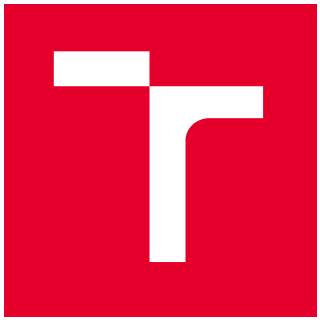

BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF TELECOMMUNICATIONS

ÚSTAV TELEKOMUNIKACÍ

SECRET SHARING AUTHENTICATION KEY AGREEMENT

AUTENTIZOVANÉ USTANOVENÍ KLÍČE S PODPOROU SDÍLENÉHO TAJEMSTVÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Pavla Ryšavá**
AUTOR PRÁCE

SUPERVISOR — **M.Sc. Sara Ricci, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study program **Information Security**

Department of Telecommunications

Student: Bc. Pavla Ryšavá ***ID:*** 203713

Year of study: 2 ***Academic year:*** 2021/22

TITLE OF THESIS:

Secret Sharing Authentication Key Agreement

INSTRUCTION:

The work is focused on the design and implementation of multi-device authentication key agreement for establishing an authenticated secure channel. The aim of the thesis is to analyze and compare existing co-signing and AKA schemes on IoT devices. Moreover, the student should explore the current solutions for allowing an authorized subset of devices to establish an authenticated secure channel. Part of the work will be the implementation of a more advance AKA protocol according to the assignment of the supervisor.

RECOMMENDED LITERATURE:

[1] Dzurenda, P., Ricci, S., Casanova, R., Hajny, J., Cika, P.: Secret Sharing-based Authenticated Key Agreement Protocol, ARES conference 2021.

[2] Zeyad Mohammad, Ahmad Abusukhon, and Thaer Abu Qattam. A survey of authenticated Key Agreement Protocols for securing IoT. JEEIT conference. IEEE, 425–430, 2019.

Date of project specification: 7.2.2022 ***Deadline for submission:*** 24.5.2022

Supervisor: M.Sc. Sara Ricci, Ph.D.

doc. Ing. Jan Hajný, Ph.D.
Chair of study program board

ABSTRACT

This thesis deals with the implementation and creation of a cryptographic library and Graphical User Interface (GUI) for the newly designed "Shamir's Secret Sharing-based Authenticated Key Agreement" (ShSSAKA) protocol. The protocol is based on the principle of AKA (Authentication and Key Agreement), Schnorr's signature and extended with Paillier's scheme to achieve multiple devices deploying signing and authentication. Benchmarks on a personal computer and RaspberryPi are also presented.

KEYWORDS

Authenticated Key Agreement, Access Control, Cryptography, Proof of Knowledge, Zero-Knowledge, Shamir's secret sharing, Paillier scheme, Sigma protocols, Schnorr's signature, Constrained Devices, Wearables, Internet of Things, Authentication, Security, Elliptic Curves

ABSTRAKT

Práce se zabývá implementací a vytvořením kryptografické knihovny a grafického uživatelského rozhraní (GUI) pro nově navržený protokol "Smlouva o autentizačním klíči na základě Shamirova sdílení tajemnství" (ang. "Shamir's Secret Sharing-based Authenticated Key Agreement", zkráceně ShSSAKA). Protokol je založený na principu AKA (autentizovaná domluva klíče), Schnorrově podpisu a rozšířen Paillierovým schématem pro možnost podílení se více zařízení na podpisu a autentizaci. Prezentovány jsou také benchmarky na osobním počítači a RaspberryPi.

KLÍČOVÁ SLOVA

Autentizovaná dohoda klíčů, řízení přístupu, kryptografie, důkazy znalosti, nulová znalost, Shamirovo sdílené tajemství, Paillierovo schéma, sigma protokoly, Schnorrův podpis, omezená zařízení, přenositelná zařízení, internet věcí, ověřování, zabezpečení, eliptické křivky

ROZŠÍŘENÝ ABSTRAKT

Zadáním této diplomové práce je implementace nově navrženého protokolu "Smlouva o autentizačním klíči na základě Shamirova sdílení tajemství" (angl. *Shamir's Secret Sharing-based Authenticated Key Agreement*, zkráceně ShSSAKA). Protokol je založený na principu Autentizované domluvy klíče (angl. *Authentication and Key Agreement*, zkráceně AKA) modifikovaná Schnorrovým podpisem a Paillierovým schématem pro možnost podílení se více zařízení na podpisu a autentizaci.

V kapitole 1 je krátce představena technologie **Internet of Things** (IoT, česky *Internet věcí*), která v dnešní době nabývá čím dál větší popularity, díky tomu, že lidem usnadňuje plnění jejich každodenních úkolů. Od těch nejjednodušších jako je např. sestavení nákupního seznamu, po ty komplikovanější jako je např. kontrola zdravotního stavu nebo kontrola znečistění ovzduší. IoT s sebou tedy nese velké výhody co se týče zvýšení životního standardu lidí. Nese s sebou ovšem i nevýhody jako je např. narůstající počet zpracovaných a posílaných dat v nezašifrované podobě nebo množství zařízení, které jsou připojovány k síti, která tato data zpracovává. Tato zařízení totiž mohou být často jednoduše penetrována, protože velká část z nich ve své základní formě nemá implementované bezpečnostní mechanismy a u části z nich to dokonce není možné kvůli omezené výpočetní kapacitě. Další velkou nevýhodou je absence standardů, které jsou ovšem postupně vytvářeny.

V rámci této kapitoly jsou také krátce představeni tři zástupci IoT frameworků. Jedná se jmenovitě o *Azure* IoT Suite z dílny společnosti Microsoft Corporation, *Google Cloud* IoT od Google LLC a AWS IoT Core vyvinutý firmou Amazon.com Inc.

Kapitola 2 představuje kryptografická primitiva, která byla použita při návrhu protokolu ShSSAKA. Zvláště se jedná o protokoly, které vedly ke vzniku **Secret Sharing-based Authentication Key Agreement** (SSAKA) navrženého kolektivem autorů v [9] a na jehož základě je nový protokol postaven.

Nejprve je v této kapitole představen princip **důkazů znalosti** (angl. *Proof of Knowledge*), což je interaktivní mechanismus poskytující výpočetní znalost o stanoveném tajemství. Tento mechanismus probíhá mezi dvěma stranami – mezi entitou dokazující znalost tajemství (tedy *dokazujícím*) a mezi entitou, která ověřuje správnost výroku (tedy *ověřovatelem*). Tento princip může fungovat i bez odhalení samotného tajemství ověřovateli. V takovém případě se jedná o metodu **důkazů s nulovou znalostí** (angl. *Zero-Knowledge Proof of Knowledge*).

Zástupcem principu důkazů s nulovou znalostí může být např. **Schnorrův protokol**, což je zástupce rodiny tzv. Σ*-protokolů*. Je založen na problému diskrétního logaritmu a používá třícestnou strukturu typu *výzva-odpověď*. Tento protokol může být také upraven do jednocestné struktury digitálního podpisu, označovaného jako *Schnorrův podpis*, kde je vytvořena dvojice hash a podpisu prokazovaného tajemství,

které jsou hromadně poslány ověřovateli.

Další primitivum, které bylo v rámci kapitoly 2 představeno je **autentizované ustanovení klíče** (angl. *Authentication Key Agreement*, zkráceně AKA). Tyto typy protokolů jsou v této době masivně používány, protože spojují techniky autentizace uživatelů a ustanovení symetrického klíče do jedné zahajovací komunikace, čímž se sníží počáteční složitost komunikace. AKA protokoly jsou založeny na důkazech s nulovou znalostí a jedná se o třícestný protokol komunikující mezi uživatelem a serverem.

Následně je představeno *schéma sdílení tajemství* (angl. *Secret Sharing Scheme*), které umožňuje distribuci částí tajemství mezi více účastníků. Jednotlivé části samy o sobě o původním tajemství nic neprozradí, ale pokud je dán dohromady definovaný počet částí, je možné jej zrekonstruovat. Počet částí, které musí být dány dohromady, je udán před distribucí. Jedná se tedy o schéma využívající práh t a jsou tak často v odborné literatuře označována dvojicí čísel (t, n), kde číslo n udává počet zařízení, která byla k distribuci použita. Výhodou tohoto schématu je, že není třeba použít všechny části nebo jejich určitou kombinaci k tomu, aby bylo tajemství zrekonstruováno. Zástupcem tohoto schématu je např. **Shamirovo sdílení tajemství** (angl. *Shamir's Secret Sharing*), které k distribuci tajemství používá polynomy stupně $t - 1$ a k jeho rekonstrukci používá Lagrangovu interpolaci.

Dalším představeným primitivem je **Paillierův kryptosystém**, což je pravděpodobnostní asymetrický algoritmus pro kryptografii veřejného klíče. Jeho hlavní výhodou je vlastnost aditivní homomorfie, která umožňuje provést součtovou operaci nad zašifrovanými daty, takže si získal popularitu zvláště ve strukturách, kde je potřeba zpracovávat data s citlivým obsahem. Hlavní nevýhodu pak představuje jeho nízská rychlost.

V rámci 28. studentské EEICT konference, jsme provedli a prezentovali optimalizaci Paillierova schématu pomocí předpočítávání zvolených hodnot na základě [17]. V rámci vytvořeného článku jsme implementovali základní Paillierovo schéma označené jako *schéma 1*, a *schéma 3*, které je již jistou optimalizací schématu 1. Dále bylo implementováno předpočítávání hodnot zprávy a ruchu používaných při šifrování a předpočítání některých konstantních hodnot, jako je např. n^2 nebo inverzní parametr μ používaný při dešifrování (implementovaný kód je k dispozici na GitHubu[1]). Hlavním cílem bylo co nejvíce zrychlit proces šifrování, které je často používáno v rámci protokolu ShSSAKA navrhovaného touto diplomovou prací.

Jako poslední byl představen protokol **smlouvy o autentizačním klíči na základě sdíleného tajemství** (angl. Secret Sharing-based Authentication Key Agreement), na kterém byl postaven protokol ShSSAKA navrhovaný v rámci této

[1]EEICT_Paillier repozitář https://github.com/Norted/EEICT_Paillier

práce. Tento protokol je založen na schématech AKA rozšířených o Shamirovo sdílení tajemství a pracuje mezi několika entitami. Těmito entitami jsou server, klient a definovaný set klientských zařízení. Tento protokol umožňuje distribuci klientského tajného klíče do dalších klientových zařízení, která se poté podílejí na jeho autentizaci. Tato zařízení mohou být např. wearables (jako třeba chytré hodinky), chytrý telefon, Raspberry Pi nebo notebook. Nevýhodou tohoto protokolu ovšem je, že pro získání tajemství je potřeba mít neustále k dispozici všechna zařízení.

V kapitole 3 byly představeny knihovny, které byly použity pro implementaci kryptografické knihovny navrhovaného protokolu ShSSAKA. Jsou to: kryptografická knihovna *Open*SSL, knihovna *GIMP ToolKit* (GTK) pro tvorbu GUI a knihovna *c*JSON pro parsování JSON souborů.

V kapitole 4 je představeno rozšíření protokolu SSAKA, jehož výsledek byl pojmenován jako **smlouva o autentizačním klíči na základě *Shamirova* sdílení tajemství** (angl. *Shamir's Secret Sharing-based Authentication Key Agreement*), aby byly tyto dva protokoly odlišeny, i když v obou těchto případech se Shamirovo schéma používá. Hlavním rozdílem mezi těmito protokoly je, že v průběhu distribuce tajemství je použito Paillierovo schéma pro možnost podílení se na výpočtu dílu tajemství všech zařízení, a tudíž je zajištěna možnost použití jakékoli podmnožiny kombinací zařízení pro autentizaci.

Kapitola 5 také popisuje implementaci knihovny navrhovaného protokolu za použití výše zmíněných technologií a prezentuje výsledky benchmarkingu Paillierova schématu (výsledky získané v rámci studentské EEICT konference) a benchmarkingu vytvořené knihovny. Kryptografická knihovna ShSSAKA byla vyvinuta v prostřední Microsoft Visual Studio Code (verze 1.67.2) na operačním systému Ubuntu 64x 20.04.4 LTS (Linux OS) v programovacím jazyce C a je veřejně dostupná pod MIT licencí na platformě GitHub[2].

