Secure Implementation of a Quantum-Future GAKE Protocol*

R. Abela, C. Colombo, P. Malo, P. Sýs, T. Fabšič, O. Gallo, V. Hromada, and M. Vella

* This work is supported by the NATO Science for Peace and Security Programme through project G5448 Secure Communication in the Quantum Era.

Abstract

Incorrect cryptographic protocol implementation and malware attacks targeting its runtime may lead to insecure execution **even if the protocol design has been proven safe.**

This research focuses on adapting a Runtime-Verification-centric Trusted Execution environment (RV-TEE) solution to a quantum-future cryptographic protocol deployment.

We aim to show that our approach is practical through an instantiation of a trusted execution environment supported by runtime verification and any hardware security module compatible with commodity hardware.

GAKE: Group Authentication Key Exchange protocols are essential for constructing secure channels of communication between multiple parties over an insecure infrastructure.

Quantum-future: Systems that result in the protection of message confidentiality from delayed data attacks using eventual quantum power, but not protecting from active quantum adversaries.

This research provides

1. A group chat app case study which uses the quantum-future GAKE protocol from Gonzalez Vasco et al.
2. An implementation of the GAKE protocol employing SEcube™, a resource-constrained hardware security module.
3. The RV setup tailored for the protocol's properties using the automata-based LARVA RV tool.
4. An empirical evaluation of the setup focusing on the user experience of the chat app demonstrating the practicality of the RV-TEE setup.

The need for secure collaboration

As the COVID-19 pandemic lockdown forced most employees into remote working, serious weaknesses in Zoom were exposed. Issues ranged from insecure key establishment to inadequate block cipher mode usage.

Other previous high-profile incidents concerning insecure cryptographic protocol implementation were caused by:

- Weak randomness.
- Insufficient checks on protocol compliance.
- Memory corruption bugs.

RV-TEE

Secure implementation via a TEE is specifically provided through an instantiation of RV-TEE, which combines the use of two components:

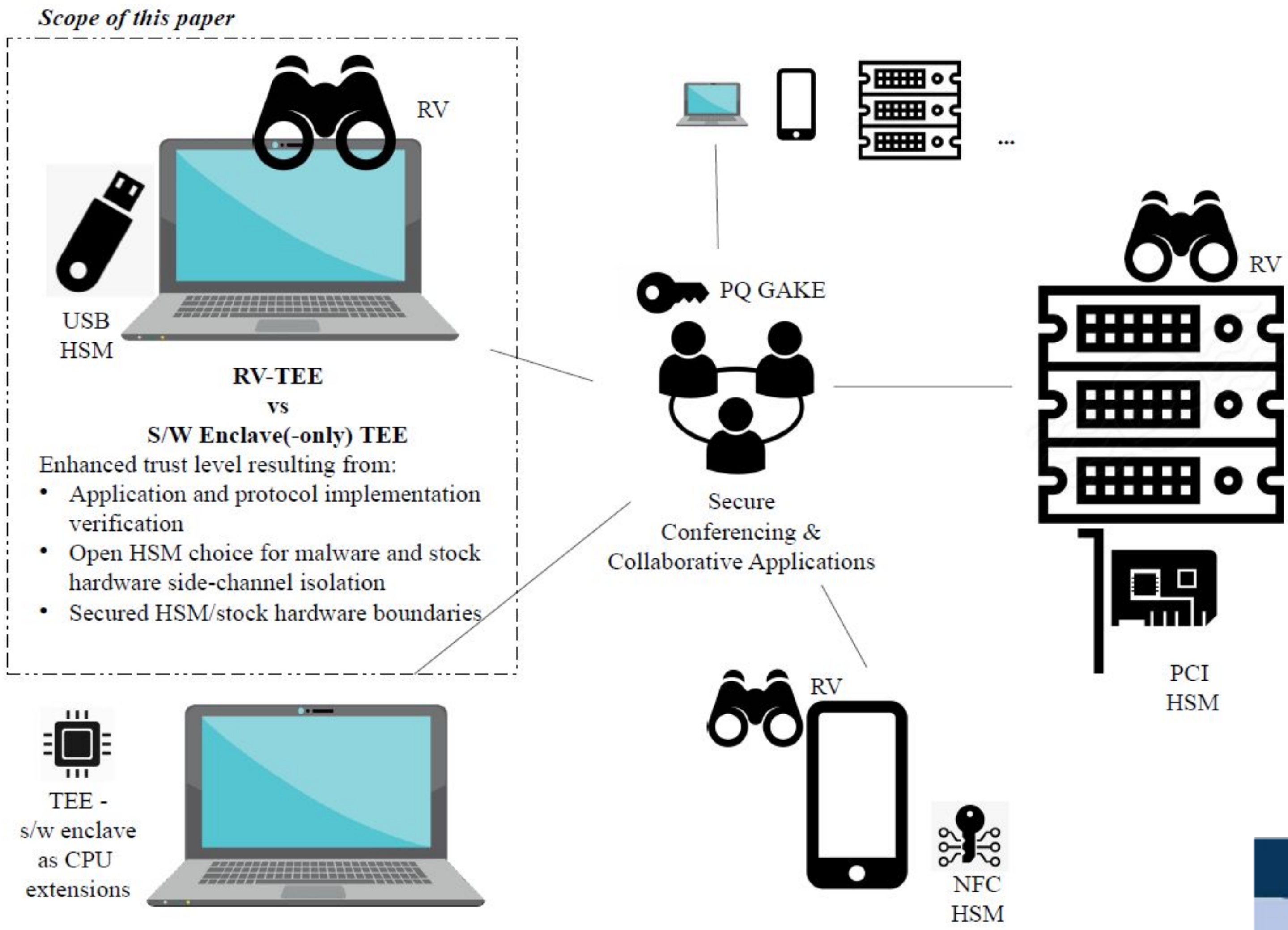
1. Runtime Verification (RV), a dynamic formal verification extension to static model checking.
2. A Hardware Security Module (HSM) of choice to provide an isolated execution environment, possibly equipped with tamper-evident features.

