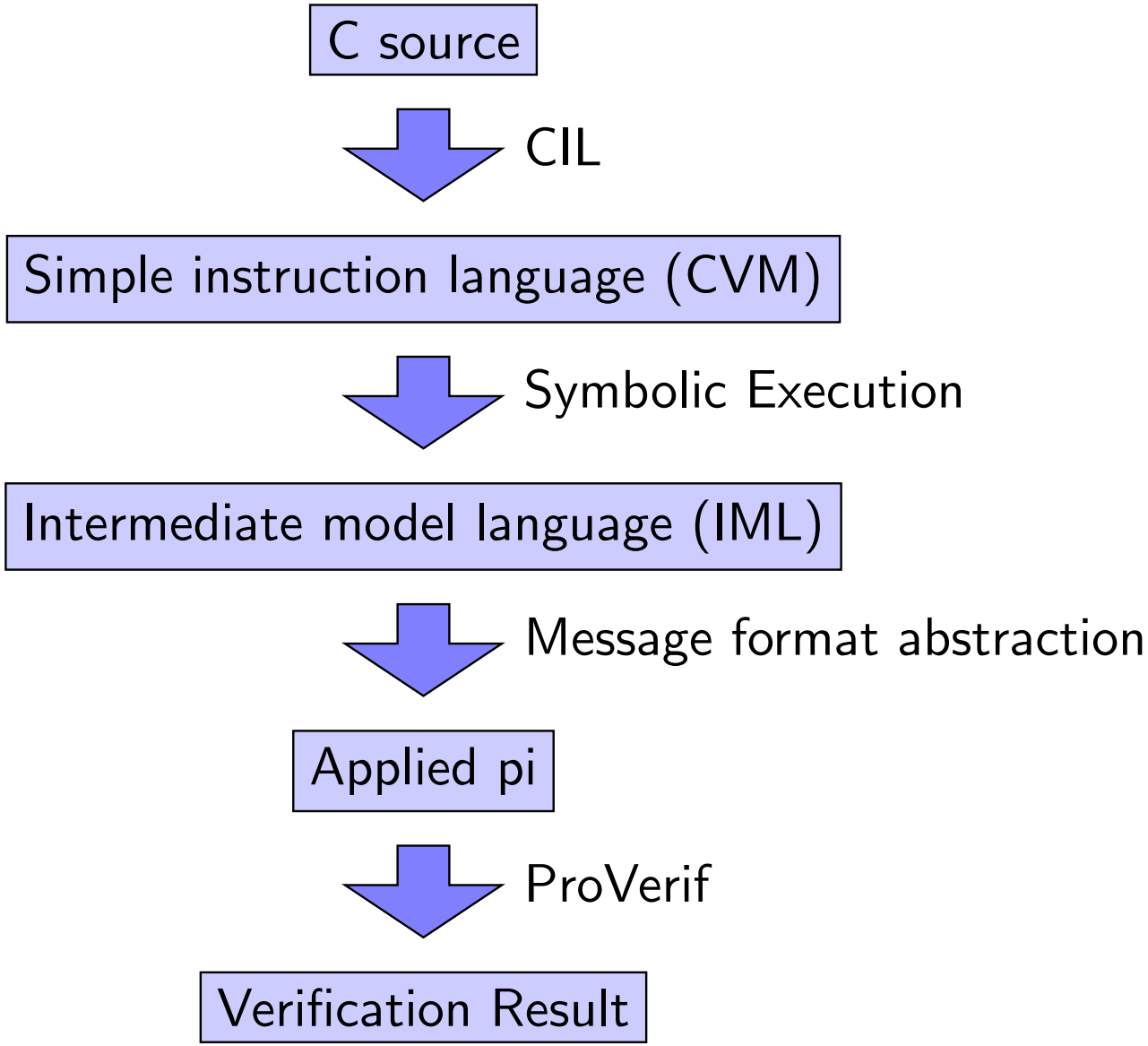
$$A \xrightarrow{\text{len}(m)|1|m|hmac(\text{len}(m)|2|m, k_{AB})} B.$$

# Overview: What




Major limitation: So far the symbolic execution only follows a single path in the program.

