Secure Implementation of a Quantum-Future GAKE Protocol

Robert Abela[1], Christian Colombo[1(✉)], Peter Malo[2], Peter Sýs[3], Tomáš Fabšič[2], Ondrej Gallo[2], Viliam Hromada[2], and Mark Vella[1]

[1] Department of Computer Science, University of Malta, Msida, Malta
christian.colombo@um.edu.mt
[2] Faculty of Electrical Engineering and Information Technology, Slovak University of Technology in Bratislava, Bratislava, Slovakia
[3] Mathematical Institute, Slovak Academy of Sciences, Bratislava, Slovakia

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. In particular, we provide: (i) A group chat application case study which uses the quantum-future group key establishment protocol from González Vasco et al., (ii) An implementation of the protocol from González Vasco et al. employing a resource-constrained hardware security module, (iii) The runtime verification setup tailored for the protocol's properties, (iv) An empirical evaluation of the setup focusing on the user experience of the chat application.

Keywords: Runtime verification · Post-quantum cryptography · Trustworthy systems

1 Introduction

Group authentication key exchange (GAKE) protocols are essential for constructing secure channels of communication between multiple parties over an insecure infrastructure [25]. Collaborative applications over the Internet, covering all different kinds of video and web conferencing software, are a prime example of applications that can benefit from GAKE, allowing symmetric-key cryptography to be used for both authentication and encryption whenever it is not possible to agree on shared secrets over the same medium of communication that requires securing.

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

© Springer Nature Switzerland AG 2021
R. Roman and J. Zhou (Eds.): STM 2021, LNCS 13075, pp. 103–121, 2021.
https://doi.org/10.1007/978-3-030-91859-0_6

The need for secure collaborative web applications has been emphasised with the onset of the COVID-19 pandemic. Literally, only a few days into the ensuing lockdown that forced most employees around the world into remote working, serious weaknesses in one of the most popular web conferencing applications were immediately exposed [30]. Issues ranged from insecure key establishment to inadequate block cipher mode usage. Yet this is only the latest in a string of high-profile incidents concerning insecure cryptographic protocol implementation. Root causes span weak randomness [42], insufficient checks on protocol compliance in remote party exchanges [23], as well as memory corruption bugs in code [35]. With malware code injection techniques becoming ever more sophisticated in bypassing security controls [38], possibly even leveraging micro-architectural side-channels of commodity hardware [7,26], the secure implementation of otherwise securely proven cryptography remains a challenge.

Quantum adversaries present one further challenge for GAKE and its applications. While it is difficult to gauge the level of threat concerned, NIST's announcement of the third round finalists of the post-quantum cryptography standardisation process has set the tone for the level of preparedness expected of the level of security for sensitive scenarios. In terms of implementation, the added burden is presented by the even larger operands involved in the lattice and code-based schemes [4], for example, as compared to those based on discrete-logarithm and factoring assumptions.

In this paper, we address the problem of securely implementing a quantum-future GAKE protocol through a Trusted Execution Environment (TEE) [36], in the setting of a secure group chat application. Specifically, we focus on a proposed protocol proven to be secure in a *quantum-future* scenario [22]. In doing so, its design aims to balance post-quantum security and implementation efficiency. A quantum-safe key encapsulation mechanism (KEM), e.g., CRYSTALS-Kyber [12], is only used for protecting the confidentiality of session key material. On the other hand, authentication is still based on cheaper discrete-log primitives, with the end result being the protection of message confidentiality from delayed data attacks using eventual quantum power, but without the 'unnecessary burden' of protecting from active quantum adversaries. Finally, user authentication is password-based.

Secure implementation via a TEE is specifically provided through an instantiation of *RV-TEE* [41]. RV-TEE combines the use of two components: The first is Runtime Verification (RV), a dynamic formal verification extension to static model checking [14,28]. The second component is a Hardware Security Module (HSM) of choice to provide an isolated execution environment, possibly equipped with tamper-evident features. Options encompass high-bandwidth network PCI cards with hardware-accelerated encryption [39], down to smaller on-board micro-controllers and/or smartcards used in resource-constrained devices that connect to stock hardware over USB or NFC for example [11,20]. 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

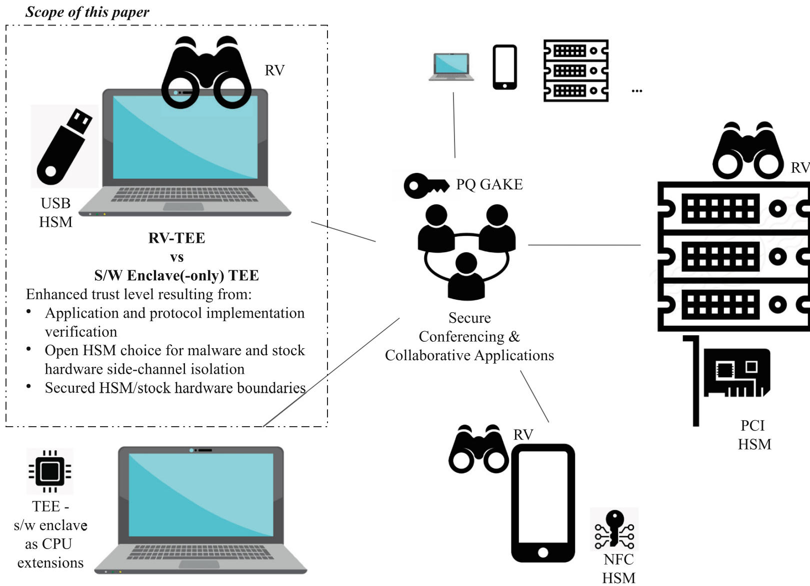

Fig. 1. The RV-TEE setup used for securing conferencing and collaborative applications based on a quantum-future GAKE.

infection while avoiding stock hardware side-channels. Furthermore, RV is also tasked with monitoring data flows between the HSM and stock hardware.

