Verifying Implementations of Security Protocols in C

Mihhail Aizatulin (avatar@hot.ee)

1. Goal

Our goal is a tool for cryptographic security analysis at implementation level:

- Source code of a cryptographic protocol implementation
- Our tools
- List of all potential security flaws or proof of security

2. Motivation: an Attack Example

Here is an example vulnerability that we would like to prevent. In January 2009 a flaw in the OpenSSL library was discovered, permitting a corrupt certificate to be recognised as valid. A function return value is wrongly interpreted:

```c
int check_certificate() {
    // later in code:
    ... 
    if (certificate_malformed) {
        if (check_certificate()) // oops!
            return -1;
        else if (!certificate_check_ok) 
            trust_certificate();
    return 0; 
    else return 1; 
}
```

Notice how the -1 return value passes the validity test! An attacker can craft a malformed certificate, pretending to be someone else:

Client: Paypal.com Certificate?

Malformed Certificate

Attacker (= Internet)

paypal.com

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

Client: Password?

Password?

Attacker (= Internet)

Password + “get $$$”

paypal.com

We would like to use formal methods to prove absence of similar flaws.

3. Assumptions

We assume that cryptographic primitives (encryption, hashing) are implemented correctly and are unbreakable.

4. Background

Very little has been done for crypto-verification of low-level languages. We rely on tools for proving security from high-level specifications of protocols. Currently we are using ProVerif.

5. Challenges

We cannot simply reuse existing program verification tools, because we don’t just verify a program, but a system, consisting of several instances of a program and an unknown attacker.

6. Symbolic Execution

Symbolic execution is a tool to simplify programs and extract their meaning. The program is “executed” with symbols as input, instead of numbers:

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

Concrete:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Symbolic:

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
</tr>
</tbody>
</table>

We use symbolic execution to derive the formats of messages that the program generates.

7. Proposed Method

We propose to use symbolic execution to extract a high-level model of the protocol and then use existing tools to verify it. Suppose we use shared key hashing for message integrity:

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

The client side implementation could look as follows. Note the extra data added to the message: the payload length and a tag.

```c
client(char * payload, int payload_len){
    int msg_len = 5 + len + SHA1_LEN;
    char * msg = malloc(msg_len);
    char * p = msg;
    p += 4;
    int x = 0;
    int f(int x, int y){
        return ++x + y++;
    }
    x = 0;
    y = b;
    (p++) = 1;
    sha1(msg, 5 + len, p);
    send(msg, msg_len);
    
    out(len) payload(01 | payload (01 | payload, k_{shared}))
```

Using symbolic execution we can summarise the effect of the function:

```
Symbolic Execution
```

```
out(len) payload(01 | payload (01 | payload, k_{shared}))
```

Here “|” stands for bitstring concatenation. High-level verification tools can’t deal with it directly, so we perform some extra analysis to prove that concatenation patterns can be abstracted as pure symbolic functions:

```
Format Abstraction
```

```
out(f(payload, hash(payload, k_{shared})))
```

This is a representation that ProVerif can easily deal with:

```
ProVerif (+ Server Side)
```

Integrity verified.

8. Current Status

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.

9. Project

The project is supervised by Andrew Gordon (Microsoft Research), Jan Jürjens (Dortmund University), and Bashar Nuseibeh (Open University). It is partially supported through the Microsoft Research PhD Scholarship programme.