Verifying Implementations of Security Protocols in C

Mihhail Aizatulin\textsuperscript{1}

supervised by
Andrew Gordon\textsuperscript{2}, Jan Jürjens\textsuperscript{3}, Bashar Nuseibeh\textsuperscript{1}

\textsuperscript{1}The Open University
\textsuperscript{2}Microsoft Research Cambridge
\textsuperscript{3}Dortmund University

CRC PhD Student Conference
Open University
June 3-4, 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,
- cryptography is unbreakable (for now).
OpenSSL library bug, January 2009:

```c
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();
}
```
An attacker can craft a malformed certificate, pretending to be someone else:

Now the attacker can impersonate the client and the bank to each other:

Client → Password → Attacker (≡ Internet) ← Password

Attacker (≡ Internet) → Password + “get $$$” → paypal.com
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.

<table>
<thead>
<tr>
<th>Low-level properties (NULL dereference, division by zero)</th>
<th>Machine Languages (C, Java)</th>
<th>Formal Languages ($\pi$-calculus, LySa)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VCC</td>
<td>Frama-C</td>
</tr>
<tr>
<td></td>
<td>ESC/Java</td>
<td>SLAM</td>
</tr>
<tr>
<td></td>
<td>CSur</td>
<td></td>
</tr>
<tr>
<td>High-level properties (secrecy, authentication)</td>
<td>Our Goal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ProVerif/CryptoVerif</td>
<td>AVISPA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LySatool</td>
</tr>
</tbody>
</table>

M. Aizatulin
Verifying Implementations of Security Protocols in C
ProVerif is a protocol verifier we would like to use. Its models are much more abstract than C.

<table>
<thead>
<tr>
<th>ProVerif</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M ::= \text{expression}$</td>
<td>$M, N ::= \text{expression}$</td>
</tr>
<tr>
<td>$a, b, c$</td>
<td>$0, \ldots, \text{'}a\text{',} \ldots$</td>
</tr>
<tr>
<td>$x, y, z$</td>
<td>$M \text{ op } N$</td>
</tr>
<tr>
<td>$f(M_1, \ldots, M_n)$</td>
<td>$u$</td>
</tr>
<tr>
<td></td>
<td>$&amp;u$</td>
</tr>
<tr>
<td>$u ::= \text{lvalue}$</td>
<td></td>
</tr>
<tr>
<td>$x, y, z$</td>
<td></td>
</tr>
<tr>
<td>$\ast u$</td>
<td></td>
</tr>
<tr>
<td>$u.field$</td>
<td></td>
</tr>
<tr>
<td>$u[i]$</td>
<td></td>
</tr>
</tbody>
</table>
Symbolic execution is a tool to simplify programs and extract their meaning.

Concrete:
- \( x = 2 \)
- \( y = 3 \)

```
int f(int x, int y){
    return ++x * y++;}
```

Symbolic:
- \( x = a \)
- \( y = b \)

```
int f(int x, int y){
    return ++x * y++;}
```

\( 9 \)

\( (a + 1)b \)
Method (I)

\[ A \xrightarrow{m, \text{hash}(m, k_{\text{shared}})} B. \]

Client side implemented as:

```c
client(char * payload, int payload_len) {
    int msg_len = 5 + len + SHA1_LEN;
    char * msg = malloc(msg_len);
    char * p = msg;
    *p = len; p += 4;  // add length
    *(p++) = 1;        // add the tag
    memcpy(payload, p, len);  // add the payload
    sha1(msg, 5 + len, p);  // add the hash
    send(msg, msg_len);    // send
}
```
Method (II)

```c
client(char * payload, int payload_len){
    int msg_len = 5 + len + SHA1_LEN;
    char * msg = malloc(msg_len);
    char * p = msg;
    *p = len; p += 4;  // add length
    *(p++) = 1;        // add the tag
    memcpy(payload, p, len); // add the payload
    sha1(msg, 5 + len, p); // add the hash
    send(msg, msg_len);   // send
}
```

Symbolic Execution

out(len(payload)|01|payload
    |sha1(len(payload)|01|payload, k_{shared}))
Method (III)

...  

Symbolic Execution

\[ \text{out}(\text{len}(payload)\|01\| \text{payload} \| \text{sha1}(\text{len}(payload)\|01\| \text{payload}, k_{\text{shared}})) ]

Format Abstraction

\[ \text{out}(f_1(\text{payload}, \text{hash}_1(\text{payload}, k_{\text{shared}}))) ]

ProVerif (+ Server Side)

Integrity verified.
Done:
- Implemented symbolic execution for fixed bitstring lengths and linear control flow.
- Proved correctness of format abstraction.

To do:
- Implement symbolic execution for variable bitstring lengths.
- Implement format abstraction.
- Add support for arbitrary control flow.
Thank you!