Separate roles

The remit of RV is primarily the verification of correct protocol usage by conferencing/collaborative applications, as well as the protocol implementation itself.

The HSM protects the execution of code associated with secret/private keys from malware infection while avoiding stock hardware side-channels.

Furthermore, RV is also tasked with monitoring data flows between the HSM and stock hardware.


Scope of this paper
RV
USB HSM
PQ GAKE
Secure Conferencing & Collaborative Applications
...
RV
PCI HSM
RV
NFC HSM
TEE - s/w enclave as CPU extensions

RV-TEE
vs
S/W Enclave(-only) TEE

Enhanced trust level resulting from:

- Application and protocol implementation verification
- Open HSM choice for malware and stock hardware side-channel isolation
- Secured HSM/stock hardware boundaries

Elevated level of trust through the RV-TEE Setup

Most stock hardware nowadays comes equipped with CPUs having TEE extensions based on encrypted memory to provide software enclaves, and which could also be a suitable choice for the HSM if deemed fit.

The entire RV-TEE setup provides better trust through:

- Application and protocol implementation verification using RV.
- Use the trusted HSM of choice to isolate from malware and stock hardware side-channels.
- RV securing the HSM/stock hardware boundaries.

Levels of concern: Software

- **Highest**: exploiting logical bugs causing the protocol implementation to deviate from the design.
- **Medium**: attacks targeting the secrecy of symmetric/private keys, along with the unavailability of plaintext without first breaking encryption.
- **Lowest**: Vulnerabilities originating from programming bugs, resulting in the deductibility of secrets.

Levels of concern: Hardware

Beneath the software threat levels, are those at the hardware level. These can pose a threat if the manufacturer cannot be trusted.

This can be particularly of concern if the hardware itself is a primitive for secure execution, is widely deployed and an application's implementation is specific to it.

In this respect, RV-TEE is designed with HSM flexibility in mind.

Runtime Verification

Provides two primary benefits:

1. Monitors are typically automatically synthesised from formal notation to reduce the possibility of introducing bugs.
2. Monitoring concerns are kept separate (at least on a logical level) from the observed system.

In our case study we use LARVA: properties are specified using LARVA scripts that capture a textual representation of symbolic timed-automata.

Industry-grade critical enablers

LARVA RV has been used extensively for verifying the correct operation of high-volume financial transactions systems.

SEcube™ HSM is integrated in various devices, several open-source projects build upon its Open SDK to demonstrate its employment in academic research.

Cortex-M4 MCU has an extensive body of work focusing on optimised cipher implementation that cover both post-quantum cryptography, as well as standard symmetric encryption.