Figure 1 shows the overall setup, delineating the scope of the case study presented in this paper. Most stock hardware nowadays comes equipped with CPUs having TEE extensions based on encrypted memory to provide software enclaves [31], and which could also be a suitable choice for the HSM if deemed fit. However, by adopting the entire RV-TEE setup—i.e., including the RV component—an elevated level of trust can be achieved through:

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

Despite providing direction for securely implementing conferencing/collaborative applications, RV-TEE is not prescriptive in the sense that, the RV properties for verification must be chosen carefully in terms of partial specification of the GAKE protocol, while protocol implementation must be split between the stock hardware and the HSM—with the exact delineation depending on the specific choice of the latter. This paper is about the work undertaken to bridge this gap, making the following contributions: (i) A group chat application case study based on a post-quantum protocol that is secure in a quantum-future scenario; ii) A protocol implementation using the SEcube™ [40], an inexpensive

yet sufficiently powerful HSM to cater for the requirements of a lattice-based KEM used by the post-quantum protocol implementation; (iii) The runtime verification setup tailored for the protocol's properties using the automata-based Larva [16] RV tool; (iv) An empirical evaluation of the setup focusing on the user experience of the chat application demonstrating the practicality of the RV-TEE setup for securing conferencing/collaborative applications based on a post-quantum GAKE.

2 Background

2.1 Quantum-Future GAKE Protocol

The cryptographic protocol chosen for our case study is the quantum-future GAKE protocol by González Vasco et al. [22]. The protocol allows members of a group to perform an authenticated key exchange. Below, we present a detailed description of the protocol.

Quantum-Future Security. The protocol is provably secure in a *quantum-future* scenario wherein it is assumed that no quantum-adversary is present during the execution of the protocol, and the established common secret is supposed to remain secure even if the adversary gains access to a quantum computer in the future.

Password-Based Authentication. The protocol uses password-based authentication. It employs a prime order group G in which the Decision-Diffie-Hellman assumption holds and assumes that the password dictionary $\mathcal{D}$ is a subset of G through some public and efficiently computable injection $\iota : \mathcal{D} \hookrightarrow G$.

Tools. The protocol uses the following tools:

- a *key encapsulation mechanism* (KEM)
- a *message authentication code* (MAC)
- a *deterministic randomness extractor*

Next, we describe these tools in more detail.

KEM - the protocol requires a KEM which is IND-CPA secure against fully quantum adversaries (as defined in [9]). A KEM $\mathcal{K}$ consists of the following three algorithms:

- a probabilistic *key generation* algorithm $\mathcal{K}.\texttt{KeyGen}(1^l)$ which takes as input the security parameter l and outputs a key pair (pk, sk),
- a probabilistic *encapsulation* algorithm $\texttt{Encaps}(pk, 1^l)$ which takes as input a public key pk and outputs a ciphertext c and a key $k \in \{0,1\}^{p(l)}$, where $p(l)$ is a polynomial function of the security parameter,
- a deterministic *decapsulation* algorithm $\texttt{Decaps}(sk, c, 1^l)$ which takes as input a secret key sk and a ciphertext c and outputs a key k or $\perp$.

MAC - the protocol requires a MAC which is *unforgeable under chosen message and chosen verification queries attack* (UF-CMVA) (as defined in [21]). A MAC $\mathcal{M}$ consists of the following three algorithms:

- a probabilistic *key generation* algorithm $\mathcal{M}.\texttt{KeyGen}(1^l)$ which takes as input the security parameter l and outputs a key k,
- a probabilistic *authentication* algorithm $\texttt{Tag}(k, M)$ which takes as input a key k and a message M and outputs a tag t,
- a deterministic *verification* algorithm $\texttt{Vf}(k, M, t)$ which takes as input a key k, a message M and a tag t and outputs a decision: 1 (accept) or 0 (reject).

Deterministic Randomness Extractor - in the protocol, (uniform random) bit-strings need to be extracted from (uniform random) elements in the group G. Given a group element $g \in G$, the authors of the protocol denote by $[g]$ (statistically close to uniform random) bits extracted deterministically from g. When the authors of the protocol need to extract two independent (half-length) bit-strings from g, they divide $[g]$ into two halves denoted by $[g]_L$ and $[g]_R$ such that $[g] = [g]_L || [g]_R$.

Protocol Specification. Let $U_0, U_1, \ldots, U_n$ be the users running the protocol. It is assumed that before the protocol is started, the users share a password pw. In addition, it is assumed that every user is aware of his index and the indices of the rest of participants.

The protocol is depicted in Fig. 2. The user U_0 has a special role in the protocol - he generates a key k and transports it to the other users. The key is transported being masked by an ephemeral key generated from the key encapsulation. To ensure authentication, each pair of users establishes a Diffie-Hellman secret, with $\iota(\texttt{pw})$ used as a generator of the group G. The resulting Diffie-Hellman secrets are then used to derive keys for authentication tags on protocol messages. Once a user verifies all tags, the key k is accepted and is used to derive a shared session key (ssk) and a session ID (sid).

2.2 RV-TEE

RV-TEE [41] was proposed as a secure execution environment in the context of a threat model comprising adversaries that target cryptographic protocol execution at four different levels. The first three levels concern software and are labelled as high, medium, and low. At the highest level, one finds attacks exploiting logical bugs causing the protocol implementation to deviate from the (typically theoretically-verified) design. The exploitation of incorrect verification of digital certificates or authentication tags, usage of insecure groups to implement Diffie Hellman-based protocols, as well as weak sources of randomness, all fall in this category [2]. At the medium level, we find attacks that target the basic assumption of any cryptographic scheme: the secrecy of symmetric/private keys, along with the unavailability of plaintext without first breaking encryption.