Repozitář obsahuje dva spustitelné soubory. Soubor `main_test.c`, jež je určen pro testování jednotlivých částí knihovny, a soubor `GUI.c`, který pomocí knihovny GTK implementuje grafické rozhraní pro uživatelsky přívětivou ukázku funkčnosti implementované knihovny. Kód knihovny je rozdělen do dvou složek: `c_files` a `header_files`, kde první z nich obsahuje jen zdrojové kódy (tedy soubory s příponou `*.c`) a druhá jen hlavičkové soubory (tedy soubory s příponou `*.h`). Ve složce `header_files` je také soubor `globals.h`, ve kterém jsou obsažena makra ovládající např. bitovou velikost parametrů (makro BITS), počet zařízení klienta (makro G_NUMOFDEVICES) nebo stupeň polynomu pro Shamirovu distribuci (makro G_POLYDEGREE).

Benchmarking byl prováděn na PC s Intel(R) Core(TM) i7-4510U CPU, 2.00

[2]Simulator-for-ShSSAKA repozitář https://github.com/Norted/Simulator-for-ShSSAKA

GHz se 4 jádry a operačním systémem Ubuntu 64-bit v20.04.4 LTS a na Raspberry Pi 3 Modelu B+.

Nejprve jsou uvedeny výsledky měření Paillierova schématu ve verzi 1 a 3 při použití jednotlivých optimalizačních technik předpočítávání. Z nich je nejúčinnější předpočítávání hodnoty ruchu, které celé schéma urychlí o jednotky sekund. Zde je nutno dodat, že výsledky byly získány pouze na výše uvedeném PC.

Následně jsou uvedeny výsledky měření implementované knihovny na obou definovaných zařízeních (tzn. jak na PC, tak i na RPi). Měřily se časové závislosti jednotlivých částí protokolu na změně počtu celkových zařízení podílejících se na distribuci sdíleného tajemství a také na změně stupně distribučního polynomu (tzn. změna minimálního počtu zařízení pro rekonstrukci sdíleného tajemství).

V případě *inicializace* protokolu (tzn. generování klíčů pro jednotlivá zařízení, klienta a server) čas generace klíčů rostl jak u zařízení, tak u serveru se zvyšovaným počtem zařízení. Se zvyšovaným stupněm polynomu však rostl pouze čas generace zařízení. Je to dáno tím, že v případě, kdy rostl počet zařízení, rostl počet potřebných výpočtů u Lagrangovy interpolace, která je potřebná pro výpočet klientova veřejného klíče uloženého na serveru, zatím co při zvyšování stupně polynomu zůstával konstantní.

V případě *autentizace* a *verifikace* rostl pouze čas verifikace ať už v případě zvyšovaného počtu zařízení n nebo zvyšování stupně polynomu t. Čas autentizace zůstával ve všech případech konstantní.

RYŠAVÁ, Pavla. *Secret Sharing Authentication Key Agreement*. Brno: Brno University of Technology, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2022, 82 p. Master's Thesis. Advised by M.Sc. Sara Ricci, Ph.D.

Author's Declaration

Author:	Bc. Pavla Ryšavá
Author's ID:	203713
Paper type:	Master's Thesis
Academic year:	2021/22
Topic:	Secret Sharing Authentication Key Agreement

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature[‡]

[‡]The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to thank my supervisor Mrs M.Sc. Sara Ricci, Ph.D. for professional guidance, consultations, patience and suggestions for my Master thesis. I would also like to thank my family and friends for making this happen and finally, special thanks to my fiancé Lukáš for patience, undying support, constant encouragement and many sleepless nights. I couldn't have done it without You.

Contents

List of Figures

List of Tables

Listings

Introduction

In the Internet of Things (IoT) [13] era, with a big amount of data flowing through the internet, there is an increasing emphasis on data secrecy and the security of their transmission. These features are achieved by *symmetric* and *asymmetric protocols*. Symmetric cryptography ensures mainly data encryption whereas asymmetric cryptography authentication and data integrity. Symmetric cryptography uses the same key for the encryption and decryption process (i. e. $k_{enc} = k_{dec}$). The asymmetric cryptography uses two sets of keys – a *public key pk* and a secret *private key sk*. Keys distributed in this way can provide data integrity, i.e. evidence that there was no unauthorised manipulation of the data, data confidentiality, and identity authentication of the owner of the keys.

Symmetric and asymmetric protocols use keys to encrypt or sign the protected data, and this leads to the problem of a secure symmetric key agreement between the involved parties since asymmetric cryptography does not work with the same encryption and decryption key. For the key arrangement is usually used the *Diffie-Hellman* (DH) protocol. This protocol is based on the discrete logarithm problem and provides a secure symmetric key agreement. The DH protocol, can be extended to work on *Elliptic Curves* (EC). However, one of the main problems still holds: how do the parties know that there is no "man-in-the-middle" or that the involved parties are, whom they claim to be?

This moment is where asymmetric cryptography steps in. One of the properties of the asymmetric type of cryptography is the use of two sets of keys. The secret key can serve as a kind of hand-written signature, thus, providing proof that the given party is who they claim to be. The process of the entity identification confirmation is so-called *authentication*.

Protocols that use authentication in the process of symmetric key agreement are so-called **Authentication Key Agreement** protocols (AKA, or sometimes marked as *Authentication Key Exchange* (AKE)) and are currently one of the most used cryptographic primitives.

This thesis focuses on the extension of a *Secret Sharing-based Authentication Key Agreement* (SSAKA) protocol proposed in [9]. Our variant combine the basic AKA scheme with the Shamir's Secret Sharing scheme. The authentication of the involved parties is provided with the **Schnorr's Signature algorithm**.

By adding the Shamir's Secret Sharing to SSAKA protocol, the client's secret can be shared among the other involved parties (except the server). This allows sharing of the key among several devices of a user(s) and, therefore, to set up a multi-device authentication. The key can be shared in more devices of the same user and a threshold, e.g., 3 out of 5, will be invoked in the authentication process.

This permits to increase in the security of the AKA protocol, where more devices need to be compromised in order, to "fake" the authentication.

Our extension of SSAKA allows that only a defined number of devices needs to be involved (not all of those as before). The newly proposed protocol was named *Shamir's Secret Sharing-based Authentication Key Agreement* (ShSSAKA).

For the distribution of the client secret to work and be secure, the *Paillier scheme* was included in the computation process as well because of its homomorphic ability. Since the Paillier scheme is time-demanding, an optimisation, especially of the encryption, was done. The optimisations were achieved through pre-computation of certain values as suggested in [17]. In this work, we use the pre-computation of the message and noise values which are then saved into *JavaScript Object Notation* (JSON) files. Also, a pre-computation of certain static values, such as the n^2 or μ, are pre-computed and are directly saved into the Paillier key-chain.

The implementation of the protocol library is written in C language an is publicly available on GitHub[1]. For the protocol implementation is used the OpenSSL library, which is an open-source implementation of the *Secure Sockets Layer* (SSL) and *Transport Layer Security* (TLS) protocols. This library also includes the implementation of the Big Numbers (BN) data type and also a user-friendly implementation of the EC. The BN implementation also contains basic mathematical functions used in cryptographic protocols such as addition, multiplication and modulo.

For the *Graphical User Interface* (GUI) implementation was used the GIMP ToolKit (GTK) widget toolkit, which is a C implemented library for (not just) GUIs. The GUI can be used as a demo-tool for the library use-case. It also demonstrates the functionality of the protocol and simulates the real-case scenario.

Part of the thesis is dedicated to the bench-marking of the implemented ShSSAKA protocol library and the optimised Paillier scheme. The main focus in the ShSSAKA is given to the change of the average time over changing the number of devices n or the degree of the polynomial t. The experimental results were run on a PC and a RPi. In particular, we used an Intel(R) Core(TM) i7-4510U CPU at 2.00 GHz with 4 cores and Ubuntu 64-bit v20.04.4 LTS operating system and a Raspberry Pi 3 Model B+.

The optimisation and comparison of the gained bench-mark results of the Paillier scheme were presented in EEICT Student Conference 2022 [37].

[1]Simulator-for-ShSSAKA repository on GitHub https://github.com/Norted/Simulator-for-ShSSAKA

1 Internet of Things

Internet of Things (IoT) [13] is a term used for a interconnected network of "things". The idea is to create a network that will be able to communicate and transfer data about the usage and environment without human interaction. To IoT network can be connected almost to anything, either from physical devices such as sensors or smart devices or through the Internet to people.

The IoT network works with the web-enabled stand-alone (smart) devices, embedded systems (that consists of micro-controller and other parts such as power source, Graphical Processing Unit (GPU) and memory) and boards (such as Raspberry Pi and Arduino). These devices can collect, send and process such data acquired from the environment.

Devices can be in general connected to the IoT platform, which integrates data from the network via either Internet (e. g. through WiFi access point), Bluetooth, NFC, 4G, 5G or by any other wireless technology. Connected "things" then communicate or cooperate on the information gained with each other directly, or they send data for analysis to the cloud, or they analyse them locally [10].

The communication between the devices can be handled either directly, or through an IoT Gateway. The latter usually facilitates the device-device or device-cloud connections, performs data pre-processing, filtering and analysis and many other functions such as security management, system diagnostic and configuration management. IoT Gateways are mainly used either in a big IoT networks and in scenarios, that needs the real-time computing, or in the process of critical data (sometimes referred to as *Edge Computing*) [30].

The goal is to help people to live "smarter", i.e., in the means of easier and somehow safer, because the IoT can fully automate many activities. An example of the IoT use can be a smart home. Part of the smart home is a smart fridge that keeps track of the amount of food inside it, and if something is consumed, the fridge sends the information about the missing product to the shopping list inside another system without the owner's interaction. The owner then does not need to keep track of what's missing and can have another device that will send the created shopping list to an online shop at a regular interval to have all the necessary ingredients always available. If well go further with this example, such automation can be very useful. For instance, in the hospitals, we would be able to always have important drugs in stock based on the counted index of use in certain periods like falls when many people fall ill with flu and there can assure smooth course of such illness epidemics.

However, the use of IoT network does not end just by smart home. IoT is used also in industry (also referred as *Industrial Internet of Things* (IIoT)). Example of usage is monitoring the manufacture processes in order to improve the product

quality, or for monitoring the human health and environmental conditions [31].

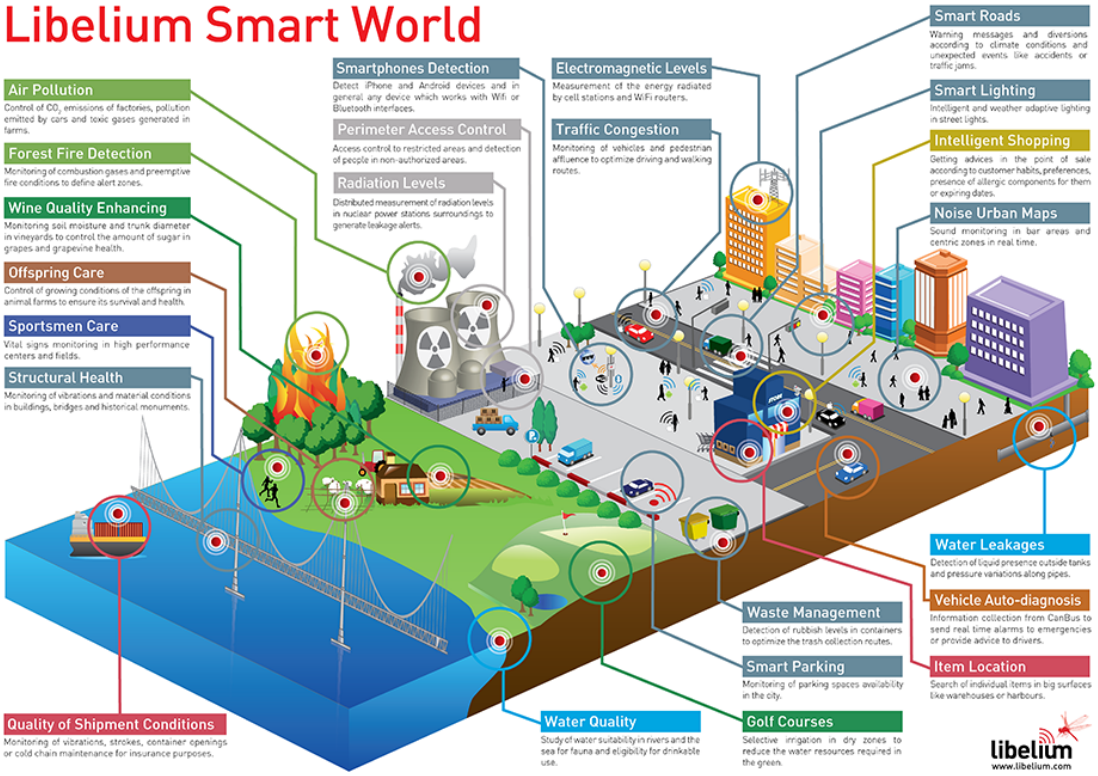

Fig. 1.1: Libelium's application of sensors for a Smarter world, adapted from [21].