# Overview: How



# Correctness (1)

**Definition (Security of protocols)**

Given a protocol $P$, attacker $E$, trace property $\rho$, and resource bound $t \in \mathbb{N}$ let

$$\mathrm{insec}(P, E, \rho, t)$$

be the success probability of $E$ against $P$ with respect to $\rho$, given resources $t$.

# Correctness (2)

**Theorem (Soundness of Model Extraction)**

*For any environment process* $P_E[\cdot,\cdot]$*, attacker* $E$*, property* $\rho$*, and resource bound* $t$

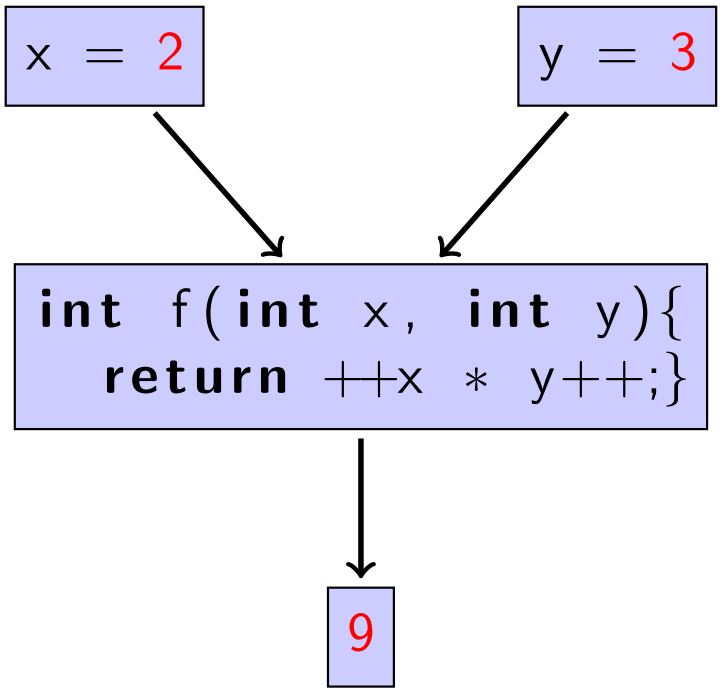
$$\begin{aligned}&\mathrm{insec}(P_E[\texttt{client.c},\texttt{server.c}],E,\rho,t)\\ \leq\ &\mathrm{insec}(P_E[\texttt{client.iml},\texttt{server.iml}],E,\rho,p_1(t))\\ \leq\ &\mathrm{insec}(P_E[\texttt{client.pv},\texttt{server.pv}],E,\rho,p_2(t))\end{aligned}$$

*with some fixed polynomials* $p_1$ *and* $p_2$*.*

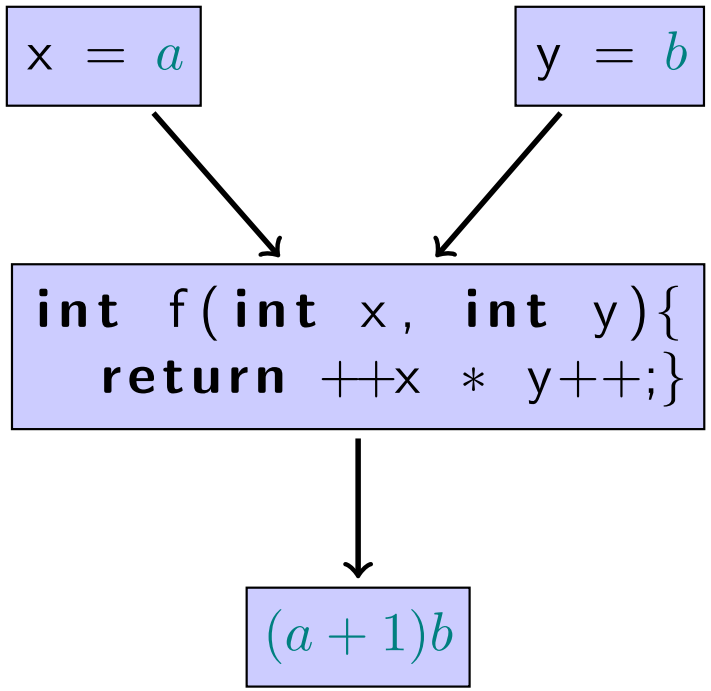
# Symbolic Execution: Basic Idea

Symbolic execution is a tool to simplify programs and extract their meaning.