Round I.

- `Computation:`
  - For $0 \leq i \leq n, U_i$ chooses $\beta_i \leftarrow \mathbb{Z}_q$ and computes $g_i = \iota(\mathtt{pw})^{\beta_i}$. U_0 sets $M_0 := (g_0)$.
  - For $1 \leq i \leq n, U_i$ computes $(pk_i, sk_i) \leftarrow \mathcal{K}.\mathtt{KeyGen}(1^l)$, and sets $M_i := (pk_i, g_i)$.
- `Communication:`
  - For $0 \leq i \leq n$, U_i broadcasts M_i.

Round II.

- `Computation:`
  - Keying material:
    - U_0 chooses $k \leftarrow \{0,1\}^{p(l)}$, and for each $1 \leq j \leq n$ computes
    $$(c_j, k_j) \leftarrow \mathtt{Encaps}(pk_j, 1^l)$$
    and sets $d_j := k \oplus k_j$, $m_{0,j} := (d_j, c_j)$.
    - For $0 \leq i \leq n$, U_i sets $g_{i,j} := g_j^{\beta_i}$ for each $j \neq i$.
  - Tags:
    - U_0 computes $t_{0,j} := \mathtt{Tag}([g_{0,j}]_L, U_0||m_{0,j}||M_0||...||M_n)$ for each $1 \leq j \leq n$.
    - For $1 \leq i \leq n$, U_i computes
    $$t_{i,j} := \begin{cases} \mathtt{Tag}([g_{i,j}]_L, U_i||M_0||...||M_n), & \text{for } 1 \leq i < j \leq n \\ \mathtt{Tag}([g_{i,j}]_R, U_i||M_0||...||M_n), & \text{for } 0 \leq j < i \leq n \end{cases}$$
- `Communication:`
  - U_0 sends $(m_{0,j}, t_{0,j})$ to $U_j (1 \leq j \leq n)$.
  - For $1 \leq i \leq n$,
  $$U_i \text{ sends } t_{i,j} \text{ to } U_j (0 \leq j \leq n, j \neq i)$$
- `Verification and key computation:`
  - All users verify all tags:
    - For $1 \leq j \leq n$, U_j runs $\mathtt{Vf}([g_{0,j}]_L, U_0||m_{0,j}||M_0||...||M_n, t_{0,j})$.
    - For $0 \leq j \leq n$, U_j runs $\begin{cases} \mathtt{Vf}([g_{j,i}]_L, U_i||M_0||...||M_n, t_{i,j}), & \text{for } 1 \leq i < j \leq n \\ \mathtt{Vf}([g_{j,i}]_R, U_i||M_0||...||M_n, t_{i,j}), & \text{for } 0 \leq j < i \leq n \end{cases}$
  - If all checks are successful, U_0 sets:
  $$\mathtt{ssk}_0 := [k]_L, \mathtt{sid}_0 := [k]_R,$$
  - For $1 \leq i \leq n$, if all checks are successful, U_i runs $\mathtt{Decaps}(sk_i, c_i)$ to obtain k_i, computes $K_i := d_i \oplus k_i$, and sets:
  $$\mathtt{ssk}_i := [K_i]_L, \mathtt{sid}_i := [K_i]_R,$$

Fig. 2. Password-based group-key establishment protocol [22]

Adversaries at this level comprise stealthy malware that makes use of code-injection [38] or side-channels [7] to break both.

Vulnerabilities at the lowest level originate from programming bugs, resulting in the deducibility of secrets via non-constant-time operations [5], or else in memory leaks via memory corruption [35]. This level is handled differently from the other levels since information flow-based RV is used during testing, rather than post-deployment, due to the heavy information-flow analysis involved. Beneath these threat levels, there is the hardware level, which can pose a threat if the manufacturer cannot be trusted, say for fear of hardware backdoors, or due to its susceptibility to side-channel attacks. This can be particularly of concern if the hardware itself is a primitive for secure execution [44], 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 (RV) [14,28] in general provides two primary benefits: Firstly, monitors are typically automatically synthesised from formal notation to reduce the possibility of introducing bugs; Secondly, monitoring concerns are kept separate (at least on a logical level) from the observed system. In our case study, we make use of Larva [16], where properties are specified using Larva scripts that capture a textual representation of symbolic timed-automata. Listing 1.1 shows a simple example property specifying expected user lock-out scenario following 30 min of inactivity or else 3 successive unsuccessful login attempts (depicted in Fig. 3). Lines 12–16 define the states of the automaton, identifying the starting, normal and bad ones. Lines 17–23 specify the state transitions, with each transition also qualified by a guard condition and the action performed within `[]` and separated by \\. Lines 2–5 declare the supporting counter `x` and a timer `t`. Lines 6–10 identify the traced method calls that trigger state transitions and initialise timer objects. While Larva natively supports the monitoring of Java code through AspectJ instrumentation, it is possible to make use of an adaptor to link it up with inline hook-based instrumentation at the binary level as well, as used in previous RV-TEE work [41].

