Verifying Implementations of Security Protocols in C

Mihhail Aizatulin$^1$, François Dupressoir$^1$
Andrew Gordon$^2$, Jan Jürjens$^{1,2}$, Bashar Nuseibeh$^1$

$^1$The Open University
$^2$Microsoft Research Cambridge

CRC PhD Student Conference
Open University
June 18-19, 2009
Our goal is a tool for security analysis at implementation level.

Source code of a cryptographic protocol implementation

Our tools

List of all potential security flaws or proof of security
The Goal

Source code of a cryptographic protocol implementation

Our tools

List of all potential security flaws or proof of security

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

- automated,
Our goal is a tool for security analysis at implementation level. The analysis should be

- automated,
- sound (not miss any bugs),

Source code of a cryptographic protocol implementation

Our tools

List of all potential security flaws or proof of security
The Goal

Source code of a cryptographic protocol implementation

Our tools

List of all potential security flaws or proof of security

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).
The Goal

Source code of a cryptographic protocol implementation

Our tools

List of all potential security flaws or proof of security

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).

The security flaws we are looking for are violations of **secrecy** and **authentication**.
A protocol fulfills secrecy if the attacker has no way to get information about exchanged secrets.

**Figure:** Secrecy is preserved.
A protocol fulfills *authentication* if the participants have consistent views of events.

**Figure:** An example of violated authentication.
Motivation

Protocols are designed on abstract level.

\[ A \xrightarrow{m, \text{hash}(k_{AB}, m)} B \]
Protocols are designed on abstract level. But ultimately it is the code that defines them. Do you think this code is secure?

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

\[
\ldots
\text{msg\_len} = \text{recv}(\text{msg}, \text{MAXLEN}) ;
\text{mac\_len} = \text{recv}(\text{mac}, \text{MAXLEN}) ;
\]

\[ \text{hash}(\text{key}, \text{msg}, \text{msg\_len}, \text{my\_mac}) ; \]

\[ \text{if}(!\text{strcmp}(\text{mac}, \text{my\_mac})) \]
\[ \text{assert}(\text{valid}(\text{msg}, \text{msg\_len})) ; \]

\[ \ldots \]
## 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
- ($\pi$-calculus, LySa)

<table>
<thead>
<tr>
<th>Low-level properties</th>
<th>Machine Languages (C, Java)</th>
<th>Formal Languages ($\pi$-calculus, LySa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(NULL dereference, division by zero)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High-level properties</th>
<th>Machine Languages (C, Java)</th>
<th>Formal Languages ($\pi$-calculus, LySa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(secrecy, authentication)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Background

There has been great progress in

- static software analysis,

<table>
<thead>
<tr>
<th>Machine Languages (C, Java)</th>
<th>Formal Languages ($\pi$-calculus, LySa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>low-level properties</td>
<td>low-level properties</td>
</tr>
<tr>
<td>(NULL dereference, division by zero)</td>
<td>(NULL dereference, division by zero)</td>
</tr>
<tr>
<td>high-level properties</td>
<td>high-level properties</td>
</tr>
<tr>
<td>(secrecy, authentication)</td>
<td>(secrecy, authentication)</td>
</tr>
</tbody>
</table>

- VCC
- Frama-C
- ESC/Java
- SLAM

Our Goal

ProVerif/CryptoVerif
AVISPA
LySatool

M. Aizatulin, F. Dupressoir
Verifying Implementations of Security Protocols in C
Background

There has been great progress in

- static software analysis,
- verification of protocol specifications.

<table>
<thead>
<tr>
<th>Machine Languages (C, Java)</th>
<th>Formal Languages ($\pi$-calculus, LySa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>low-level properties (NULL dereference, division by zero)</td>
<td>VCC</td>
</tr>
<tr>
<td>high-level properties (secrecy, authentication)</td>
<td>ESC/Java</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M. Aizatulin, F. Dupressoir

Verifying Implementations of Security Protocols in C
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>Machine Languages (C, Java)</th>
<th>Formal Languages (π-calculus, LySa)</th>
</tr>
</thead>
<tbody>
<tr>
<td>low-level properties (NULL dereference, division by zero)</td>
<td>VCC</td>
</tr>
<tr>
<td></td>
<td>ESC/Java</td>
</tr>
<tr>
<td>high-level properties (secrecy, authentication)</td>
<td>CSur</td>
</tr>
<tr>
<td></td>
<td>Our Goal</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What is so high-level about security properties? Why can’t we simply use general-purpose verification tools?
What is so high-level about security properties? Why can’t we simply use general-purpose verification tools?