Also, the idea of the "smart world" is not inconceivable. For example, the Spanish corporate *Libelium* [22] focuses on IoT integration into daily life. They also offer many IoT solutions for many branches such as agriculture, waste and water management and smart cities. Moreover, they have their e-shop with their products enabling the proposed schemes, and they manage a blog "Libelium world" that publishes news in the IoT development. On this blog, they have published an idea of the application of sensors for a "smart world" depicted in Figure 1.1.

1.1 IoT frameworks

In this section, some of the existing frameworks from known distributors such as Google LLC or Microsoft are shortly introduced. The mentioned platforms are *Azure* IoT Suite, *Google Cloud* IoT, and *Amazon Web Services* (AWS) *Core*.

Azure IoT Suite Platform [8] from Microsoft Corporation is a scenario-focused solution on one platform. It is a collection of managed platform services in edge

and cloud with included end-to-end security, operating systems for the devices in the IoT network and other tools, which can help with the building, deployment and management of the IoT applications [8].

Google Cloud IoT The Cloud IoT [23] from Google LLC presents a service for secure connection and data management and ingestion received from globally dispersed devices. It supports the *Hypertext Transfer Protocol Secure* (HTTPS) and publish-subscribe network protocol *Message Queue Telemetry Transport* (MQTT) for message transport between devices. Service runs on Google's server-less infrastructure that allows the almost real-time reaction to changes.

AWS IoT Core Amazon Web Services (AWS) IoT [15] is a platform developed by Amazon.com Inc. It is presented as easy and secure with the ability to connect billions of devices to the IoT network and the ability to route trillions of messages to the services without the infrastructure management. This platform supports many communication protocols used in IoT such as MQTT (also in the variant over WSS), HTTPS and *Long Range Wide Area Network* (LoRaWAN). Moreover, AWS includes authentication, end-to-end encryption, and implements many other tools for indigestion of the received data.

1.2 Security analysis

In order to sum up the IoT properties, security analysis can be useful. Accessing to the processed data can be gained from anywhere at any time which saves time and money and it can help improve the quality of services and the standard of living through automation. However, like any technology, IoT also comes with some disadvantages, especially with the increasing number of connected devices to the network as the amount of transferred information between the devices, their collection, storage, management and analysis. Another great disadvantage is the absence of many standards. Of course, some are already approved and applied like the MQTT, which is a publisher-subscriber messaging model, LoRaWAN for IoT media access control, and many other protocols such as WiFi standards, Bluetooth, and HTTPS [12]. However, there exists an IEEE Internet of Things Initiative that is constantly working on the improvements in this direction. Nowadays the Initiative works on some current projects which are e.g., the IEEE 1451-99 that focuses on developing the standard for harmonisation and security of the IoT devices and systems or the IEEE P2510 that focuses on the quality measures, parameters and definition for the implementation of the IoT sensors [3].

Another massively discussed topic is the security of the IoT network since there is not just the thread of the data compromise, but also the threat of active control of the device itself. If one of the connected devices is compromised, then the whole network becomes corrupted. In the "hardware level", there is a gap between the hardware of the IoT devices themselves and the existing frameworks because of the requirements each device and platforms have because some frameworks are implemented with standard programming languages, others are programming language-specific. However, the security on each platform tends to enforce the same security standards or at least trends. For instance, for secure message transfer between devices, the developers use the SSL/TLS protocol, and they usually support various cryptographic primitives based on the supported cryptographic libraries or modules and based on the computation limitations of the involved parties and used communication protocols. For access control, there are mainly two types of approaches – using the sandbox techniques or implementation of their own access control model [1].

Based on these precautions, the IoT architecture of each platform should be secure if the administrators have good practices when setting up the IoT network. The main problem are Commercial of the Shelf (COTS) microcontrollers which represents the majority of the devices used in the IoT network. Such deployed COTS devices are without hardware security support, and they are not taken into consideration in the security architectures used in the frameworks mainly because of the lack of computing power needed for cryptographic operations. Some architectures try to avoid this problem by injecting the secret keys into the device, but the keys are used then throughout the whole lifetime of the device, which is, in the case of the compromise of the device both in the "hardware level" (in the means of stolen or physically accessed), and in the "software level" a security problem [19].

2 Used cryptographic primitives

The Authentication Key Agreement protocols are one of the most frequently used cryptographic primitives. They allow the establishment of a secure communication channel between two or more entities by the agreement on a symmetric session key. These protocols are a combination of key agreement protocols with the digital signature algorithms and are thus more secure than, for instance, the DH protocol. They do not provide just the symmetric key agreement, but they also provide mutual authentication of the involved parties.

This chapter introduces used cryptographic primitives, that are used in the proposed ShSSAKA protocol introduced in the next chapter.

First, in Section 2.1, the **proof of knowledge** mechanism used in the basic AKA protocol is explained. Specifications on the *Schnorr protocol* and the *Okamoto* Σ-*protocol* are also given. Following, Section 2.2 describes the separate AKA protocol, that forms the base of the SSAKA.

After that, in section 2.3 is explained, how can be a secret distributed between more parties and is introduced the *Shamir's Secret Sharing* (SSS) protocol, that is then used in proposed extension to SSAKA protocol. In Section 2.4 is described the *Paillier scheme* used during the computation of the ShSSAKA secret keys of the devices. In Section 2.5 is described the *Secret Sharing-based Authentication Key Agreement* (SSAKA) proposed by collective of authors in [9].

2.1 Proof of Knowledge

Proof of knowledge [11] is an interactive cryptographic mechanism providing computational proof of "knowledge". This mechanism works between two parties, namely the **prover**, that wants to prove his knowledge of some secret to a second party, and the **verifier**, that accepts or rejects the veracity of the published statement. This scheme can also work without the necessity to unravel the secret to the verifier, as in the case of so-called *zero-knowledge proof*.

The two-party proof of knowledge, *without the knowledge-error*, is built upon two properties:

- `Completeness` that states that if the send statement to the verifier is true, the verifier will always accept the proof, and
- `Validity` that requires that the success probability of knowledge must be at least as high as the success probability of the *prover* in convincing the *verifier*, as this property guarantees, that some prover without the knowledge can succeed in convincing the *verifier*.

With the knowledge-error is the validity property replaced with the property **soundness**, that states, that there is only a small probability of accepting a wrong statement by the *verifier* [4].

To prove various statements, such as those built on the discrete logarithm problem (also in a specific form), can be verified by using so-called **sigma protocols** (sometimes marked as the Σ-*protocols*). These protocols use a three-way challenge-response structure with the first step called the *commitment*. The *prover*, to verify some knowledge interact with the verifier as follows:

1. The *prover* generates parameter r from $\mathbb{Z}_q^*$ at random, counts parameter $c = g^r$, where q and g are a published parameters, and sends the parameter c to the *verifier*.
2. The *verifier* generates parameter e from $\mathbb{Z}_q^*$ at random and sends it to the *prover*.
3. After receiving parameter e, the *prover* counts and sends to the *verifier* parameter $z = r + ew \mod q$, where $w \in \mathbb{Z}_q^*$ is the statement to verify.

The *verifier* than accepts if the $g^z \equiv h^e c \mod q$, where $h = g^w$, is true [38]. One of the most used and simplest sigma protocol is the **Schnorr protocol**, which permits the proof of knowledge of statements regarding the discrete logarithm problem. Schnorr protocol is depicted on Figure 2.1.

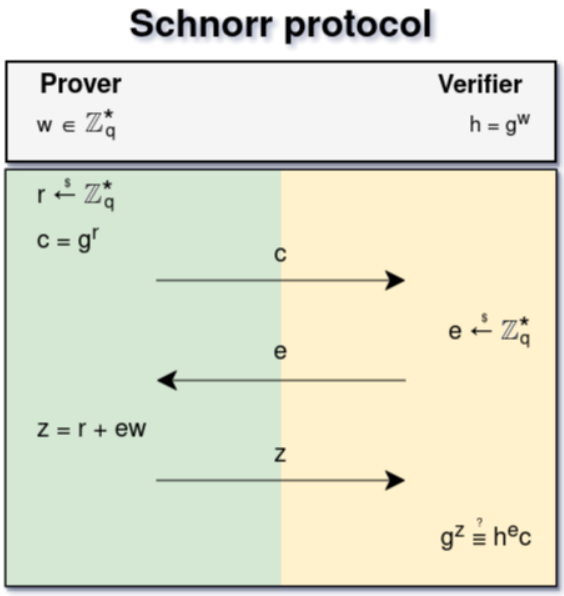

Fig. 2.1: Schnorr protocol scheme with the discrete logarithm based statement w.

Schnorr protocol has a set of properties:

- `Completeness` property, that states, as mentioned above, that if the send statement to the verifier is true, the verifier will always accept the proof,

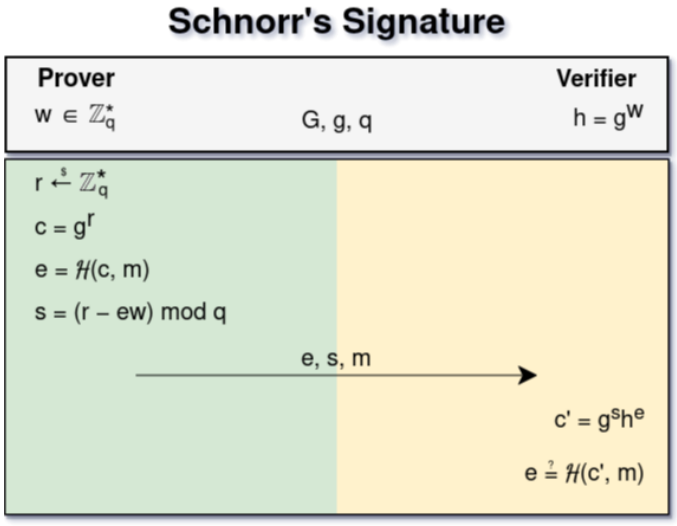

Fig. 2.2: Schnorr's Signature algorithm scheme.

- `Special Soundness` property, that states that if the statement is false, then no cheating prover can deceive the verifier about the veracity of the statement, and
- `Special Honest Verifier Zero-Knowledge` property that states that the verifier does not find out any information about the verified statement, except about its authenticity.

The Schnorr protocol modification can provide a signature protocol. The scheme is then called the *Schnorr's signature* algorithm [39]. Such protocol can then provide the mutual authentication of involved parties. The protocol also provides randomness generation such as the challenge values, that protect the algorithms from replay attacks and provide the ability to perform the DH protocol for the establishment of a symmetric key. The generated key can then stand for a session key. Schnorr's signature algorithm scheme is depicted in Figure 2.2.

There exists also an "extension" of the Schnorr protocol – the Okamoto Σ-protocol, which allows extending the protocol with the ability to use two private keys and provision of the witness indistinguishability. Extension scheme is shown in Figure 2.3 [27].

2.2 Authentication Key Agreement scheme

Authentication Key Agreement (AKA, sometimes referred to as *Authentication Key Exchange* (AKE) [5]) is a challenge-response based protocol used to negotiate a symmetric session key between two mutually proven parties. By "symmetric key" is

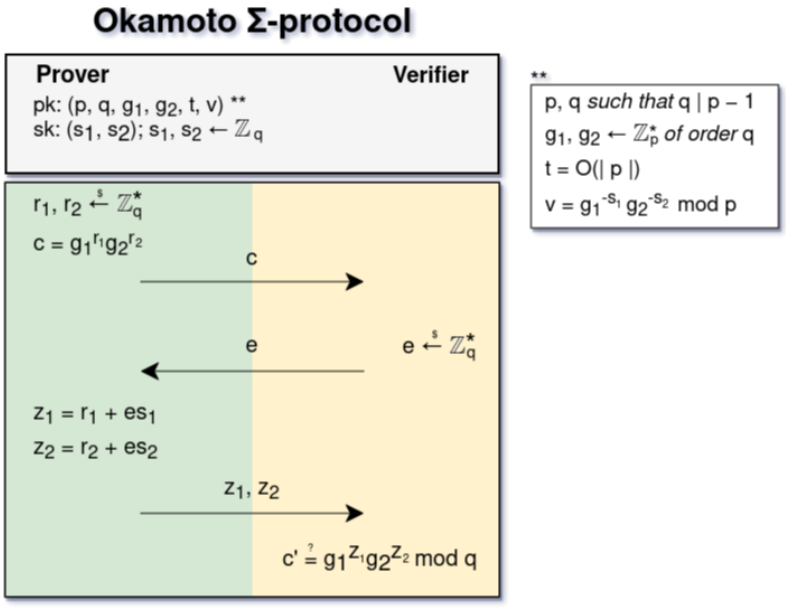

Fig. 2.3: Okamoto Σ-protocol.

meant a key for a symmetric cryptography that uses the same key for data encryption and decryption (i. e. the $k_{enc} = k_{dec}$). In AKA is used the concept of *Diffie-Hellman algorithm* (DH), which is a cryptographic algorithm for symmetric key agreement between two parties.