Runtime verification properties

Property layers	**Chat app**	**Library**	**All (incl. Primitives)**
Assertion	Printable decrypted characters	Sensitive data scrubbed	Valid function parameters and returns
Temporal	Chatroom lifecycle, standard sockets		Correct function call sequence
Hyper		Randomness quality	

Properties

Function input/output: valid inputs and output (with respect to the inputs).

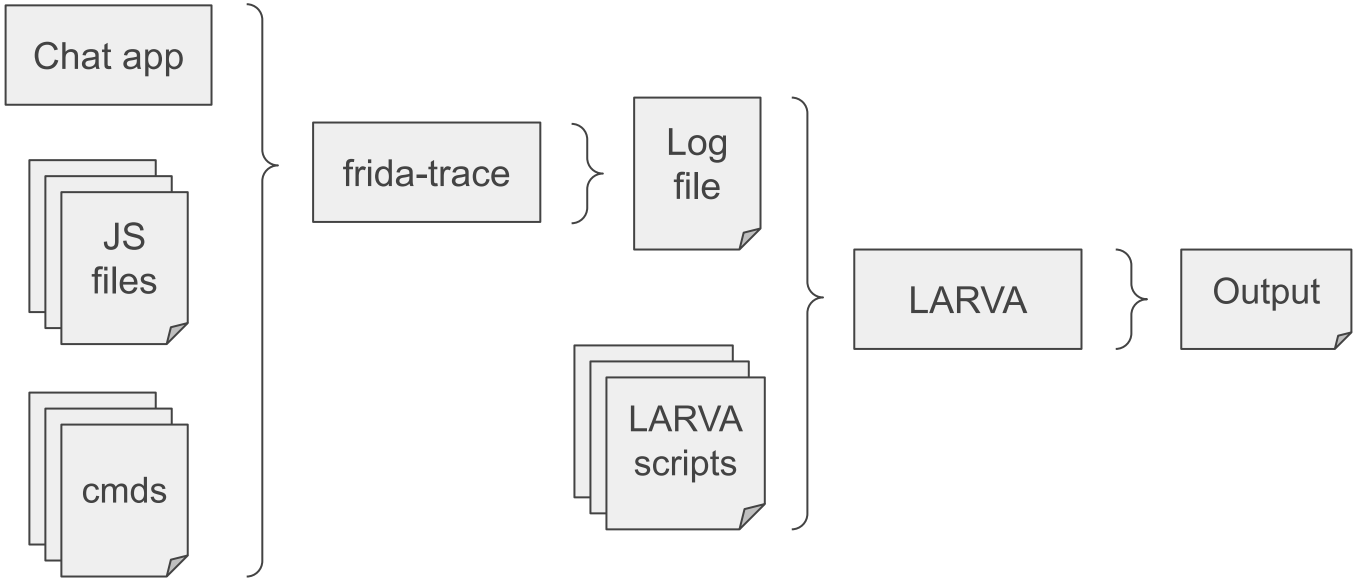
Data scrubbing: sensitive data is properly destroyed after use, once secure communication is established.

Sequences of actions: permitted by protocol participants depending on the context. In our case study, the protocol follows a high level sequence of rounds.

Randomness: high quality random number generation.

Application-specific: dealing with the chatroom lifecycle.

Instrumentation overview



```
/**
 * Called synchronously when about to call get_participant_by_id.
 * struct _participant* get_participant_by_id(
 *     const struct _protocol_run* p_run, const uint8_t* id);
 */
onEnter(log, args, state) {
  this.prun = args[0];
  pest_log(log,
    `get_participant_by_id(p_run=${this.prun}, id=${args[1].readU8().toString()})`
  );
}

/**
 * Called synchronously when about to return from get_participant_by_id.
 */
onLeave(log, retval, state) {
  pest_log(log, `get_participant_by_id() retVal: ${retval}`);
  if (this.prun != get_prun_by_participant(log, retval))
    pest_warn(log, `unknown participant ${retval} for protocol run ${this.prun}`);
}
```

```
STATES {
  BAD {
      bad{System.out.println("Wrong");}
  }
  NORMAL {
      round1{System.out.println("Started round 1");}
      ...
  }
  STARTING {
      round1{System.out.println("Started round 1");}
  }
}
TRANSITIONS {
  ...
  init_protocol_run_env -> bad[init_protocol_run_env]
  init_protocol_run_env -> init_participant[init_participant]
  init_protocol_run_env -> bad[round_one]
  init_protocol_run_env -> bad[load_pw]
  ...
}
```

Scenarios

Scenario A: 3 clients involved, with the monitored client creating a room following the protocol steps for an **initiator participant** U_0

Scenario B: 3 clients involved, with the monitored client joining the room following the protocol steps for a **non-initiator participant** $U_{1 \leq i \leq n}$

The scenarios include 20 and 13 seconds of thread sleeps respectively to mimic a realistic chat. This will be factored in in the results discussion.

