

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

Mihhail Aizatulin¹

supervised by

Andrew Gordon², Jan Jürjens³, Bashar Nuseibeh¹

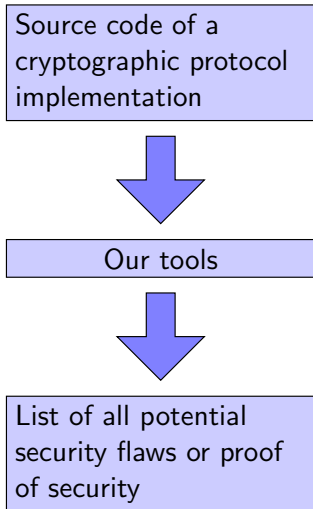
¹The Open University

²Microsoft Research Cambridge

³Dortmund University

Cryptoforma

October 22, 2010



Our goal is a tool for security analysis at implementation level.

The analysis should be

- automated,
- sound (not miss any bugs),
- scalable (aim to verify OpenSSL or Kerberos).

We assume that

- cryptographic primitives are implemented correctly.

OpenSSL library bug, January 2009:

```
int check_certificate(){
    ...
    if(certificate_malformed)
        return -1;
    else if(!certificate_check_ok)
        return 0;
    else return 1;}

// later in code:
...
if(check_certificate()) // oops!
{
    trust_certificate();
}
```

	Machine Languages (C, Java)	Formal Languages (π -calculus, LySa)
low-level properties (NULL dereference, division by zero)		
high-level properties (secrecy, authentication)		

Background

There has been great progress in

- static software analysis,

	Machine Languages (C, Java)	Formal Languages (π -calculus, LySa)
low-level properties (NULL dereference, division by zero)	<ul style="list-style-type: none">• VCC• ESC/Java• SLAM	<ul style="list-style-type: none">• Frama-C
high-level properties (secrecy, authentication)		

Background

There has been great progress in

- static software analysis,
- verification of protocol specifications.

	Machine Languages (C, Java)	Formal Languages (π -calculus, LySa)
low-level properties (NULL dereference, division by zero)	<ul style="list-style-type: none">• VCC• ESC/Java• SLAM	<ul style="list-style-type: none">• Frama-C
high-level properties (secrecy, authentication)		<ul style="list-style-type: none">• ProVerif/CryptoVerif• AVISPA• LySatool

Background

There has been great progress in

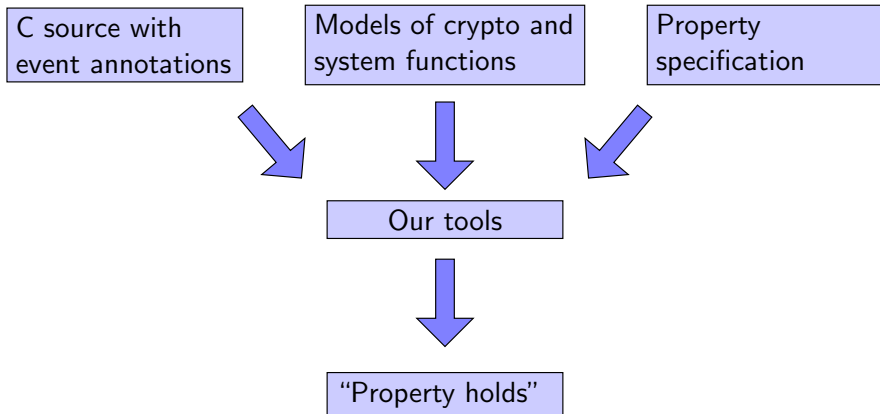
- static software analysis,
- verification of protocol specifications.

But so far very little progress in where the two meet.