The basis of the protocol is the zero-knowledge-proof, which is a method that allows veracity of a statement between the two parties (the *verifier* and the *prover*), without uncovering any information held in the declaration. The description of the proof of knowledge method is in section 2.1.

The user (or the device) can also be an active part of the authentication as the process can be carried out by using a PIN, password or a Public Key Infrastructure to prove the identity of the carrier [20]. Among the mutual authentication, this protocol also allows the collateral of data integrity, information and data encryption and can protect the agreement from rogue devices. This protocol is nowadays used in the 3G networks or for the one-time password generation mechanisms for digest access authentication.

The AKA scheme employs two parties:

- `Client` – a user with an access to a specific (online) service. To gain access to the service, the user has to authenticate by providing the access key. The access key is usually securely stored on the user's device. The device can be anything that possesses enough computational power. Such devices can be wearables, microcontrollers, mobile phones, personal computers etc.

- `Server` – any device, often a powerful computer, that provides the desired service (or more services). Nowadays, in the time of IoT, are however on the rise computationally weaker devices such as Raspberry Pi, Nvidia's Jetson or other single-board computers that are much cheaper and allow more use-cases, such as the desired service server.

The algorithm of the AKA consists of three steps in which the client authenticates to the server and receives the symmetric session key κ. At first, the client sends the **Client Hello** message containing:

- the unique *Client Identification* (CID) assigned to the client at the registration process,
- *cypher suite* which specifies the client preferred cryptographic algorithms (can be listed as all supported algorithms), and
- the client's challenge, which is usually a randomly generated number protecting the system against replay attacks.

When the server receives the *Client Hello* message, it creates a message Y, which includes the CID, used cypher suites, clients challenge and confirmation of chosen cypher suite. Then the server performs the first part of the `Server SignVerify` algorithm, which generates the *Schnorr's signature* σ from the created Y message and a generated number r at random with the server's secret key and provides in this way an authentication proof. The random value r represents the session key but also the challenge of the server since it is generated for each session anew. The σ and Y are then sent back to the client in the **Server Hello** message.

After receiving the σ and Y, the client performs the `Client ProofVerify` algorithm for the veracity check of the sent server signature, computes the session key κ and also the proof of knowledge π of the client secret key. The π value is than sent back to the server to the second part of the `Server SignVerify` algorithm where does the server compute its own (but the same, if the computed values are correct) session key κ and verifies the clients proof of knowledge of its secret key. The above described algorithm is depicted in Figure 2.4 [9].

The AKA protocol should also meet all following security attributes:

- `Mutual authentication` that states that both parties authenticate each other during the protocol run (i.e. through *Schnorr's signature protocol*),
- `Key compromise impersonation` that states that if a long-term key of one entity is compromised, then the entity that gained such key can only impersonate the compromised entity and no other,
- `Parallel session attack` that states that knowledge of a previous session key does not allow the adversary to compromise other session keys, which can be achieved, for instance, by using randomised session key values,
- `Denial of service attack` that states that an evil CID and IP address will

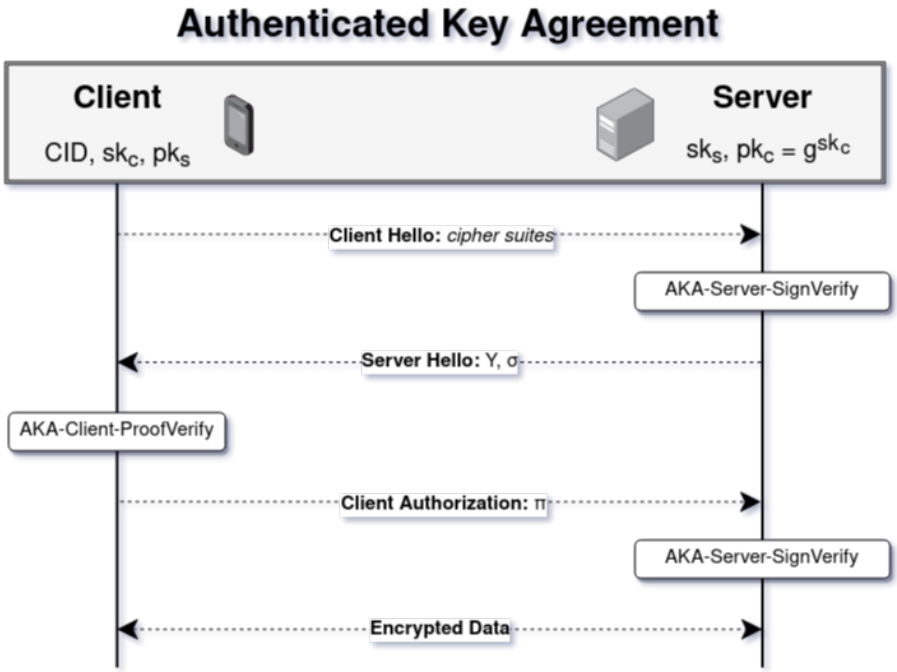

Fig. 2.4: Basic Authentication Key Agreement protocol scheme.

be black-listed if an evil entity would try to consume the server resources by sending a fake login,

- `Replay attack` that states that if an attacker catches the communication between the parties, then the use of this communication to gain access to the service will be meaningless,
- `Man-in-the-Middle attack` that states that the protocol implements such mechanism that prevents this sort of attack such as the Schnorr's signature algorithm [26].

2.3 Secret Sharing scheme

A secret sharing scheme (or *secret splitting* [18]) is a method used for a secret distribution between a group of a defined number of parties, where each then stores a share of the secret. Distributed shares alone are of no use. Such distributed secret can then be reconstructed by, at least, a defined minimum of parties. Such minimum is called the *threshold* and is in most secret sharing schemes marked as t and with the number of involved parties can describe the scheme as (n, t)- or (t, n)-threshold schemes. There are two particular cases of trivial sharing, and those are cases where $t = 1$, where is the whole secret distributed to all participants and where $t = n$, where to recover a secret, all parties must participate. Of course, there exist constructions that can reconstruct the given secret without the defined threshold,

such as the *Secret Sharing Scheme Realising General Access Structure* proposed by Mitsuru Ito, Akira Saito, and Takao Nishizeki [16].

The scheme employs a **dealer** that owns the secret and gives out the shares of such secret to n-number of *players* under specific conditions, which allows the secret reconstruction operation from the distributed shares. The requirements for the secret sharing are two:

- `Correctness` that states that the distributed secret can reconstruct any authorised set of party $\geq t$, and
- `Perfect Privacy` that states that every unauthorised set of parties can learn no theoretical information about the secret from their shares.

Secret sharing schemes can use many cryptographic protocols as building blocks such as multiparty computation, generalised oblivious transfer, or attribute-based encryption [9]. Two most known secret sharing schemes are the *Shamir's Secret Sharing* (SSS) which is a simple (t, n)-threshold scheme and the *Blakley's scheme*, which use the $(n-1)$-dimensional nonparallel hyperplanes to recover the secret that is represented by the planes intersection [7].

2.3.1 Shamir's Secret Sharing

The Shamir's Secret Sharing (SSS) [40] was formulated by Adi Shamir in 1979 in his article "How to share a secret". There he first proposed a simple, ideal and perfect (t, n)-threshold scheme based on polynomial interpolation over finite fields in 2D plane [40]. The share is distribute by the computation of a polynomial of degree $t-1$. For instance in order to reconstruct the secret with *three devices* chosen from the complete set of n the distribute polynomial will be

$$f(x) = ax^2 + bx + \kappa, \tag{2.1}$$

where κ is the secret to distribute and a and b are at random chosen values from $t-1$ set.

The distributed share, if distributed correctly, can be in case of SSS recovered by Lagrange interpolation stated in Equation 2.1.

$$L(x) := \sum_{j=0}^{k} y_j \mathcal{L}_j(x), \tag{2.2}$$

where y_j is the y coordinate from the given set of $t+1$ data points (x_j, y_j) with no two values x_j equal, and $\mathcal{L}_j$ is a Lagrange polynomial basis stated in Equation 2.3.

$$\mathcal{L}_j(x) := \prod_{\substack{0 \leq m \leq k \\ m \neq j}} \frac{x - x_m}{x_j - x_m} = \frac{x - x_0}{x_j - x_0} \cdots \frac{x - x_{j-1}}{x_j - x_{j-1}} \cdot \frac{x - x_{j+1}}{x_j - x_{j+1}} \cdots \frac{x - x_k}{x_j - x_k}, \tag{2.3}$$

The secret, that is distributed is the free coefficient of the polynomial. Considering that this secret can be recovered, when computing the interpolation for $x = 0$. For given t-number of values to find $f(0)$, the Lagrange interpolation formula can be simplified as

$$f(\kappa) = \sum_{j=0}^{t-1} y_j \prod_{\substack{m=0 \\ m \neq j}}^{t-1} \frac{x_m}{x_m - x_j}. \tag{2.4}$$

The protocol can also be modified to Elliptic Curves (EC) by generating the public key as a point on the EC by computing the $G \cdot sk$, where sk is chosen scalar at random ranging to the order of GF_q, and by choosing the κ again as a random point on the EC.

In the implementation of the ShSSAKA demonstrative application we use the EC modification.

2.4 Paillier scheme

The Paillier cryptosystem [32] is a probabilistic asymmetric algorithm for public-key cryptography first proposed by Pascal Paillier in 1999 in his paper "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". This paper offers a trapdoor mechanism from the family of trapdoors in the discrete logarithm technique based on composite residuosity classes.

This scheme has an additive homomorphic property, i.e., allows adding two ciphertexts without corrupting the result. The computation behaves just as if the corresponding plaintexts are added (without the need of decryption).

However, a drawback of the Paillier cryptosystem is its efficiency which is a common issue for any scheme based on homomorphic encryption. Several optimisation propositions has already been made. For instance, Paillier himself proposed a variant, namely *Scheme 3*, of the original proposal, namely *Scheme 1*, which uses abound α on exponentiation power size to speed up the computation of the encryption in particular. Another improving method can be the use of pre-computation of some values, for instance, the exponentiation of either the message g^m or the noise r^n as suggested in the article [17].

The use-cases of the Paillier cryptosystem are, for instance, in electronic voting [2], machine learning on encrypted data [6], and can be also used in other algorithms that demand high confidentiality even during processing between mutually authenticated parties because of other features such as the *self-binding ability* (ability to change one ciphertext into another without changing the content of the decryption) [33].

2.4.1 Paillier *Scheme 1*

In *Scheme 1* are the *public key pk* and *secret key sk* generated by the following algorithm. First are chosen two large prime numbers p and q at random such that they are independent of each other and that $\text{GCD}(pq, (p-1)(q-1)) = 1$ (this property is assured if both primes are of equal length). Than is computed the parameter $n = pq$ and parameter λ as $\text{LCM}(p-1, q-1)$ followed by the generation of parameter $g \in \mathbb{Z}^*_{n^2}$ at random. At this point must be ensured that n divides the g by checking the existence of modular multiplicative inverse μ as

$$\mu = (L(g^\lambda \mod n^2))^{-1} \mod n, \tag{2.5}$$

where function $L(x) = \frac{x-1}{n}$ where the notation $\frac{a}{b}$ denotes the quotient of a divided by b. The public key pk is than the ordered set of parameters n and g and the secret key sk is the ordered set of parameters λ and μ.

The **encryption** of a message m is computed as

$$c = g^m \cdot r^n \mod n^2, \tag{2.6}$$

where m must be in range $\langle 0, n)$ and parameter r is a generated number from interval $(0, n)$ at random. The **decryption** process is essentially exponentiation modulo n^2. The ciphertext is decrypted as

$$m = L(c^\lambda \mod n^2) \cdot \mu \mod n. \tag{2.7}$$

2.4.2 Paillier *Scheme 3*

In *Scheme 3* are the *public key pk* and *secret key sk* generated similarly to the *Scheme 1* only with few modifications. The n and λ parameters are computed the same as in the case of the *Scheme 1*. Afterwards is chosen the α as the divisor of the λ parameter with the restriction of $1 \leq \alpha \leq \lambda$. The parameter g is again generated the same as in the case of the *Scheme 1* from $\mathbb{Z}^*_{n^2}$ at random but with the restriction that g must be of order αn.