SEcube^{TM}-powered hardware [40], e.g., the USEcube™ USB token[1], is our chosen HSM for the group chat case study. The chip comprises an MCU, CC EAL5+ -accredited SmartCard, and an ultra-low power FPGA, all on the same chip, with the latter components being callable through specific MCU instructions. The MCU is an STM32F4 - ARM 32-bit Cortex-M4 CPU. Its 2 MiB of Flash memory and 256 KiB of SRAM are required to host all HSM-side of the GAKE protocol's implementation, primarily the KEM. Programming the SEcube™ is facilitated by an openly available SDK[2] exposed as a 3-layered API on the host side. On the device-side, the SDK is a layer on top of an STM32Cube[3] MCU package comprising peripheral drivers and middleware. The overall setup aids in

[1] https://www.secube.blu5group.com/products/usecube-bundle-including-5-usecube-tokens-and-1-devkit/.
[2] https://www.secube.blu5group.com/resources/open-sources-sdk/.
[3] https://www.st.com/content/st_com/en/ecosystems/stm32cube-ecosystem.html.

```
1  GLOBAL {
2   VARIABLES {
3    int x = 0;
4    Clock t;
5   }
6   EVENTS {
7    badlogin() = {*.badlogin()}
8    timer30() = {t@30*60}
9   ...
10  }
11  PROPERTY users {
12   STATES {
13    BAD { badlogins inactive }
14    NORMAL { loggedin }
15    STARTING { loggedout }
16   }
17   TRANSITIONS {
18    loggedout -> badlogins [badlogin\x>2\]
19    loggedout -> loggedin  [goodlogin\\t.
         reset();]
20    ...
21    loggedout -> loggedout [badlogin\\x++;]
22    loggedin  -> inactive  [timer30\\]
23   }
24  }
25 }
```

Listing 1.1. LARVA script for: *There are no more than 3 successive bad logins and 30 minutes of inactivity when logged in* [16].

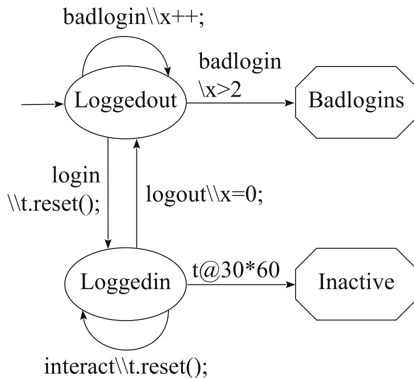

Fig. 3. Diagrammatic representation of the automaton.

developing robust firmware. For the time being, our case study focuses solely on executing all firmware on the MCU.

2.3 Related Work

Unlike model checking of cryptographic protocol design [32], literature on the verification of their software implementations is sparse. As for the available runtime verification approaches [6,33], none involve instrumentation of compiled code with the disadvantage of either dealing directly with the source code, or missing out on internal data and events. Furthermore, up to our knowledge, RV-TEE [41] is the first to attempt a comprehensive solution at securing cryptographic protocol implementations through a TEE and thereby addressing a broad threat model. Besides having been studied extensively in the setting of secure web browsing over TLS 1.3 [17], other work focused on SSH [18].

In our instantiation of RV-TEE, the LARVA RV tool, the SEcube™ HSM, and its Cortex-M4 MCU are critical enablers, and which are all proven to be of industry-grade. LARVA has been used extensively for verifying the correct operation of high-volume financial transactions systems [15]. As for SEcube™, besides its integration in various devices, several open-source projects build upon its Open SDK to demonstrate its employment in academic research [10]. Applications range from OS-agnostic file systems to securing a network's link layer,

secure password wallets, and even secure IoT hubs. As for its Cortex-M4 MCU, an extensive body of work focusing on optimised cipher implementation cover both post-quantum cryptography [1,24], as well as standard symmetric encryption [37].

3 Case Study

For the purpose of this project we need a proof-of-concept application for group communication secured by quantum-safe cryptography. Therefore, we have created a library with the working title 'GKE library', and a simple text-based group chat application based on it.

The GKE library is written in C, exposing a high-level API for running the GAKE protocol[4] from [22], and AES-128 [19] en/decryption functionality in CCM mode [43] using the shared session key `ssk` established in the GAKE protocol. The library can be used in two different configurations: a configuration which uses both a PC and the SEcube™ chip, or using only a PC.

An example session run of the chat application follows:

```
 1 ps@Diane:~/Dokumenty/Skola/phd/NATO/gke/GKE$ ./bin/chat --repeater=localhost \\
 2 --pin --id 12
 3 Secube login with password 4:test
 4 GKE|> /help
 5 Supported commands:
 6   COMMAND       ARGUMENTS         DESCRIPTION
 7   /room new   - roomname id... - create new room
 8   /room enter - roomname       - set active room
 9   /room list  -                - list active rooms and it users
10   /unsecure   - msg            - write unsecure message to current room
11   /msg        - ids... -- msg  - send unsecure multicast to ids
12   /sleep      - nsec           - sleep for nsec seconds
13   /get usage  -                - print cpu clock value
14   /exit       -                - exit application
15   /help       -                - print this help
16               - msg            - send secure message to current room
17 GKE|> /room new myRoom 42 43
18 Creating room 'myRoom'
19 Room created
20 GKE|> /room enter myRoom
21 GKE|myRoom> Hello 42
22 [myRoom]12: Hello 42
23 [myRoom]42: Hello you!
24 GKE|myRoom> /exit
```

First, the user with `id 12` launches the chat application, in this case connecting to a chat server running on the same machine (lines 1–2). Next, a PIN-based login to SEcube™ (line 3) is followed by a command dump (lines 5–16). Once a new chat session is requested (lines 17–18) with users `42` and `43`, a full GAKE protocol run is executed, returning an instance of the shared session key `ssk` with the other two users. Successful completion of the GAKE protocol results in the `Room created` prompt. Once the user accesses the newly created session (line 20), all subsequently sent messages (lines 21–23) get routed through SEcube™

[4] We describe the implementation of the GAKE protocol in more detail in Sect. 4.

for encryption with the session key. Likewise, the received encrypted messages get decrypted without `ssk` ever leaving the HSM. The `/exit` command (line 24) terminates the session.

4 Implementation of the GAKE Protocol

In this section, we present some details of our implementation of the GAKE protocol from [22]. We will focus on the implementation which uses both a PC and the SEcube™ chip.