	Machine Languages (C, Java)	Formal Languages (π -calculus, LySa)
low-level properties (NULL dereference, division by zero)	<ul style="list-style-type: none">• VCC• ESC/Java• SLAM	<ul style="list-style-type: none">• Frama-C
high-level properties (secrecy, authentication)	<ul style="list-style-type: none">• CSur• JavaSec	<ul style="list-style-type: none">• ProVerif/CryptoVerif• AVISPA• LySatool

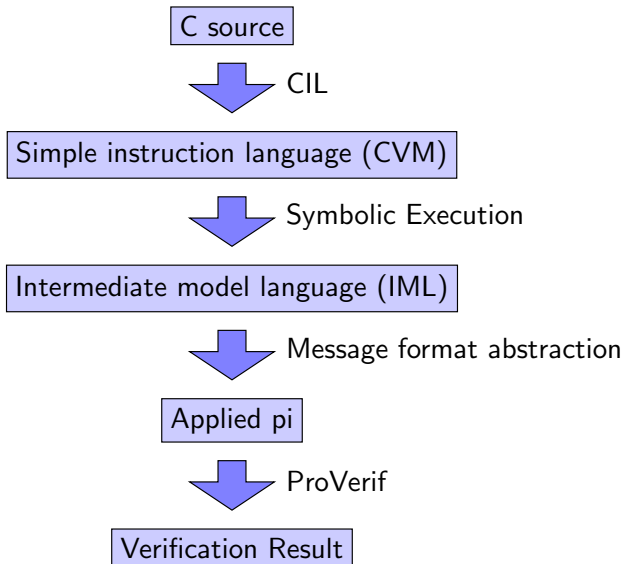
• Our Goal

Overview: What



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

Overview: How



Abstract protocol:

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

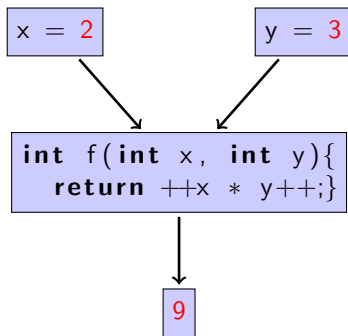
From C to Intermediate Model Language (IML): like computational pi calculus, but with extra bitstring operations:

$L, M, N ::=$	expression
$[00..FF]^*$	concrete bitstring
x, y, z	variable
$f(M_1, \dots, M_n)$	crypto function
$M N$	concatenation
$M\{N, L\}$	substring extraction
$\text{len}(M)$	string length
$\text{op}(M_1, \dots, M_n)$	primitive operation

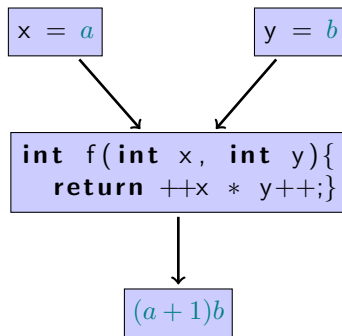
Symbolic Execution: Idea

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

Concrete:



Symbolic:



Symbolic Execution: Cryptographic Challenge

In all previous symbolic execution frameworks a symbol corresponds to a fixed-width variable.

Key point 1: Symbolic execution for cryptography needs to assign symbols to bitstrings, i.e. work with symbols whose length is itself symbolic.

Symbolic Execution: Example

```
int len;
```

Symbolic Execution: Example

Apply "read" 0; StackPtr "len"; Const 4; Store;

```
int len;  
read(&len; sizeof(len));
```

Stack len \rightsquigarrow read[1]<4>

Symbolic Execution: Example

```
StackPtr "len"; Load; Const 20; Apply "<" 2; Test;
```

```
int len;  
read(&len; sizeof(len));  
if(len < SHA1.LEN) exit();
```

```
Stack len ↔ read[1]<4>
```

```
if read1 >= 20 then
```

Symbolic Execution: Example

```
NewHeapPtr; StackPtr "buf"; Store;
```

```
int len;  
read(&len; sizeof(len));  
if(len < SHA1_LEN) exit();  
char * buf = malloc(len + 20);
```

```
Stack len  $\rightsquigarrow$  read[1]<4>
```

```
Stack buf  $\rightsquigarrow$  Heap 1
```

```
if read1  $\geq$  20 then
```