Another change comes in the **encryption** process of a message m and is computed as

$$c = g^m \cdot (g^n)^r \mod n^2, \tag{2.8}$$

where m must be in range $\langle 0, n)$ and parameter r is a generated number from interval $(0, \alpha)$ at random. The **decryption** process uses parameter α instead of the parameter λ as shown in following Equation 2.9.

$$m = L(c^\alpha \mod n^2) \cdot \mu \mod n. \tag{2.9}$$

The differences between the *Scheme 1* and *Scheme 3* are depicted and highlighted in Figure 2.5.

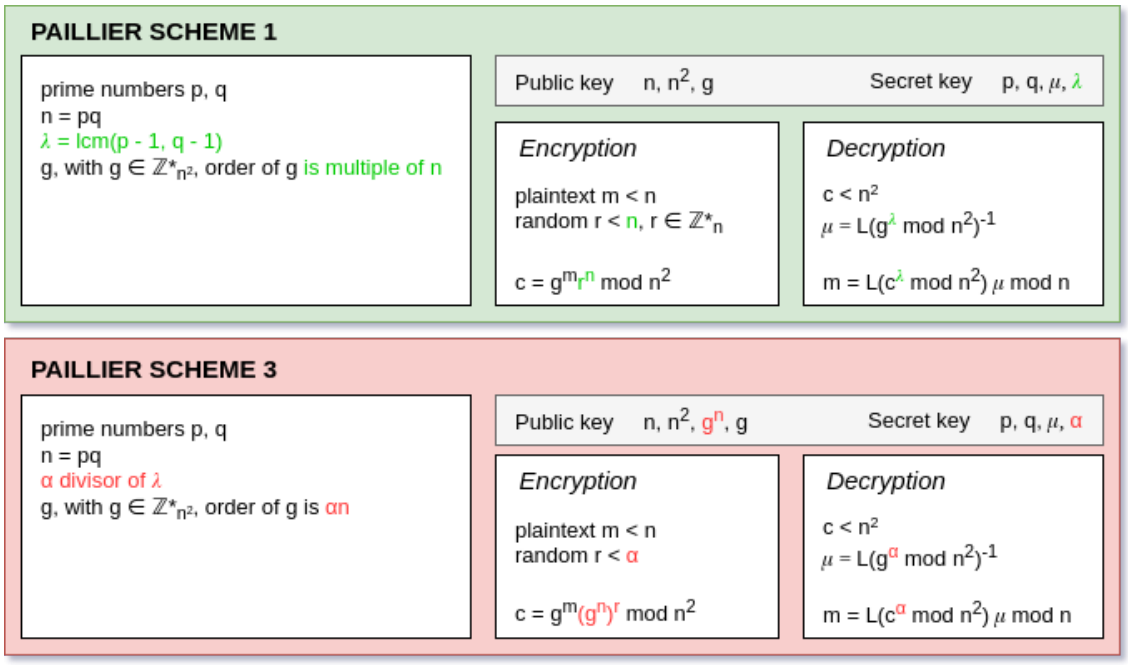

Fig. 2.5: Paillier *Scheme 1* and *Scheme 3* description with highlighted differences in green and red colours.

2.4.3 Homomorphic properties

Important features of the Paillier scheme in general are the *additive* and *multiplicative* **homomorphic properties**. Above mentioned abilities allow computing addition and multiplication of two ciphertexts. The computation behaves just as if it were operations over the corresponding plaintexts. The mathematical expression of these properties are as follows:

- *homomorphic addition* of plaintexts, where decryption of the product of two ciphers (or a product of a cipher and a generator g raised to the power of the plaintext) results in the addition of the two plaintexts:
  - $D(E(m_1, r_1) \cdot E(m_2, r_2) \mod n^2 = m_1 + m_2 \mod n$,
  - $D(E(m_1, r_1) \cdot g^{m_2} \mod n^2 = m_1 + m_2 \mod n$,
- *homomorphic multiplication* of plaintexts, where decryption of a ciphertext raised to the power of plaintext results to the multiplication of the two plaintexts:
  - $D(E(m_1, r_1)^{m_2} \mod n^2 = m_1 m_2 \mod n$,
  - $D(E(m_2, r_2)^{m_1} \mod n^2 = m_1 m_2 \mod n$,
  - more generally, where k is a constant: $D(E(m_1 m r_1)^k \mod n^2 = k m_1 \mod n$.

However, there is no known way to compute an encryption of the product of m_1 and

m_2 without knowing the private key sk. It is because of the Paillier characteristic of the encryption of the two messages.

2.5 Secret Sharing-based Authentication Key Agreement

The Secret Sharing-based Authentication Key Agreement (SSAKA) scheme is based on the AKA protocol described in Section 2.2 extended with the secret sharing property of the Shamir's Secret Sharing protocol.

The deployed entities in this protocol are of four types:

- `Server` and `Client`, which are the same entities as described in Section 2.2,
- `Device`, which is an additional device with usually less computational power than the *client's* primary device, such as the microcontrollers, wearables etc. and are involved in the authentication process to strengthen the security, and
- `Party` that specifies a device, its password, PIN or Public Key Infrastructure element, used in the protocol.

The basic AKA protocol is also for the use of SSAKA extended by three new algorithms to modify it for working in the multi-device environment. The new algorithms are

- the `SSAKA Client AddShare` algorithm that generates a new client's secret key, where each involved secondary device knows only a share of the client's secret key,
- the `SSAKA Client RevShare` algorithm that allows revoking of a device's share of the secret from the client's key, and
- the `SSAKA Device Proof` algorithm that computes the proof of knowledge of each share of the secret gained from involved devices.

The scheme of the SSAKA protocol is depicted in Figure 2.6.

The algorithm *Client AddShare* is running between all of the desired devices, client and server and as it was stated before. In this algorithm is the clients secret key used during the authentication sk_c computed as $sk_c = sk_0 + \sum_{i=1}^{DEV} sk_i$, where the sk_0 is the clients device secret key and the $sk_1, \ldots, sk_{DEV}$ are the secret keys of the involved devices. All of the keys are generated in a standard way as a at random generated sk_i and $pk_i = g^{sk_i}$.

Corresponding pk_c, that is securely stored on the server is computed as follows

$$pk_c = pk_c \cdot \bar{pk_c}, \tag{2.10}$$

where the $\bar{pk_c} = \prod_{i=1}^{NEW} pk_i$ for all new devices, that are supposed to be added to the set of possible devices for the future authentication.

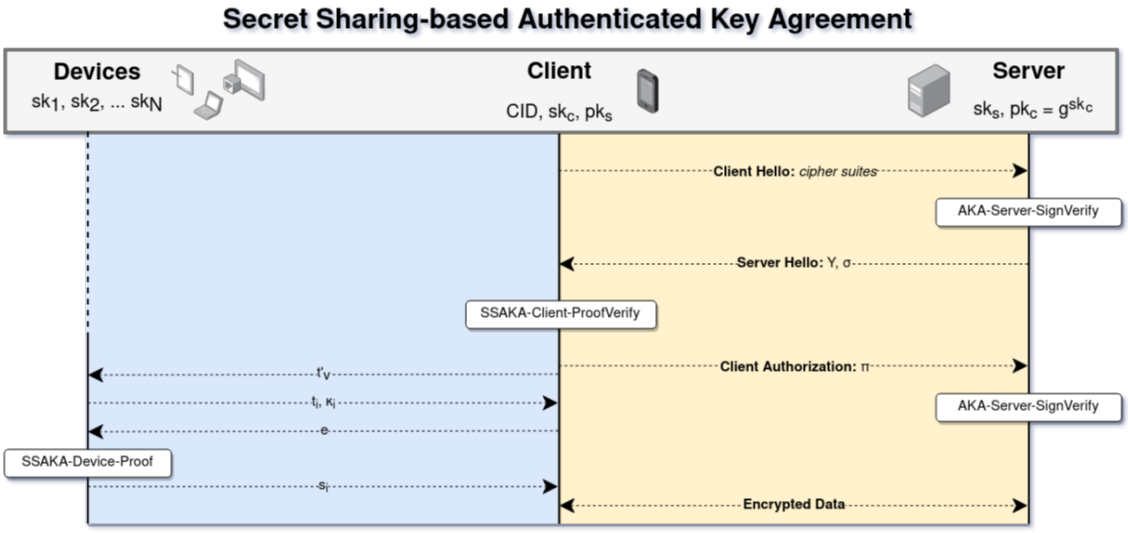

Fig. 2.6: SSAKA protocol scheme [9].

The algorithm *Client RevShare* revokes the public and the secret key of a set of devices, that are no longer wanted for authentication. The above mentioned equation for the algorithm *Client AddShare* is only modified for the use of revoking the public keys as

$$pk_c = pk_c \cdot \bar{pk}_c^{-1}, \tag{2.11}$$

where the parameter $\bar{pk}_c = \prod_{i=1}^{REV} pk_i$ for all devices for revocation.

The algorithm SSAKA Client ProofVerify is slightly modified from the basic AKA implementation and it consists of three phases:

1. In Phase 1, client computes the parameter $t'_s = g^{s_s} \cdot pk_s^{e_s}$, where the parameters s_s and e_s are stored in the servers σ signature. Then the algorithm continues with the computation of the hash of the message Y and of the computed t'_s. The computed hash then must be equal with the parameter e_s, otherwise the algorithm exits. If the equation statement is true, then the parameter t'_s is sent to the first part of the *DeviceProof* algorithm, running on the Devices. That generates and computes the random challenge t_i and a part of the session key κ_i.
2. In Phase 2, the client receives the number of shares in need of distribution. The client computes its session key κ and creates the hash of the message Y, commitment t and session key κ and sends this hash back to all devices, to the second part of the *DeviceProof* algorithm, where does the devices computes the proof of knowledge s_i of each desired device client share and sends them to the third part of the *Client ProofVerify* algorithm.

3. In Phase 3, the client puts all of the computed device proofs with the proof of knowledge of the client together and sends the "collective proof of knowledge" to the server.

The rest of the protocol runs as in the case of the Authentication Key Agreement scheme.

3 Used libraries

This chapter introduces the two libraries and one repository, that are used in the implementation. The first library is `OpenSSL`, which is an open-source implementation of some of the cartographic functions such as AES, SHA-2 or RSA and of the *Secure Sockets Layer* (SSL) and *Transport Layer Security* (TLS) protocols used for the secure communication over the internet. The second library is the *GIMP ToolKit* (GTK) widget, that is used for the application GUI implementation. The used repository is Dave Gamble's cJSON available on GitHub[1].

Both libraries are implemented in ANSI C and both bear certain advantages. The OpenSSL library can be used within the OS itself through the command line and PowerShell in the case of Microsoft Windows or Terminal in the case of Linux or MacOS. The GTK have the ability of Language Bindings, which means, that it can be connected to any programming language (if correctly set).

3.1 OpenSSL

OpenSSL [29] is a robust toolkit library for cryptographic and secure communication purposes over the computer network. The library is written in ANSI C and implements basic cryptographic functions and other supportive utilities such as the library for mathematical operations over big numbers. OpenSSL also contains the open-source implementation of the protocols Secure Sockets Layer (SSL) and Transport Layer Security (TLS) and is licensed under the permissive Apache License 2.0 that allows both free and commercial use. Some of the implemented algorithms that are supported by OpenSSL library are:

- **symmetric cryptography**:
  - Advanced Encryption Standard (AES),
  - Data Encryption Standard (DES, also in variant Triple DES),
  - Rivest Cipher 4 (RC4),
  - Blowfish, etc.
- **asymmetric cryptography**:
  - Rivest-Shamir-Adleman algorithm (RSA),
  - Digital Signature Algorithm (DSA),
  - Elliptic Curves (EC), etc.
- **hash functions**:
  - Secure Hash Algorithm version 1 (*unsecure*), 2 and 3 (SHA-1, SHA-2 and SHA-3),

[1] Dave Gamble's cJSON GitHub repository page https://github.com/DaveGamble/cJSON

  - Message-Digest algorithm version 2, 4 and 5 (*unsecure*, MD2, MD4 and MD5)
  - Whirpool, etc.