Concrete:




Symbolic:



# Symbolic Execution with Symbolic Lengths (1)

Introducing new values:

```
size_t key_len; void * key;
key = malloc(MAX_KEY_LEN);
keygen(key, &key_len);
```

$\text{stack}\, key \rightsquigarrow \text{ptr}(\text{heap}\, 1, 0)$

$\text{heap}\, 1 \rightsquigarrow k$, for some fresh $k$

$\text{stack}\, key\_len \rightsquigarrow \text{len}(k)$

Generate "$(\nu k)$;" in the IML model. The way of modelling keys is specified in keygen_proxy().

# Symbolic Execution with Symbolic Lengths (2)

Pointer arithmetic:

---

$\text{stack}\, len \rightsquigarrow \text{len}(x)$

---

```
void * msg = malloc(msg_len);
void * p = msg + sizeof(len) + len;
```

---

$\text{stack}\, msg \rightsquigarrow \text{ptr}(\text{heap}\, 2, 0)$

$\text{stack}\, p \rightsquigarrow \text{ptr}(\text{heap}\, 2, 4 + \text{len}(x))$

# Symbolic Execution with Symbolic Lengths (3)

Writing through pointers:

---

$\mathrm{stack}\, p \rightsquigarrow \mathrm{ptr}(\mathrm{heap}\, 2, 4 + \mathrm{len}(x))$

$\mathrm{heap}\, 2 \rightsquigarrow \mathrm{len}(x)|x|y$

Fact: $\mathrm{len}(y) = \mathrm{len}(k)$

---

```
xor(p, key, key_len);
```

---

$\mathrm{heap}\, 2 \rightsquigarrow \mathrm{len}(x)|x|y \oplus k$

# Symbolic Execution with Symbolic Lengths (4)

Output:

---

$$\text{stack}\, msg \rightsquigarrow \text{ptr}(\text{heap}\, 2, 0)$$
$$\text{heap}\, 2 \rightsquigarrow \text{len}(x)|x|y \oplus k$$
$$\text{stack}\, msg\_len \rightsquigarrow 4 + \text{len}(x) + \text{len}(y)$$

---

```
write(msg, msg_len);
```

---

Generate IML "**out**$(\text{len}(x)|x|y \oplus k)$;".

# Symbolic Execution with Symbolic Lengths (5)

Extracting a substring:

```
void * buf = malloc(MAX_LEN);
size_t len = read(buf, MAX_LEN);
size_t field_len = *((size_t *) buf);
```

$\text{stack}\, \mathit{field\_len} \rightsquigarrow x\{0, 4\}$

Where $x$ is a fresh variable and we generate IML "**in**($x$);".

# Symbolic Execution with Symbolic Lengths (6)

Extracting a substring:

---

$\text{stack}\, \mathit{field\_len} \rightsquigarrow x\{0, 4\}$

---

```
void * field = malloc(field_len);
memcpy(field, buf + sizeof(field_len), field_len)
```

---

$\text{stack}\, \mathit{field} \rightsquigarrow \text{ptr}(\text{heap}\, 3, 0)$

$\text{heap}\, 3 \rightsquigarrow x\{4, x\{0, 4\}\}$

# Symbolic Execution: Example

```
#define MAC_LEN 20
#define MAX_LEN 1000
int len;
```

# Symbolic Execution: Example

```
#define MAC_LEN 20
#define MAX_LEN 1000
int len;
read(&len, sizeof(len));
```

---

stack $len \rightsquigarrow r_1$ $\quad\quad$ $\mathrm{len}(r_1) = 4$

---

**in**$(r_1)$;

# Symbolic Execution: Example

```
#define MAC_LEN 20
#define MAX_LEN 1000
int len;
read(&len, sizeof(len));
if((len < MAC_LEN) || (len > MAX_LEN)) exit();
```

