Applying Runtime Verification to Group Key Establishment

Secure Communication in the Quantum Era (SPS G5448)

Slovakia - January 2019

Christian Colombo
The Project

Collaboration between Slovakia, Malta, US, and Spain:

Secure Communication in the Quantum Era
The Project

Collaboration between Slovakia, Malta, US, and Spain:

Secure Communication in the Quantum Era

Attacker with Quantum resources

Secure communication
Authenticated group key establishment (AGKE)

Secure communication depends on establishing **common secret key**

Project will focus on securing AGKE
Authenticated group key establishment (AGKE)

First step: Designing a protocol

a) $A$ sends $B$ the message $(A, E_B(MA), B)$,
b) $B$ answers $A$ by sending $(B, E_A(MB), A)$. 
Authenticated group key establishment (AGKE)

First step: Designing a protocol

Second step: Proving it is correct in principle

a) $A$ sends $B$ the message $(A, E_B(MA), B)$,
b) $B$ answers $A$ by sending $(B, E_A(MB), A)$.
Authenticated group key establishment (AGKE)

First step: Designing a protocol

Second step: Proving it is correct in principle

Third step: What can go wrong at runtime?

a) $A$ sends $B$ the message $(A, E_B(MA), B)$,
b) $B$ answers $A$ by sending $(B, E_A(MB), A)$. 
What can go wrong at runtime?

(High level) Wrong protocol implementation
The protocol implementation might deviate from the verified (theoretical) design

Low level threats
Arithmetic overflows, undefined downcasts, and invalid pointer references

Hardware
Can hardware be trusted?
Passive / Active attacks

Something bad accidentally happens

Vs

Something bad actively sought
What can go wrong at runtime?

...but in practice is far from enough

(High level) Wrong protocol implementation

The protocol implementation might deviate from the verified (theoretical) design

Low level threats

Medium level threats: Malware, Data leaks, etc

and invalid pointer references

Hardware

Can hardware be trusted?
Unintended consequences

- Timing attacks
- Cache timing attacks
- Microarchitecture side-channel attack
- Power/EM/acoustic attacks
- Fault attacks
- Reaction attacks
- Data remanence attacks
- Attacks on random number generators
Timing attack

If (secret)
   Do something lengthy
Else
   Do something simple

An external observer can learn the secret by observing the duration of the execution.
(or the power used or any other side effect)
What can we do?

Static + Dynamic analysis of the code to make sure secrets can’t be leaked
Dynamic analysis

\[
\begin{align*}
&\text{if } h = 1 \text{ then } b := 1 \text{ else skip;} \\
&\text{if } b \neq 1 \text{ then } l := 1 \text{ else skip;} \\
&\text{output}_L(l)
\end{align*}
\]

Assume $h$ is a high security variable. When $h \neq 1$, Monitor can’t mark $b$ as high (without analysing the “if” statically).
Soundness/Completeness of dynamic analysis

(a) Flow-insensitive analysis
(b) Flow-sensitive analysis
(c) Flow-sensitive analysis, hybrid monitors
Identifying secure programs

Secure programs

Programs accepted by analysis
Soundness/Completeness of dynamic analysis

- Secure programs
  - Statically Analysed
  - Dynamically Analysed

- Secure programs
  - Dynamically + Statically Analysed
  - Statically Analysed
How do we use these techniques in practice?
Runtime Verification

What specification language to use

What tool to use

What protocol to adopt between the system and monitor/verifier
What has been done?
High level logic

- Before any data is sent by the client, the server hash is verified to match the client's version.
- If the operation is of type "Send", then the message receiver ID must be in the set of approved receiver IDs.
Low level considerations

General considerations for any code

- Arithmetic overflows
- Undefined downcasts
- Invalid pointer references
Mid-level

Applicable to any crypto protocol

Data flow monitoring

E.g. Ensuring no control is decided on secret data

(which affects the timing of the program)
Frama-C

Is a framework supporting all of these levels
(*combining static and dynamic checking*)

- Low-level is inbuilt through standard checks
- Mid-level is provided through library support
- High-level is provided through specification languages
Other tools/frameworks?

Copilot and other tools focus more on high level properties

Work on hyperproperties

I.e. properties on several runs of a program
What are the challenges?
Challenges for RV

Over and above the usual correctness and **overheads** concerns

The monitor can present an additional security vulnerability

>> As a piece of code

>> As a reaction-triggering device
Other techniques?

For example Multi-execution approach:

Generate low security outputs with only low security inputs in the system

→ **Result:** No high security output may depend on low security input
Our plan of comprehensive approach: Trusted Execution Environment (TEE)
Questions to be answered

To what extent existing tools/frameworks are immune to attacks themselves

Is there any effect of the quantum prospect on RV?

What mix of new/existing techniques + technologies to adopt
Non-trusted OS and apps

Full isolation

TEE  Trusted OS/app/code fragment

Isolated allocation of

Requires a complete context switch (CPU, memory, bus) and encryption (storage) in case shared with non-trusted code.

Non-trusted OS and apps

User-mode

Kernel-mode

Call or provision code

Secure monitor

Trust boundary crossing: thorough checking of data flows

Secure boot

TPM

Boot loader + Keys
Trusted isolated hardware - avoid hardware threats

Non-trusted OS and apps
- User-mode
- Kernel-mode

TEE
Trusted OS/app/code fragment
- Isolated allocation of
- Requires a complete context switch (cpu, memory, bus) and encryption (storage) in case shared with non-trusted code.

Secure monitor
- Trust boundary crossing: thorough checking of data flows

Full isolation

Call or provision code

Secure boot

TPM
- Boot loader + Keys
Highly restricted and monitored communication to the outside world - Avoiding leaks, malware, etc.
Applying Trusted Execution Environment to AGKE
All crypto related actions happen here
Keys never leave flash memory

Commodity h/w and stock OS
- User-mode
- Kernel-mode
- AGKE code
- Token calls
- Kernel trap
- USB/network drivers

Crypto h/w token
- Crypto OS
  - Generate/Retrieve
  - APDU
  - MCU
  - Keys
  - Ciphers

AGKE network exchanges
- Crypto OS calls
- Plain/ciphertext data exchanges
- NO key transfers
- NO external code provisioning
Possible comprehensive approach

- Monitor high level protocol implementation
- Monitor interaction across isolation boundary

Diagram:
- Commodity h/w and stock OS
  - User-mode
    - AGKE code
    - Token calls
  - Kernel-mode
    - Kernel trap
    - USB/network drivers
- Crypto h/w token
  - Generate/Retrieve
  - APDU
  - Keys
  - Ciphers

- AGKE network exchanges
- Crypto OS calls
- Plain/ciphertext data exchanges
- NO key transfers
- NO external code provisioning
What can go wrong at runtime?

...but in practice is far from enough

(High level) Wrong protocol implementation

Monitor 1

Monitor 2

Low level threats

By trusted execution env.

Hardware

The protocol implementation might deviate from the verified (theoretical) design

Medium level threats: Data leaks, malware, etc

and invalid pointer references

Can hardware be trusted?
Overheads concerns

Instrument at binary level

   Enables full optimisation

Taint tracking is expensive

   Taint inference as an optimisation (Trades precision with efficiency)
Future Work

Frama-C specification language

Frama-C with modifications

Binary level instrumentation