As was mentioned in Sect. 2.2, the SEcube™ chip comprises an MCU, a SmartCard and an FPGA, supported with an openly available SDK. Our implementation uses only the MCU (STM32F4 - ARM 32-bit Cortex-M4 CPU). The reason for this is that we integrated our implementation in the available SDK and the SDK (version 1.4.1) does not provide functionality to use the FPGA or the SmartCard.

At present, our implementation does not contain protection against side-channel attacks beyond the immediate protection derived from stock hardware isolation, e.g. exploitation of non-constant time operations involving protocol secrets are still possible. We plan to address the issue of securing the implementation against side-channel attacks in future work.

4.1 Role of SEcube™

Our implementation utilizes the SEcube™ chip as follows:

- The SEcube™ chip stores the password `pw`, and the password never leaves the SEcube™. The password `pw` is a long-term password shared by members of the group. During a run of the GAKE protocol members of the group use this password for authentication.
- The SEcube™ chip generates all secret values used in the GAKE protocol, and these values never leave SEcube™.
- All computations in the GAKE protocol which involve secret data are performed by the SEcube™ chip.
- The shared session key `ssk` established in the GAKE protocol is stored on SEcube™ and never leaves it.
- We note that in this instantiation no RV component is deployed on the SEcube™. In future implementations this could be considered, bearing in mind the resource constrained nature of the HSM.

4.2 Protocol Instantiation

Our implementation targets a 128 bit security level. The protocol needs to be instantiated with proper choices of a *key encapsulation mechanism* (KEM), a Diffie-Hellman group G, a *deterministic randomness extractor*, a *message authentication code* (MAC), and a *random number generator* (RNG). Below, we describe the instantiation choices which we made in our implementation.

KEM - for KEM we chose CRYSTALS-Kyber [12], which is IND-CPA secure against fully quantum adversaries, as required by the protocol. In particular, we chose the version Kyber512 which aims at security roughly equivalent to AES-128 [19]. CRYSTALS-Kyber is one of the four KEMs selected as finalists in Round 3 of the NIST Post-Quantum Cryptography Standardization Process. It is a lattice-based cryptosystem and has favourable sizes of the key pair and the ciphertext. We use the implementation of Kyber512 from [3].

Group G - for the group G we chose the elliptic curve Curve25519 [8] which offers 128 bit security. We use the implementation of Curve25519 from [29].

Deterministic Randomness Extractor - for the deterministic randomness extractor we chose SHA-256 [34] (available within the SEcube™ API), which provides 128 bit security.

MAC - for MAC we chose HMAC-SHA-256 [27] (available within the SEcube™ API), which provides 128 bit security.

RNG - we use the RNG function provided by the SEcube™ API.

4.3 Protocol Adjustment

We made a small adjustment to the protocol to make it more amenable for resource-constrained HSM implementation (in our case, the SEcube™ chip). The adjustment is as follows.

As can be seen in Fig. 2, the protocol requires a user to compute and verify authentication tags—a computation of a tag is represented by the function `Tag` and verification of a tag is represented by the function `Vf`. Furthermore (referring once more to the figure), both these functions require $M_0||\dots||M_n$ as a part of their input. Since these functions also require the secret value $g_{i,j}$ as an input, they have to be computed on the SEcube™ chip. This means that after the user receives values M_j from other users, he needs to transport these values from his PC to the SEcube™ chip. With our instantiation of the protocol, the size of M_j is 832 bytes if $j \neq 0$ and 32 bytes if $j = 0$. If the user runs the protocol with n other users, this means that the user has to transport approximately $32+n \times 832$ bytes from a PC to SEcube™. Transporting data this large slows down the execution of the protocol. In addition, once transported, the data occupies a large amount of memory on SEcube™.

To avoid the above mentioned limitations, we adjusted the protocol as follows: instead of $M_0||\dots||M_n$, the functions `Tag` and `Vf` take as part of their input the hash value of $M_0||\dots||M_n$. This adjustment has no negative effect on the security of the protocol and reduces the number of transferred and stored bytes to just the 32 bytes of the hash.

Figure 4 shows our measurements of the execution time of the protocol, i.e., how long it takes a participant to run the protocol depending on: the number of participants involved in the protocol run; whether the participant is the initiator or not; and whether the protocol was implemented with our adjustment or not. Our measurements do not include the time required by participants to exchange messages (i.e. in our measurements we assumed that a participant receives messages from other participants instantly). These measurements were executed using SEcube™ and a PC with the following characteristics: Lenovo Thinkpad x220, Intel(R) Core(TM) i5-2520M CPU @ 2.50 GHz (2 Cores, 4 Threads, with AES-NI), RAM: 16 GiB (2×8 GiB) DDR3 Synchronous 1600 MHz (0,6 ns), OS: Linux (Linux kernel version 5.4.0–77-generic, distribution build), Ubuntu 18.04.3 LTS (Bionic Beaver) with i3wm desktop environment. We can see that for a larger number of participants our adjustment slightly improves the execution time of the protocol. What is, however, more important, our adjustment allows the protocol to run for larger sizes of the group of participants. Without the adjustment, we were not able to execute the protocol for a group of participants greater than 44. This was due to memory constraints on SEcube™. The adjustment solves this limitation.[5]

5 RV Component Implementation

In this section, we focus on the runtime verification component which checks the correct implementation of the protocol from the perspective of a chat application client. The protocol implementation and its incorporation within the chat application is organised in terms of a number of layers (Fig. 5): starting from the primitives and protocol calculations deployed on the HSM (SEcube™), which are called through the GKE library, which is in turn accessed by the chat application.

5.1 Properties

Runtime verification can be used to monitor a wide range of properties: from assertions, to temporal properties, to hyper properties [13], where each one can be considered a special case of the other: assertions as a special case of temporal properties considering one instant in time, and temporal properties being a special case of hyper properties considering only one execution of the protocol.