OpenSSL is available for most Unix-like Operation System such as Linux OS, MacOS), and is available even for Microsoft Windows the Berkeley Software (or Standard) Distribution OS (BDS) [28]. The official page of the OpenSSL library the developers also maintain a blog with the news and the list of known vulnerabilities of the library. The last stable build of this library is version 3.0.0 (to the 29th of November 2021).

3.2 GTK

GIMP ToolKit (GTK) [43] is an object-oriented widget toolkit for building the GUI managed by GNOME. It is a free and open-source cross-platform licensed under the *GNU Lesser General Public License* which allows use in both free and proprietary software. The main advantage of this tool widget is its ability of Language Binding. This property enables the use of multiple programming languages such as C++, JavaScript, Pearl, Python or Rust, though the widget itself was implemented in ANSI C [43].

The GTK project is dependable on multiple architectures. The main libraries the GTK is built upon are:

- **GLib**, which is a low-level core library providing the non-graphical functionality and provides the common GTK data types, main loop implementation, and a large set of utility functions for strings and general portability across different platforms,
- **Pango**, which is a library for international layout and text rendering,
- **Cairo**, which is a library for 2D graphics with support for multiple output devices,
- **GdkPixbuf**, which is a library for loading the graphical assets of the application such as icons, and
- **ATK**, which is a library providing the interface accessibility [42].

The GTK community also maintain good and broad documentation, which is accessible through the project official site. The GTK is available for all Operation System (Windows, Linux and MacOS), and the latest stable release is version 4.4.1 (to the 29th of November 2021).

3.3 cJSON

The cJSON repository by Dave Gamble is presented as an ultra-lightweight JSON parser implemented in ANSI C. The core of the code consists of one source code with a corresponding header file, which is used within the code and is completely independent.

This repository allows easier parsing of the JSON file including its creation, search within and work with the created cJSON object containing the JSON values.

The repository contains also some other useful implemented functions, such as string or number values comparators or encoders from/to pointer values. These procedures are implemented in other source file named `cJSON-Utils.c` with corresponding header file.

The repository can be installed by compiling the code by the directions written in the `README.md` file on the GitHub repository page. For the compilation is needed the *cmake* tool. If is this repository installed, then is this library included with the `#include <cjson/cJSON>` command.

The application implemented in this work is used only the standard cJSON functions for JSON file parsing so there is no need to install this repository since it's included in the source code.

4 Proposal of ShSSAKA protocol

The proposed extension of the SSAKA protocol [9] is focused on the fact that the user needs all of the registered devices to authenticate. The proposed scheme requires that t out of n registered devices collaborate to recover the secret key. To do so, we use the SSS and the *Paillier cryptographic scheme* to provide a secure distribution of the client authentication secret key computed as $\sum_{i=1}^{N} sk_i$, where sk_i are secret keys of devices and the client. The created share is then used as a device N secret key.

The polynomial used during the distribution is composed of at random generated values $d_{i,t}$, where i is the number of the device and t is the threshold value equal to the degree of the polynomial. For N devices is the polynomial stated as

$$f(x) = (d_{1,1} + \cdots + d_{N,t})x^t + (d_{1,2} + \cdots + d_{N,2})x^{t-1} + \cdots + \\ + (d_{N,1} + \cdots + d_{N,t})x + \sum_{i=1}^{N} \kappa_i, \tag{4.1}$$

where the $\sum_{i=1}^{N} \kappa_i$ is the sum of secret keys of the devices including the client. The $d_{1,t}, \ldots d_{N,t}$ sums and the κ_i sum are achieved by the sum of the computed $d_{i,1}x^t + \cdots + d_{i,t}x + \kappa_i$ polynomial, where each value x (in case of this protocol it is equal to the public key of the device to which is the share $f(x)$ computed to) of such "sub" polynomial is encrypted by Paillier scheme. The encryption than provides the ability to carry out parts of the authentication sk computation on each device, so none of the secret keys leaves the corresponding device.

The whole process of the secret distribution is carried out in the following step for the computation of each $f(x_j)$ for $j = 1, \ldots, n$. Let h be equal to $j + 1$:

1. D_h generates random value $r_{j,h}$ and compute $\chi_{j,h} = Enc(x_j, r_h)$,
2. D_h generates random value $v_{j,h}$ and compute

$$c_h = \chi_{j,h}^{x_j^{t-2} * d_{t-1}^{(h)}} * \chi_{j,h}^{x_j^{t-3} * d_{t-2}^{(h)}} * \cdots * \chi_{j,h}^{d_1^{(h)}} * Enc(k_j, v_{j,h})$$

   ,
3. if $h = j + 1$, then D_h sends c_h to D_{h+1},
4. if $h \neq j$, then set $h = h + 1 \mod n$ and go to Step (1)
5. if $h = j$, then D_j computes

$$f(x_j) = Dec(c_{j-1}) + d_{t-1}^{(j)} x_j^{t-1} + \cdots + d_1^{(j)} x_j + k_j$$

Above stated "computation circle" is depicted in Figure 4.1.

Once the key-pairs of all devices are computed, than can be created the client's authentication key with the use of the interpolation equation stated in Equation 2.4.

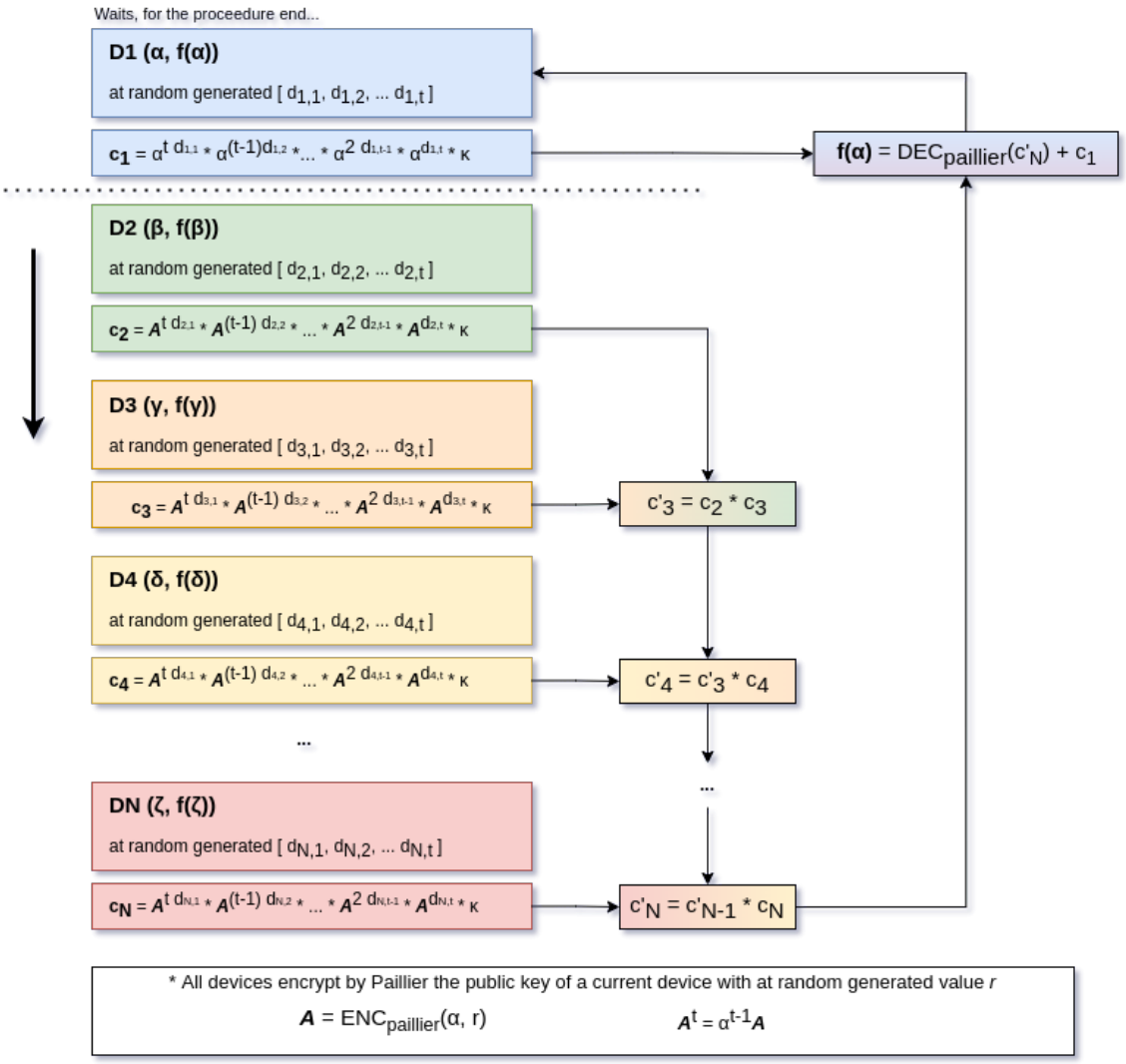

Fig. 4.1: Scheme of ShSSAKA private key computation cycle for each involved device.

The interpolation is used during the ShSSAKA Client ProofVerify part of the algorithm during the computation of the signature s_i of each involved device. The signature s_i is than computed as follows

$$s_i = r_i - e_c \cdot \mathcal{I}(sk_i) \mod q, \tag{4.2}$$

where the $\mathcal{I}(x)$ is a computation of one part of the $\sum_{j=0}^{k-1} y_j \mathcal{L}_j(0)$ of the simplified Lagrange interpolation stated in 2.4, where the y_j is the devices secret key sk_i.

To proof the `Completeness` of the proposed algorithm we consider the verification of the client's hash $e_c = \mathcal{H}(Y, t, \kappa)$ with servers reconstructed hash $e_s = \mathcal{H}(Y, t', \kappa)$. In order for the stated hashes, the values t and t' must equal thus the t value must be correctly reconstructed since the κ values are computed from the reconstructed

t-values of server and client (where client's t value is composed of the t values of devices).

$$\begin{aligned} t' &= g^{s_c} \cdot pk_c^{e_c} = g^{r_0 - e_c \cdot sk_0 + \cdot + r_i - e_c \cdot sk_i} \cdot g^{e_c \cdot sk_0 + \cdots + e_c \cdot sk_i} \\ &= g^{\sum r_i - e_c \cdot \sum sk_i} \cdot g^{e_c \cdot \sum sk_i} = g^{\sum r_i - e_c \cdot \sum sk_i + e_c \cdot \sum sk_i} \\ &= g^{\sum r_i} = t \end{aligned} \tag{4.3}$$

The κ_i values are computed from the reconstructions of the t and t_s values, then if the control hashes equal then the server κ and clients κ must equal as well. The above described code is implemented and published in the GitHub repository named "Simulator-for-ShSSAKA"[1].

[1] Simulator-for-ShSSAKA repository on GitHub https://github.com/Norted/Simulator-for-ShSSAKA

5 Implementation

In this chapter, the whole implementation of the protocol ShSSAKA proposed in this Master thesis is described.

Section 5.2 states the development of the application. Following, Section 5.1 explains the installation of the application and its startup. In the second section the functionality of the application is described and in the last section are presented the achieved experimental results.

The application demonstrates the ShSSAKA functionality and is developed on Ubuntu (Linux OS). Chosen programming language is the C language for its compilation speed and support on many platforms. For the Graphical User Interface was used the GTK widget toolkit, which is a C widget library. Moreover, GTK is able of multiple other Language Bindings such as C++, JavaScript, Python3.

For the development was used the Microsoft Visual Studio Code editor (Version 1.67.2), with the **C/C++ Extension Pack** (Version 1.1.0) developed by *Microsoft* [25]. This package contains the following list of extensions:

- *C/C++* by Microsoft (Version 1.7.1) for IntelliSense, debugging and code browsing,
- *C/C++ Themes* by Microsoft (Version 1.0.0) for *User Interface* (UI) themes,
- *CMake* by twxs (Version 0.0.17) for CMake language support,
- *CMake Tools* by Microsoft (Version 1.9.2) for extended CMake support in the VS Code,
- *Remote Development Extension Pack* by Microsoft (Version 0.21.0), that allows to open any folder in a container, remote machine or in *Windows Subsystem for Linux* (WSL),
- *GitHub Pull Requests and Issues* by GitHub (Version 0.32.0) for revision and management of the pull requests and issues on the GitHub platform,
- *Visual Studio Codespaces* by Microsoft (Version 1.0.3076), that provides cloud-hosted development environment,
- *LiveShare Extension Pack* by Microsoft (Version 0.4.0) for live collaboration,
- *Doxygen Documentation Generator* by Christoph Schlosser (Version 1.3.2), that simplifies the generation of the Doxygen Documentation, and
- *Better C++ Syntax* by Jeff Hykin (Version 1.15.10) for better C++ syntax visualisation.