Symbolic Execution: Example

Apply "read" 0; StackPtr "buf"; StackPtr "len"; Load; Store;

```
int len;
read(&len; sizeof(len));
if(len < SHA1_LEN) exit();
char * buf = malloc(len + 20);
read(buf, len);
```

```
Stack len  ⇔ read[1]<4>
Stack buf  ⇔ Heap 1
Heap 1     ⇔ read[2]<read[1]<4>>
```

```
if read1 >= 20 then
```

Symbolic Execution: Example

```
StackPtr "buf"; StackPtr "len"; Load; Apply "hmacsha1" 2;  
StackPtr "buf"; StackPtr "len"; Apply "+" 2; Const 20; Store;
```

```
int len;  
read(&len; sizeof(len));  
if(len < SHA1_LEN) exit();  
char * buf = malloc(len + 20);  
read(buf, len);  
hmacsha1(buf, buf + len, len);
```

```
Stack len  $\rightsquigarrow$  read[1]<4>  
Stack buf  $\rightsquigarrow$  Heap 1  
Heap 1  $\rightsquigarrow$  read[2]<read[1]<4>>|hmacsha1(read[2]<read[1]<4>>><20>
```

```
if read1  $\geq$  20 then
```

Symbolic Execution: Example

```
StackPtr "buf"; StackPtr "len"; Apply "+" 2; Const 20; Load;  
StackPtr "buf"; Const 20; Load;  
Apply "cmp" 2; Const 0; Apply "==" 0; Test;
```

```
int len;  
read(&len; sizeof(len));  
if(len < SHA1_LEN) exit();  
char * buf = malloc(len + 20);  
read(buf, len);  
hmacsha1(buf, buf + len, len);  
if(memcmp(buf, buf + len, 20) == 0)
```

```
Stack len  $\rightsquigarrow$  read[1]<4>  
Stack buf  $\rightsquigarrow$  Heap 1  
Heap 1  $\rightsquigarrow$  read[2]<read[1]<4>>|hmacsha1(read[2]<read[1]<4>>><20>
```

```
if read1  $\geq$  20 then  
if read2{0, 20} = hmacsha1(read2) then
```

Symbolic Execution: Example

```
StackPtr "buf"; Const 20; Load; Event "wow";
```

```
int len;  
read(&len; sizeof(len));  
if(len < SHA1_LEN) exit();  
char * buf = malloc(len + 20);  
read(buf, len);  
hmacsha1(buf, buf + len, len);  
if(memcmp(buf, buf + len, 20) == 0)  
    event("wow", buf, 20);
```

```
Stack len  $\rightsquigarrow$  read[1]<4>
```

```
Stack buf  $\rightsquigarrow$  Heap 1
```

```
Heap 1  $\rightsquigarrow$  read[2]<read[1]<4>>|hmacsha1(read[2]<read[1]<4>>><20>
```

```
if read1  $\geq$  20 then
```

```
if read2{0, 20} = hmacsha1(read2) then
```

```
event wow(read2{0, 20})
```

For a CVM program P

$$\underbrace{[[[P]_S]_C]}_{IML} \simeq [[P]_C,$$

where \simeq means that the I/O-behaviour is the same.

We would like to abstract away concatenation and parsing patterns as pi calculus constructors and destructors:

$$c_1/2 := \lambda xy. \text{len}(x)|1|x|y$$

$$d_1/1 := \lambda x.x\{5, x\{0, 4\}\}$$

Key point 2: Need computational soundness in presence of such bitstring-mangling primitives.

Major limitation so far: only consider linear control flow.

Done:

- Implemented model extraction and verification for several protocol examples, including CSur.

Now working on:

- Extracting a model from OpenSSL.
- Formulating soundness proofs.

Next year:

- Add support for arbitrary control flow.

Thank you!