[5] Note that although the maximum number of participants in Fig. 4 is 60, this is not the limit of our implementation with the adjustment. The maximum number of participants for which we were able to successfully run our implementation was 789.

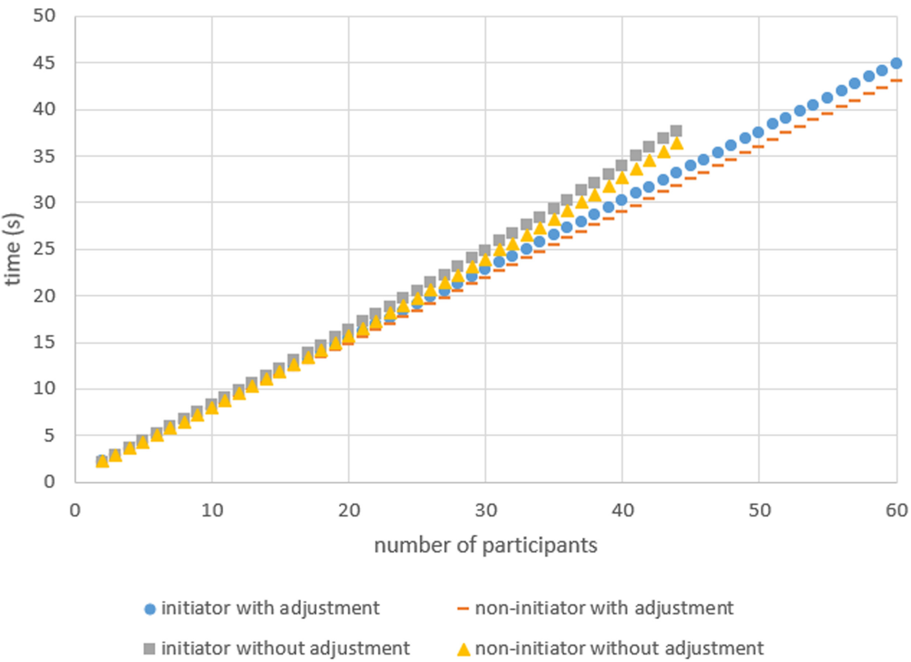

Fig. 4. Execution times of the protocol.

Table 1. Identified properties for runtime verification

Property layers	Chat app	Library	All (incl. Primitives)
Assertion	Printable decrypted chars	Sensitive data scrubbed	Non-null params, valid returns
Temporal	Chatroom lifecycle, standard sockets		Correct call sequence
Hyper		Randomness quality	

Taking each architectural layer and kind of property, we classify the properties in terms of a grid as shown in Table 1. This organisation of properties allows us to consider various aspects of the protocol systematically[6]. Other works involving the specification of properties in the case of other protocols [17,18,41] has shown a number of common properties:

[6] Strictly speaking, the chat app properties are not protocol properties and arguably not part of the RV-TEE. However, checking that the chat app works as expected, means that it is more likely that the underlying protocol is also being used in a correct manner.

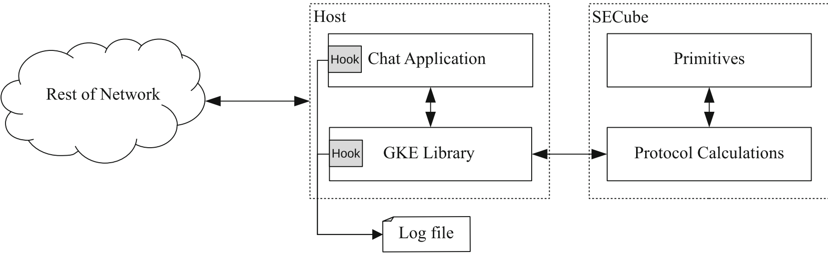

Fig. 5. Chat application architecture

Function input/output Check that function inputs are valid (e.g., non-null), and the output is valid with respect to the inputs.

Data scrubbing It is crucial to ensure that sensitive data is properly destroyed after use[7]. For example, in the case of the protocol under consideration, the generated random exponent *beta* and the ephemeral Diffie-Hellman keys of each participant need to be destroyed once secure communication is established.

Sequences of actions Protocols involve sequences of permitted actions by the participants depending on the context. In our case study, the protocol follows a high level sequence of round one followed by round two, which can be further split into sequences of actions such as loading the password, generating *beta*, and calculating the Diffie-Hellman group generator g, etc.

Randomness To ensure security, protocols depend on high quality random number generation, e.g., the exponent *beta* is randomly generated in our case study.

Together, this description list covers all the protocol properties in Table 1. The rest of the properties are chat application-specific properties, particularly properties dealing with the chatroom lifecycle.

5.2 RV Experimentation Setup

Frida, a dynamic instrumentation toolkit, and more specifically frida-trace[8] was used for instrumenting the chat application. This decision came with the advantages of not having to recompile the chat application and allowing for relatively simple JavaScript code to hook the required functions. During the frida-trace engine initialisation stage (before the first function handler is called), JavaScript code defines globally visible functions (e.g., helpers and logging) and a state object. This object is passed to every event handler to maintain information across function calls. Adding fields to the state objects allows for sequential

[7] This applies to the PC only implementation configuration. In the case of PC + HSM, the sensitive data never leaves the hardware security module.

[8] https://frida.re/docs/frida-trace/.

tracking, monitoring native pointers, keeping shadow copies of data and mapping API context to protocol run and eventually to participants.

We extended the chat application to accept scripted session runs and created two testing scenarios. In each one, the chat client with `id=1` was instrumented, while all the other clients and server were running on the same machine.