All the above stated extensions are available in the Visual Studio Marketplace for free. They can be downloaded and installed separately or as all-in-one package (under the name "C/C++ Extension Pack", as it is mentioned above). The installation of the extensions can be made through the *Visual Studio Code Quick Open*. This "code" can be opened inside the VS Code with keyboard shortcut `Ctrl+P` by en-

tering the `ext install <author>.<package-name>` command which can be found on the official page of the extension in the Visual Studio Marketplace or through VS Code build-in module *Extensions* which is in default shown on the sidebar of the VS Code application (if not, than it can be accessed through `View > Extensions` or by keyboard shortcut `Ctrl+Shift+X`).

The application is implemented in C due to its speed and complex support on all thinkable platforms. The source code of the proposed protocol is available on the GitHub platform[1] and is published under the MIT license.

5.1 Installation

To run the implemented application "Simulator-for-ShSSAKA" must be first installed all the dependencies of the GTK library (if it is desired to use the GUI), the OpenSSL library (is the only library that needs to be installed in order to use the library) and the cJSON repository (does not need to be installed since the library is included in the code). Both the GTK[2] and OpenSSL[3] libraries have documentation regarding the installation and building of the libraries on their official websites. In case of the cJSON[4] are the instructions in the `README.md` file cloned within the repository.

The application can be built with the `Makefile` used with the GNU `make` tool. The `make` commands for Windows, Linux OS and MacOS respectively are shown in Listing 5.1.

Listing 5.1: Installation of the application Simulator-for-SSAKA

```
1  // Windows make command -- run cmd.exe as admin:
2  PS C:\> make --file Makefile
3
4  // Linux OS and MacOS make command:
5  user@pc:~$ cd /path/to/code/dir
6  user@pc:~/path/to/code/dir$ sudo ./Linux-install.sh
```

5.2 Application development

The main file `GUI.c` of the implemented application is situated in the `src` directory. In this file, the whole GUI environment is implemented. The environment im-

[1] Simulator-for-ShSSAKA repository on GitHub https://github.com/Norted/Simulator-for-ShSSAKA

[2] GTK toolkit installation documentation page https://www.gtk.org/docs/installations/

[3] OpenSSL library installation documentation page https://www.openssl.org/source/

[4] Dave Gamble's cJSON `README.md` file https://github.com/DaveGamble/cJSON/README.md

plements mainly button elements to provide the step-by-step ShSSAKA algorithm functionality demonstration.

The implemented algorithm itself is implemented in the file `ShSSAKA.c` and is located in the `c_file` directory alongside with the implemented Paillier scheme in file `paillier_scheme.c` and supportive library file `support_functions.c` that is implementing functions necessary mathematical and other procedures for the correct function of the application. Furthermore are implemented the SSS with the Paillier modification (file `paishamir.c`) and Schnorr's signature with the EC modification (file `schnorrs_signature.c`).

Inside the code is also included the *c*JSON library (source code and header file, each in corresponding directory), that were downloaded from the cJSON repository on GitHub mentioned in Section 3.3.

Each of the above mentioned `*.c` file (except the `GUI.c` file) have implemented corresponding header file in the `headers` directory containing definitions of structures, global variables and function declarations. All of the common includes, structures and global variables are situated in the `globals.h` file. Here are also defined macros such as the `G_NUMOFDEVICES`, that defines the maximal number of devices used during the process or `G_POLYDEGREE` that defines the minimal number of devices to reconstruct the secret value.

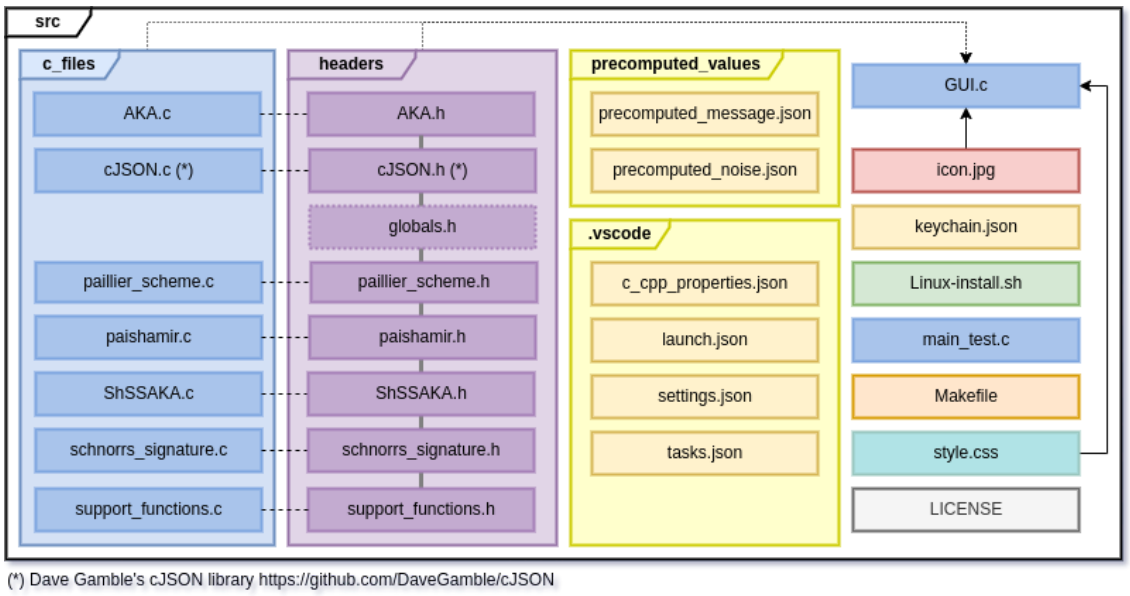

Fig. 5.1: Simplified diagram of the source code directory.

In the directory, `src` are also included `Makefile` for building the application, `icon.jpg`, that server as the application icon (only visible at Windows OS) and

`style.css` file which allows to personalise the default GUI GTK elements and applied in the application through a function shown in Listing 5.2. In the `src` directory is also situated the `Linux-install.sh` file for the installation of the application together with the required external libraries such as the OpenSSL. In Figure 5.1 is shown simple directory diagram for better orientation in the code structure with colour-coded structure. Blue colour states for *C language files*, violet colour for *header files*, red colour for *JPG files*, orange colour for *Makefile*, yellow color for JSON files (used for configurations, pre-computed and saved values), green files for *Shell Script* (installation script) and turquoise colour for *CSS file*.

Listing 5.2: Function loading the `style.css` file

```
1   void setCSS () {
2     GtkCssProvider *provider;
3     GdkDisplay *display;
4     GdkScreen *screen;
5
6     provider = gtk_css_provider_new ();
7     display = gdk_display_get_default ();
8     screen = gdk_display_get_default_screen (display);
9
10    gtk_style_context_add_provider_for_screen (screen,
11          \acs{gtk}_STYLE_PROVIDER (provider),
12          \acs{gtk}_STYLE_PROVIDER_PRIORITY_APPLICATION);
13
14    const gchar *cssFile = "style.css";
15    GError *error = 0;
16
17    gtk_css_provider_load_from_file(provider,
18          g_file_new_for_path(cssFile), &error);
19    g_object_unref (provider);
20  }
```

5.3 Application visualisation

The main window of the application is divided into three parts:

- the **Spin-Button** "Message" providing the Y parameter instead of the concatenation of the chosen cipher suite,
- the **Button-Box** containing all of the buttons with bindings to the separate parts of the ShSSAKA algorithm, and the

- **Label** serving as the Console for listing the information about the state of the protocol.

In the Button-Box, there are several buttons that perform the algorithm. There are a few elements enabled at the start of the application:

- the *Message Spin-Box* substituting the agreed cypher suite,
- *"Add" Spin-Box*, that states, how many devices will be added with ShSSAKA ClientAddShare button,
- the *"Revoke list" Entry box*, in which can be stated devices, that should be removed from the process, and
- *Setup button* initialising the parameters and keys used in the ShSSAKA algorithm.

The first two buttons provide the ShSSAKA AddShare and ShSSAKA RevShare parts of the algorithm described in the Section 2.5. Below is button providing the ShSSAKA Server SignVerify, that performs the verification between client-server and devices-client.

Between the last two buttons and the ShSSAKA Server SignVerify is situated the *Devices list Entry box* and two *check-boxes* labelled as "Pre-computation". In the entry box can be stated, how many devices will be used during the verification (and during the secret reconstruction), other wise all included devices are used. The *Pre-computation check-boxes* define whether the pre-computation of the Paillier scheme values will be used during the procedure.

And last two buttons are the *Keys* and *Parameters Buttons*. Both open message dialogue window that lists generated keys or the initialised parameters. Parameters shown in the dialogue are the order of the group q, chosen generator g and the loaded message parameter Y.

Entities, that are as default implemented are server, client and three secondary devices. The number of devices can be changed in both in the `ShSSAKA.h` file or through the ShSSAKA ClientAddShare. The described GUI is depicted in Figure 5.2.

5.4 Experimental results

In this chapter, we present the benchmarks and the comparison results from the implemented protocol. The experimental results were run on a PC and a RPi. In particular, we used an Intel(R) Core(TM) i7-4510U CPU at 2.00 GHz with 4 cores and Ubuntu 64-bit v20.04.4 LTS operating system and a Raspberry Pi 3 Model B+

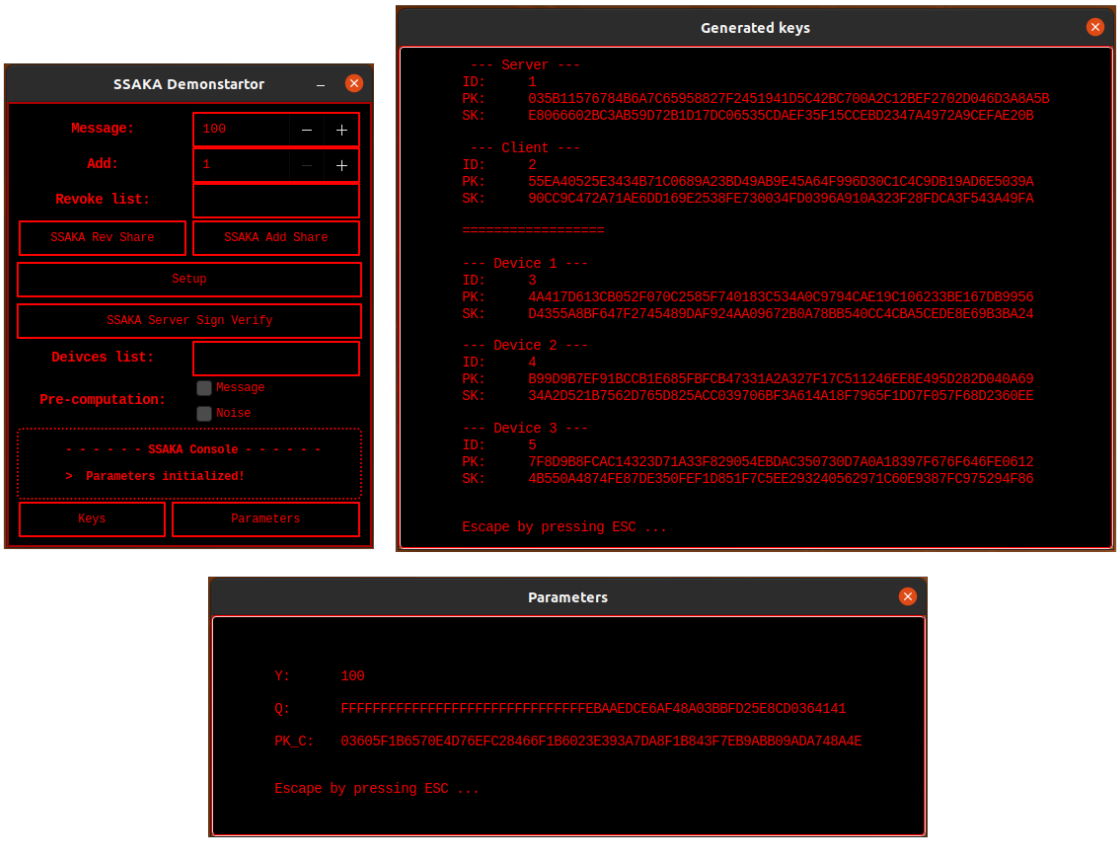

Fig. 5.2: Simulator-for-ShSSAKA application Graphical User Interface

[5]. The compiler version is GCC v9.3.0. The PC results are marked as "PC" and the results gained on RPi3 are marked as "RPi".