---

stack $len \rightsquigarrow r_1$ $\quad\quad$ $\mathrm{len}(r_1) = 4$

---

**in**$(r_1)$;
**if** $\neg((r_1 < 20) \vee (r_1 > 1000))$ **then**

# Symbolic Execution: Example

```
#define MAC_LEN 20
#define MAX_LEN 1000
int len;
read(&len, sizeof(len));
if((len < MAC_LEN) || (len > MAX_LEN)) exit();
char * buf = malloc(len + MAC_LEN);
```

---

stack $len \rightsquigarrow r_1$ $\qquad\qquad$ $\mathrm{len}(r_1) = 4$
stack $buf \rightsquigarrow \mathrm{ptr}(\mathrm{heap}\ 1,\ 0)$

---

**in**$(r_1)$;
**if** $\neg((r_1 < 20) \vee (r_1 > 1000))$ **then**

# Symbolic Execution: Example

```
#define MAC_LEN 20
#define MAX_LEN 1000
int len;
read(&len, sizeof(len));
if((len < MAC_LEN) || (len > MAX_LEN)) exit();
char * buf = malloc(len + MAC_LEN);
read(buf, len);
```

---

$\text{stack } \mathit{len} \rightsquigarrow r_1$ $\qquad$ $\text{len}(r_1) = 4$

$\text{stack } \mathit{buf} \rightsquigarrow \text{ptr}(\text{heap } 1, 0)$ $\qquad$ $\text{len}(r_2) = r_1$

$\text{heap } 1 \rightsquigarrow r_2$

---

**in**$(r_1)$;
**if** $\neg((r_1 < 20) \vee (r_1 > 1000))$ **then**
**in**$(r_2)$;

# Symbolic Execution: Example

```
#define MAC_LEN 20
#define MAX_LEN 1000
int len;
read(&len, sizeof(len));
if((len < MAC_LEN) || (len > MAX_LEN)) exit();
char * buf = malloc(len + MAC_LEN);
read(buf, len);
hmac(buf, buf + len, len);
```

---

| | |
|---|---|
| stack $len \rightsquigarrow r_1$ | $\text{len}(r_1) = 4$ |
| stack $buf \rightsquigarrow \text{ptr}(\text{heap } 1, 0)$ | $\text{len}(r_2) = r_1$ |
| heap $1 \rightsquigarrow r_2 \vert hmac(r_2)$ | $\text{len}(hmac(r_2)) = 20$ |

---

**in**$(r_1)$;
**if** $\neg((r_1 < 20) \vee (r_1 > 1000))$ **then**
**in**$(r_2)$;

# Symbolic Execution: Example

```
#define MAC_LEN 20
#define MAX_LEN 1000
int len;
read(&len, sizeof(len));
if((len < MAC_LEN) || (len > MAX_LEN)) exit();
char * buf = malloc(len + MAC_LEN);
read(buf, len);
hmac(buf, buf + len, len);
if(memcmp(buf, buf + len, MAC_LEN) == 0)
```

---

| | |
|---|---|
| stack $len \rightsquigarrow r_1$ | $\text{len}(r_1) = 4$ |
| stack $buf \rightsquigarrow \text{ptr}(\text{heap } 1, 0)$ | $\text{len}(r_2) = r_1$ |
| heap $1 \rightsquigarrow r_2 \vert hmac(r_2)$ | $\text{len}(hmac(r_2)) = 20$ |

---

**in**$(r_1)$;
**if** $\neg((r_1 < 20) \vee (r_1 > 1000))$ **then**
**in**$(r_2)$;
**if** $r_2\{0, 20\} = hmac(r_2)$ **then**

# Symbolic Execution: Example

```
#define MAC_LEN 20
#define MAX_LEN 1000
int len;
read(&len, sizeof(len));
if((len < MAC_LEN) || (len > MAX_LEN)) exit();
char * buf = malloc(len + MAC_LEN);
read(buf, len);
hmac(buf, buf + len, len);
if(memcmp(buf, buf + len, MAC_LEN) == 0)
  event("wow", buf, MAC_LEN);
```