- Scenario A: 3 clients involved, with client `id=1` creating a room (following the protocol steps for an initiator participant U_0).
- Scenario B: 3 clients involved, with client `id=1` joining the room (following the protocol steps for a non-initiator participant $U_{1 \leq i \leq n}$).

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

5.3 Instrumentation Overhead Results

Starting by looking at the instrumentation (i.e. the introduction of frida-trace), Table 2 gives an overview of the overhead penalty, ranging from 0.41s to 1.38s (using the same setup as reported in Sect. 4.3). Given that sleeps are included in the scenarios, this overhead is substantial. We also try the same experiment when the implementation uses the HSM vs. when it doesn't: It is clear that there is a penalty for using the resource constrained HSM, but to a lesser extent than instrumentation. Overall, considering the instrumentation and the HSM, we have an increase of (34.98–33.03) 1.95s over the whole duration of the chat scenario. The number of experiments is too limited to extrapolate this result to the general case, however, the indication is that the overheads are within acceptable bounds, especially considering the case study of a chat application where a few milliseconds of delay for each command would go unnoticed.

Table 2. Instrumentation overheads measured per scenario, with/out the HSM.

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

5.4 Runtime Verification Empirical Results

Using RV, we checked six properties[9]: three classified as control flow, and three as data properties. The control flow properties checked the sequence of actions

[9] Our analysis leaves out assertion checks such as non-null arguments. Our reasoning is that these checks could be implemented as simple assertions in the code and thus arguably not strictly part of RV.

for the protocol and chatapp 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, assessing the quality of the generated random numbers, and finally, checking that all input characters are printable. As expected, the scenario involving U_0 required more RV effort as it plays a bigger role in the setting up of the chatroom. Looking at the property categories (control flow vs. data), there is no substantial difference. However, we note that randomness checking is rather basic—involving an entropy check, a monobits test, and a runs test. A complete state-of-the-art randomness check would certainly push the numbers substantially higher.

The properties were expressed as Larva properties and monitored offline by parsing the log files on a MacBook Pro with 2.3 GHz Quad-Core Intel Core i5 machine with 8Gib 2133 MHz LPDDR3 RAM. Each experiment was run ten times and the results shown in Table 3 are the average (excluding log parsing time).

Table 3. RV time taken by property and scenario.

Time (μs)	Control Flow				Data				RV component[a] (ms)
Scenario	Protocol	Chatapp	Socket	All	Scrub	Random	Printable	All	
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

[a] The measurement of each property only captures the time inside the property logic, leaving out other generic RV logic particularly parameter bindings and monitor retrieval; this explains why the RV component takes significantly more time than the total of all properties.

5.5 Discussion

The empirical results indicate that the overheads introduced by the instrumentation and the HSM are non-negligible. However, in this work, our main aim was to show the feasibility of the approach rather than to have an optimal solution. We note several ways in which instrumentation could have been carried out more efficiently. We use frida-trace at the level of JavaScript not only for setting up in-line hooks dynamically, but also for the instrumentation code itself that records the events of interest. Alternatively, it is also possible to make use of natively-compiled code for this purpose, but this would require more extensive testing to make sure that the application's robustness is not compromised. Furthermore, through instrumentation, we 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 keep the use of JavaScript to a minimum and limit the events to those strictly needed.

Compared to the other overheads, the time required for the actual verification is small. Therefore this could just as well be done online and in sync with the chat app execution, i.e., the application waits for the monitor's go ahead at every step. However, in case heavier RV is needed (e.g., for more thorough randomness checking), one could consider splitting the properties into two categories: those which just involve a state check (*if the current monitor state is X,*

then A is expected to happen), and those which involve checking data (*data is scrubbed; data is random*). The former category of properties can be monitored synchronously, while the latter can be monitored asynchronously.

6 Conclusion

Insecure execution of theoretically-proven communication protocols is still a major concern, particularly due to malware attacks ready to exploit any vulnerability at the various logical levels of the implementation. Moreover, with the advancements of quantum computers, coming up with novel quantum-safe protocols is inevitable. In this work, we have proposed an RV-TEE instantiation for the quantum-future group key establishment protocol from González 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.

Future work is to leverage RV-TEE further by hardening the execution environment by extending RV to the HSM code and implementing a taint inference-based RV monitor to fend off code-injection malware targeting the chat application's process memory on the PC.

References

1. Alkim, E., Bilgin, Y.A., Cenk, M., Gérard, F.: Cortex-M4 optimizations for {R, M} LWE schemes. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**, 336–357 (2020)
2. Aumasson, J.P.: Serious Cryptography: A Practical Introduction to Modern Encryption. No Starch Press, San Francisco (2017)
3. Avanzi, R., et al.: Kyber - public GitHub repository (2021). https://github.com/pq-crystals/kyber. Accessed 13 July 2021
4. Barak, B.: The complexity of public-key cryptography. In: Tutorials on the Foundations of Cryptography. ISC, pp. 45–77. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57048-8_2
5. Barthe, G., Grégoire, B., Laporte, V.: Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time". In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF), pp. 328–343. IEEE (2018)
6. Bauer, A., Jürjens, J.: Runtime verification of cryptographic protocols. Comput. Secur. **29**(3), 315–330 (2010)
7. Bernstein, D.J.: Cache-timing attacks on AES (2005)
8. Bernstein, D.J.: Curve25519: new Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer, Heidelberg (2006). https://doi.org/10.1007/11745853_14
9. Bindel, N., Herath, U., McKague, M., Stebila, D.: Transitioning to a quantum-resistant public key infrastructure. In: Lange, T., Takagi, T. (eds.) PQCrypto 2017. LNCS, vol. 10346, pp. 384–405. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59879-6_22