General-purpose tools speak about the local state of the program, but security properties need to refer to

- the state of an unknown attacker,
- the states of the other protocol participants.
Proceed in two steps: first extract a clean model from the code, then use existing high-level techniques to verify it.
Approach

Proceed in two steps: first extract a clean model from the code, then use existing high-level techniques to verify it.

To extract the model use symbolic execution: Keep track of mappings $(\text{ptr}, \text{length}) \mapsto x$ from memory locations to corresponding terms.
Proceed in two steps: first extract a clean model from the code, then use existing high-level techniques to verify it. To extract the model use symbolic execution: Keep track of mappings \((\text{ptr, length}) \mapsto x\) from memory locations to corresponding terms. Looking at our motivational example:

\[
\text{msg\_len} = \text{recv}(\text{msg}, \text{MAXLEN}); \quad \text{x} = \text{recv}() \\
(\text{msg}, \text{msg\_len}) \mapsto x \\
\text{hash}(\text{key}, \text{msg}, \text{msg\_len}, \text{my\_mac}) \\
\text{if} \quad (\text{!strcmp}\ (\text{mac}, \text{my\_mac})) \quad \text{assert}(\text{valid}(\text{msg}, \text{msg\_len})) \\
\text{x} = \text{recv}() \\
(\text{msg}, \text{msg\_len}) \mapsto x
\]
Proceed in two steps: first extract a clean model from the code, then use existing high-level techniques to verify it. To extract the model use symbolic execution: Keep track of mappings \((\text{ptr}, \text{length}) \mapsto x\) from memory locations to corresponding terms. Looking at our motivational example:

\[
\begin{align*}
\text{msg}_\text{len} &= \text{recv}(\text{msg}, \text{MAXLEN}); \\
\text{mac}_\text{len} &= \text{recv}(\text{mac}, \text{MAXLEN}); \\
\end{align*}
\]

\[
\begin{align*}
x &= \text{recv}() \\
(\text{msg, msg}_\text{len}) &\mapsto x \\
y &= \text{recv}() \\
(\text{mac, mac}_\text{len}) &\mapsto y
\end{align*}
\]
Approach

Proceed in two steps: first extract a clean model from the code, then use existing high-level techniques to verify it.

To extract the model use symbolic execution: Keep track of mappings \((\text{ptr}, \text{length}) \mapsto x\) from memory locations to corresponding terms. Looking at our motivational example:

\[
\begin{align*}
\text{msg\_len} &= \text{recv}(\text{msg, MAXLEN}); & x &= \text{recv}() \\
\text{mac\_len} &= \text{recv}(\text{mac, MAXLEN}); & (\text{msg, msg\_len}) &\mapsto x \\
\text{hash}(\text{key, msg, msg\_len, my\_mac}); & y &= \text{recv}() \\
& (\text{mac, mac\_len}) &\mapsto y \\
& (\text{my\_mac, mac\_len}) &\mapsto \text{hash}(k_{AB}, x)
\end{align*}
\]
Proceed in two steps: first extract a clean model from the code, then use existing high-level techniques to verify it. To extract the model use symbolic execution: Keep track of mappings $(\text{ptr}, \text{length}) \mapsto x$ from memory locations to corresponding terms. Looking at our motivational example:

\begin{align*}
\text{msg\_len} &= \text{recv}(\text{msg}, \text{MAXLEN}); & x &= \text{recv}() \\
\text{mac\_len} &= \text{recv}(\text{mac}, \text{MAXLEN}); & (\text{msg}, \text{msg\_len}) &\mapsto x \\
\text{hash}(\text{key}, \text{msg}, \text{msg\_len}, \text{my\_mac}); & y &= \text{recv}() \\
& & (\text{mac}, \text{mac\_len}) &\mapsto y \\
\text{if}(!\text{strcmp}(\text{mac}, \text{my\_mac})) & \text{assume}(\text{string}(y) == \text{string}(\text{hash}(k_{AB}, x))) \\
\text{assert}(\text{valid}(\text{msg}, \text{msg\_len})); & \text{assert}(\text{valid}(x))
\end{align*}