Scenario B: Commands for monitored client

```
/sleep 6
/room enter ROOMNAME
/sleep 1
secure msg from one
/sleep 3
goodbye (1)
/sleep 3
/exit
```

Instrumentation overheads

Time (s)	Without SEcube™			Using SEcube™		
Scenario	A	B	All	A	B	All
Non-instrumented	20.02	13.01	33.03	20.18	13.27	33.45
Instrumented	20.44	14.39	34.83	21.30	13.68	34.98
Increase	0.44	1.38	1.70	1.12	0.41	1.53

Runtime verification empirical results

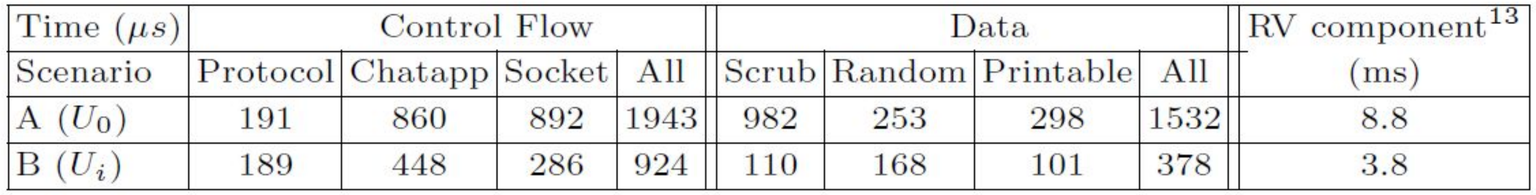
RV checked 6 properties: 3 classified as control flow, and 3 as data properties.

The **control flow properties** checked the sequence of actions for the protocol and chat app execution, while the third property kept track of sockets being written to, reporting any suspicious ones.

The **data flow properties** involved checking data is scrubbed, basic assessment of the quality of the generated random numbers, and checking that all input characters are printable.

Discussion

1. The empirical results indicate that the overheads introduced by the instrumentation and the HSM are non-negligible.
2. However, in this work, our main aim was to show the feasibility of the approach rather than to have an optimal solution.



Time (μs)	Control Flow				Data				RV component[13]
Scenario	Protocol	Chatapp	Socket	All	Scrub	Random	Printable	All	(ms)
A (U_0)	191	860	892	1943	982	253	298	1532	8.8
B (U_i)	189	448	286	924	110	168	101	378	3.8

Efficiency improvements

- We use frida-trace at the level of JS not only for setting up in-line hooks dynamically, but also for the instrumentation code itself that records the events of interest. The use of natively compiled code would make this faster (but require more extensive testing).
- The instrumentation gathered all events which could be useful for RV. This provided us with experimental flexibility at the expense of higher overheads.
- We foresee substantial immediate gains if we minimise the use of JS and limit the events to those strictly needed.

Limitations

- Present implementation does not contain protection against side-channel attacks beyond the immediate protection derived from stock hardware isolation. We plan to address the issue in future work.
- We performed all verification asynchronously. Since the time required for the actual verification is small, this could be done online and in sync with the chat app execution. If heavier RV is needed (e.g., more thorough randomness checking), properties could be split into two categories:
  - state checks (monitored synchronously)
  - data checks (monitored asynchronously).

Conclusion

Insecure execution of theoretically-proven communication protocols is still a major concern, due to vulnerabilities at the various levels of the implementation. With the advancements of quantum computers, novel quantum-safe protocols are inevitable.

We proposed an RV-TEE instantiation for the quantum-future GAKE protocol from Gonzalez Vasco et al., securing the protocol implementation from the hardware level, up till the logical level of the application utilising it.

Through an empirical evaluation based on a chat application case study, we show the feasibility of the approach involving substantial overhead, yet with minimal to no impact from a usability perspective.

Thank you

robert.abela@um.edu.mt