10. Blu5 Labs: SEcube - open source projects (2021). https://www.secube.blu5group.com/resources/open-source-projects/. Accessed 13 July 2021
11. Blu5 Labs: SEcube - reconfigurable silicon (2021). https://www.secube.eu/site/assets/files/1145/secube_datasheet_-_r7.pdf. Accessed 16 June 2021
12. Bos, J., et al.: CRYSTALS-Kyber: a CCA-secure module-lattice-based KEM. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 353–367. IEEE (2018)
13. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23–25 June 2008, pp. 51–65. IEEE Computer Society (2008). https://doi.org/10.1109/CSF.2008.7
14. Colin, S., Mariani, L.: 18 run-time verification. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 525–555. Springer, Heidelberg (2005). https://doi.org/10.1007/11498490_24
15. Colombo, C., Pace, G.J.: Industrial experiences with runtime verification of financial transaction systems: lessons learnt and standing challenges. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 211–232. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_7
16. Colombo, C., Pace, G.J., Schneider, G.: LARVA – safer monitoring of real-time java programs (tool paper). In: Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM), pp. 33–37. IEEE Computer Society, November 2009
17. Colombo, C., Vella, M.: Towards a comprehensive solution for secure cryptographic protocol execution based on runtime verification. In: Proceedings of the 6th International Conference on Information Systems Security and Privacy, ICISSP, pp. 765–774. SCITEPRESS (2020)
18. Curmi, A., Colombo, C., Vella, M.: Runtime verification for trustworthy secure shell deployment (2021)
19. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer, Heidelberg (2002). https://doi.org/10.1007/978-3-662-04722-4
20. Das, S., Russo, G., Dingman, A.C., Dev, J., Kenny, O., Camp, L.J.: A qualitative study on usability and acceptability of Yubico security key. In: Proceedings of the 7th Workshop on Socio-Technical Aspects in Security and Trust, pp. 28–39 (2018)
21. Dodis, Y., Kiltz, E., Pietrzak, K., Wichs, D.: Message authentication, revisited. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 355–374. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_22
22. González Vasco, M.I., Pérez del Pozo, Á.L., Steinwandt, R.: Group key establishment in a quantum-future scenario. Informatica **31**(4), 751–768 (2020)
23. Jager, T., Schwenk, J., Somorovsky, J.: Practical invalid curve attacks on TLS-ECDH. In: Pernul, G., Ryan, P.Y.A., Weippl, E. (eds.) ESORICS 2015. LNCS, vol. 9326, pp. 407–425. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24174-6_21
24. Kannwischer, M.J., Rijneveld, J., Schwabe, P.: Faster multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (eds.) ACNS 2019. LNCS, vol. 11464, pp. 281–301. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21568-2_14
25. Katz, J., Yung, M.: Scalable protocols for authenticated group key exchange. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 110–125. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45146-4_7

26. Kocher, P., et al.: Spectre attacks: exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1–19. IEEE (2019)
27. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: keyed-hashing for message authentication. IETF RFC 2104 (1997)
28. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebraic Program. **78**(5), 293–303 (2009)
29. Libsodium (2021). https://libsodium.gitbook.io/doc/. Accessed 13 July 2021
30. Marczak, B., Scott-Railton, J.: Move fast and roll your own crypto: a quick look at the confidentiality of zoom meetings. Technical report (2020). https://citizenlab.ca/2020/04/move-fast-roll-your-own-crypto-a-quick-look-at-the-confidentiality-of-zoom-meetings/. Accessed 15 June 2021
31. McKeen, F., et al.: Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In: Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, pp. 1–9 (2016)
32. Meier, S., Schmidt, B., Cremers, C., Basin, D.: The TAMARIN prover for the symbolic analysis of security protocols. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 696–701. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_48
33. Morio, K., Jackson, D., Vassena, M., Künnemann, R.: Modular black-box runtime verification of security protocols (2020)
34. NIST: FIPS PUB 180–2, SHA256 Standard (2002)
35. Poulin, C.: What to do to protect against Heartbleed OpenSSL vulnerability (2014). https://www.yubico.com/. Accessed 13 July 2021
36. Sabt, M., Achemlal, M., Bouabdallah, A.: Trusted execution environment: what it is, and what it is not. In: 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (2015)
37. Schwabe, P., Stoffelen, K.: All the AES you need on Cortex-M3 and M4. In: Avanzi, R., Heys, H. (eds.) SAC 2016. LNCS, vol. 10532, pp. 180–194. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69453-5_10
38. Somech, N., Kessem, L.: Breaking the ice: a deep dive into the IcedID banking trojan's new major version release (2020). https://securityintelligence.com/posts/breaking-the-ice-a-deep-dive-into-the-icedid-banking-trojans-new-major-version-release/. Accessed 15 June 2021
39. Thales: High assurance hardware security modules (2020). https://cpl.thalesgroup.com/encryption/hardware-security-modules/network-hsms. Accessed 13 July 2021
40. Varriale, A., Prinetto, P., Carelli, A., Trotta, P.: SEcube (TM): data at rest and data in motion protection. In: Proceedings of the International Conference on Security and Management (SAM), p. 138 (2016)
41. Vella, M., Colombo, C., Abela, R., Špaček, P.: RV-TEE: secure cryptographic protocol execution based on runtime verification. J. Comput. Virol. Hack. Tech. **17**(3), 229–248 (2021). https://doi.org/10.1007/s11416-021-00391-1
42. Wang, Z., Yu, H., Zhang, Z., Piao, J., Liu, J.: ECDSA weak randomness in Bitcoin. Futur. Gener. Comput. Syst. **102**, 507–513 (2020)
43. Whiting, D., Housley, R., Ferguson, N.: Counter with CBC-MAC (CCM). IETF RFC 3610 (2003)
44. Wojtczuk, R., Rutkowska, J.: Attacking intel trusted execution technology. Black Hat DC 2009 (2009)