---

| | |
|---|---|
| stack $len \rightsquigarrow r_1$ | $\text{len}(r_1) = 4$ |
| stack $buf \rightsquigarrow \text{ptr}(\text{heap } 1, 0)$ | $\text{len}(r_2) = r_1$ |
| heap $1 \rightsquigarrow r_2 \vert hmac(r_2)$ | $\text{len}(hmac(r_2)) = 20$ |

---

**in**$(r_1)$;
**if** $\neg((r_1 < 20) \vee (r_1 > 1000))$ **then**
**in**$(r_2)$;
**if** $r_2\{0, 20\} = hmac(r_2)$ **then**
**event** $wow(r_2\{0, 20\})$

# Message Format Abstraction (1)

An IML model:

**let** A $=$
  **in**$(x)$;
  **event**$(send(x))$;
  **out**$(\mathrm{len}(x)|1|x|hmac(x, k_{AB}))$.

**let** B $=$
  **in**$(m)$;
  **if** $\mathrm{len}(m) < m\{0,4\} + 5$ **then**
  **if** $m\{4,1\} = 1$ **then**
  **let** $x = m\{5, m\{0,4\}\}$ **in**
  **let** $h = m\{5 + m\{0,4\}, \mathrm{len}(m) - 5 + m\{0,4\}\}$ **in**
  **if** $h = hmac(x, k_{AB})$ **then**
  **event**$(accept(x))$.

P $= !(\nu k_{AB};$ (!A | !B)).

# Message Format Abstraction (2)

We prove that IML bitstring manipulation expressions implement pairing.

$$
\begin{aligned}
c_1/2 :=& \lambda xy.\, \mathrm{len}(x)|1|x|y, \\
d_1/1 :=& \lambda x.\textbf{if } \mathrm{len}(m) < x\{0,4\} + 5 \textbf{ then} \\
& \qquad \textbf{if } x\{4,1\} = 1 \textbf{ then } x\{5, x\{0,4\}\} \textbf{ else } \bot, \\
d_2/1 :=& \lambda x.\textbf{if } \ldots \textbf{ then } x\{5 + x\{0,4\}, \mathrm{len}(x) - 5 + x\{0,4\}\} \textbf{ else } \bot.
\end{aligned}
$$

Properties:

- all concatenation functions have disjoint ranges,
- for all $x$ and $y$: $d_1(c_1(x,y)) = x$ and $d_2(c_1(x,y)) = y$,
- whenever $d_1(m) \neq \bot$ or $d_2(m) \neq \bot$, there exist $x, y$ such that $m = c_1(x,y)$.

# Message Format Abstraction (3)

Pi calculus translation of the IML model:

**reduc** $d_1(c_1(x, y)) = x$; $d_2(c_1(x, y)) = y$.

**query ev**:$accept(x) \Longrightarrow$ **ev**:$send(x)$.

**let** A $=$
  **in**$(x)$;
  **event**$(send(x))$;
  **out**$(c_1(x, hmac(x, k_{AB})))$.
**let** B $=$
  **in**$(m)$;
  **let** $x = d_1(m)$ **in**
  **let** $h = d_2(m)$ **in**
  **if** $h = hmac(x, k_{AB})$ **then**
  **event**$(accept(x))$.

**process** !$(\nu k_{AB}$; (!A | !B)).

# Current Status

| | C LOC | IML LOC | outcome | result type | time |
|---|---|---|---|---|---|
| simple mac | $\sim 250$ | 12 | verified | symbolic | 4s |
| RPC | $\sim 600$ | 35 | verified | symbolic | 5s |
| NSL | $\sim 450$ | 40 | verified | computat. | 5s |
| CSur | $\sim 600$ | 20 | flaws found | — | 5s |
| Metering | $\sim 1000$ | 51 | flaws found | — | 15s |

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

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

Working on:

- Using CryptoVerif for verification of models, removing need for computational soundness results.
- Adding support for arbitrary control flow.

# Thank you!