In Section 5.4.1, we present experimental results of several optimised implementations of the Paillier cryptosystem. Since the speed of the "plain" cryptosystem is insufficient for the needs of current protocols, we have modified the cryptosystem based on the [17] in order to be faster. We have especially concentrated on the encryption procedure since it is used more often in the ShSSAKA. The implemented code is publicly available on GitHub under the MIT license. Experimental results were published in [37] within the 28th EEICT Student Conference.

In Section 5.4.2, we present experimental results of the implemented library of the proposed ShSSAKA protocol. Within this section, we give our main focus to the change of the average time over changing the total number of devices n or changing the degree of the polynomial t.

[5]Raspberry Pi 3 Model B+ official page https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/

5.4.1 Paillier scheme experimental results

Experimental results for implemented Paillier scheme were gained only on the above-defined PC for a DSA modulus n of 2048-bit long. Encryption results of one message were averaged over 100 iterations and 10 different message values were considered.

The application used to obtain the experimental results of the Paillier scheme is publicly available on GitHub repository[6] which was implemented within the 28th Conference STUDENT EEICT 2022.

Scheme 1			
	Encryption $[ms]$	Decryption $[ms]$	Total $[ms]$
Plain	2.172961	1.915783	4.088744
Random	0.036213	2.084682	2.120895
Message	2.137328	1.913080	4.050408
Both	0.008832	2.060453	2.069285

Scheme 3			
	Encryption $[ms]$	Decryption $[ms]$	Total $[ms]$
Plain	2.530260	2.387416	4.917676
Random	0.096640	2.514155	2.610795
Message	2.414621	2.359763	4.774384
Both	0.025454	2.521757	2.547211

Tab. 5.1: Average time of encryption, decryption and total time in ms for 2048-bit length of n. "Plain" states for in plain mode, "random" for pre-computed noise values, "message" for pre-computed message values, and "both" for both pre-computations.

Table 5.1 depicts *Scheme 1* and *Scheme 2* performances averaged over 10 messages, respectively. Based on the gained results, the decryption speed is for all variants comparable. This is due to fact that the decryption operations are limited using the pre-computation optimisation. Regarding the encryption, *Scheme 1* is slightly quicker than *Scheme 3* despite expectations. This probably occurs because of the smaller size of the exponentiation base r in *Scheme 1* with respect to g^n in *Scheme 3*. The results were obtained with parameter `RANGE` sets to 100 where the pre-computed message file has size 64.9 kB for both schemes and the pre-computed noise files have size 64.3 and 125.9 kB for *Scheme 1* and *Scheme 3*, respectively.

Figure 5.3 depicts *Scheme 1* and *Scheme 3* computational cost of one message encryption averaged over 100 times iteration of the considered protocol. From the analyses, we can deduce that the most computationally intensive operation is noise

[6]EEICT_Paillier GitHub repository https://github.com/Norted/EEICT_Paillier

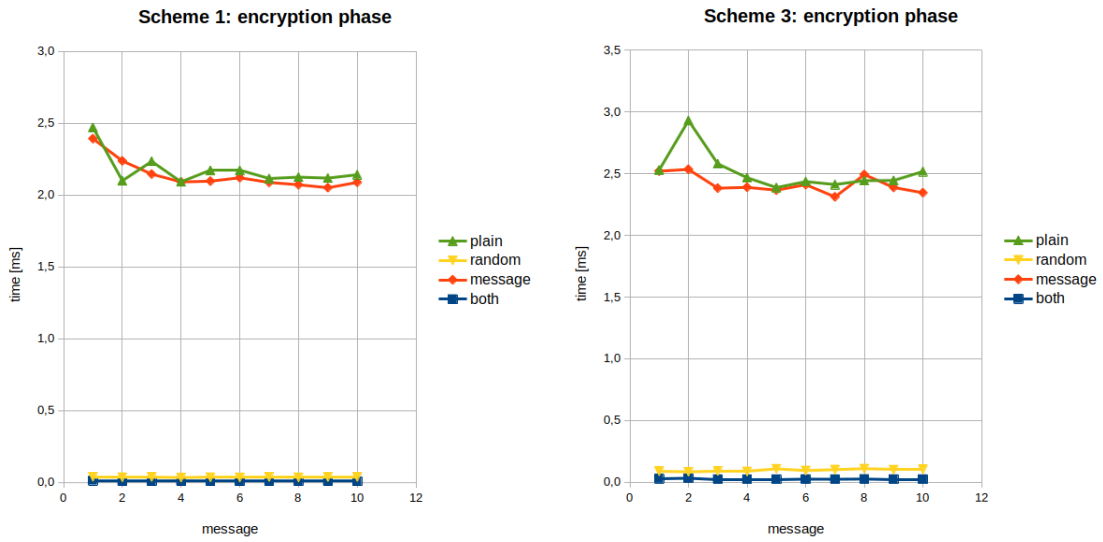

Fig. 5.3: Encryption overtime of both *Schemes 1 and 3* with their optimisations. "Plain" states for in plain mode, "random" for pre-computed noise values, "message" for pre-computed message values, and "both" for both pre-computations.

computation. This is the main reason for the drastic speedup when the noise pre-computation is applied to the process.

Optimisation technique	Scheme 1 saved time $[ms]$	Scheme 3 saved time $[ms]$
Noise pre-computation	3.286700	3.660420
μ pre-computation	2.544364	3.100727
Optimisation technique	Schemes 1 and 3 saved time $[ms]$	
Message pre-computation	0.039340	
n^2 pre-computation	0.039340	
g^n pre-computation	13.568727	

Tab. 5.2: Average time in *ms* saved during each optimisation of the considered Paillier schemes.

Table 5.2 contains the saved times in average over 10 protocol executions of the considered pre-computation technique. Note that μ and n^2 acceleration allow saving time during the whole process since they are applied multiple times in the protocols.

5.4.2 ShSSAKA experimental results

Experimental results for implemented ShSSAKA library were gained both on the above-defined PC and RPi. The procedure was always settled on 10 runs (the

number was chosen with respect to RPi performance).

The results were gained on EC named "Secp256k1" and n 1024-bits big. The results were achieved by changing the number of used devices n or by changing the polynomial degree t (i. e. how many devices are needed to reconstruct the client's secret). The list of devices used for the secret reconstruction was chosen at random from the generated set of devices. Also, the Paillier pre-computation of noise and message values was used during the process.

Table 5.3 shows the average time in *ms* that is needed to setup the key-chains, based on the number n of all devices used for the share distribution.

PC			
n	Devices [*ms*]	Server [*ms*]	Time [*ms*]
5	134.832	1.494	537.075
10	572.798	2.947	1074.883
15	1316.103	5.380	1742.135

RPi			
n	Devices [*ms*]	Server [*ms*]	Time [*ms*]
5	3198.331	13.949	20878.600
10	12689.495	26.456	26125.445
15	30068.082	43.671	42696.684

Tab. 5.3: Average time in *ms* that takes to setup the ShSSAKA key-chains and corresponding variables on PC and RPi based on the number of overall devices n. "Devices" stands for average time to generate all given devices (by global *currentNumOfDevices*), "server" stands for average time to generate the server key-chain and "time" stands for the average time of the whole setup procedure.

Table 5.4 shows the average time in *ms* to verify and authenticate the number of involved parties in the process by changing the parameter n (the number of overall devices) on the PC and RPi.

Based on these two tables we can state, that with the rising number of overall used devices (raising the n value), the time to set up the ShSSAKA protocol raises. The verification and authentication are constant no matter the number of used devices to reconstruct the client's secret. It is related to the fact that computing $f(\alpha)$ (where α is the private key of the device) is heavy since it uses the Paillier and encryption whereas computing s_i involves fast computation (since its only multiplications and inversion operations).

Table 5.3 also shows, that the time to generate server key-chain raises with the number of devices n. It is given by the necessity to compute more parts of the sum

in the Lagrange interpolation (for each device).

		PC			RPi		
	#	Auth [*ms*]	Ver [*ms*]	Time [*ms*]	Auth [*ms*]	Ver [*ms*]	Time [*ms*]
5 devices	2	1.080	5.063	6.143	19.031	91.187	110.218
	3	1.183	6.425	7.608	18.956	120.360	139.316
	4	1.119	7.833	8.952	20.436	135.001	155.437
10 devices	2	1.165	5.010	6.175	11.237	54.2413	65.478
	3	1.072	6.719	7.792	11.311	65.8393	77.150
	4	1.084	7.679	8.764	10.923	77.8033	88.726
	5	1.105	8.940	10.045	11.261	93.3406	104.601
	6	1.155	10.236	11.391	11.452	106.3357	117.787
	7	1.216	11.787	13.004	11.256	118.2103	129.466
	8	1.099	13.180	14.279	11.018	134.8705	145.888
	9	1.260	14.722	15.982	11.359	149.7626	161.121
15 devices	2	1.115	5.183	6.298	12.388	59.231	71.619
	3	1.086	6.607	7.693	12.191	72.192	84.383
	4	1.070	7.507	8.577	12.199	86.545	98.744
	5	1.073	9.627	10.700	12.152	98.902	111.054
	6	1.212	10.782	11.994	11.819	114.493	126.312
	7	1.081	11.877	12.958	12.377	131.901	144.278
	8	1.066	13.011	14.078	12.394	141.784	154.178
	9	1.084	15.257	16.341	11.851	158.925	170.776
	10	1.244	15.964	17.208	12.404	176.937	189.341
	11	1.063	18.194	19.257	12.078	192.839	204.917
	12	1.086	19.364	20.450	12.054	205.419	217.472
	13	1.143	20.975	22.118	12.185	225.460	237.644
	14	1.095	23.100	24.195	11.828	240.571	252.399

Tab. 5.4: Average time in *ms* that takes to the authentication and verification phases of the ShSSAKA procedure on PC and RPi based on the overall number of devices n used for share distribution. "#" stands for the number of used devices, "Auth" stands for the average time of the authentication phase, "Ver" stands for the average time of the verification phase and "Time" stands for the average time of the whole procedure.

Table 5.5 shows the average time in *ms* taken to generate the key-chains and corresponding variables for the ShSSAKA protocol based on the raising polynomial degree t (i.e. the number of devices needed to reconstruct the client's secret). The number of devices n was fixed on number 15.

The time raises with the value of the polynomial degree t. It is given by the complexity of the computed polynomials and the number of heavy operations that need to be done in order to distribute the secret value. Another factor that elongates the computation cycle is the fact, that each device needs to wait for the previous device's result in order to continue computing (i.e. parallel processing is not possible). The time to set the server key-chain is constant in all of the cases (both on PC and RPi).

PC			
t	Devices $[ms]$	Server $[ms]$	Time $[ms]$
3	1512.297	5.381	1927.840
4	1696.351	5.398	2138.364
5	1867.251	5.419	2308.336
6	2053.460	5.438	2395.146
7	2235.280	5.431	2638.272
8	2423.738	5.471	2825.779
9	2604.093	5.391	3216.640
10	2780.702	5.450	3174.993

RPi			
t	Devices $[ms]$	Server $[ms]$	Time $[ms]$
3	33114.160	39.129	52997.142
4	39978.608	47.260	61979.679
5	44279.505	44.114	73563.325
6	95079.671	90.164	126338.344
7	103980.745	84.875	140226.446
8	117340.567	89.997	158898.715
9	125602.765	91.029	169098.090
10	135528.451	90.340	163223.281

Tab. 5.5: Average time in *ms* that takes to setup the ShSSAKA key-chains and corresponding variables on PC and RPi based on the polynomial degree t. "Devices" stands for average time to generate all given devices (by global *currentNumOfDevices*), "server" stands for average time to generate the server key-chain and "time" stands for the average time of the whole setup procedure.

Tables 5.6 and 5.7 show the average time taken to run the authentication and verification procedure also based on the number of devices used for the secret reconstruction. As can be seen, the time is constant for the authentication phase, but the time slightly raises for the verification phase with the polynomial degree t.

As can be seen from the tables, the verification time raises with the polynomial degree t. It is given by the number of operations that needs to be done in order to reconstruct the distributed secret of the client. The time of the authentication phase remains constant.

		PC			RPi		
	#	Auth [ms]	Ver [ms]	Time [ms]	Auth [ms]	Ver [ms]	Time [ms]
Degree 3	10	1.075	16.324	17.399	13.416	195.111	208.526
	11	1.082	17.761	18.843	13.426	212.667	226.094
	12	1.155	20.312	21.467	13.419	212.357	225.776
	13	1.120	21.102	22.222	12.029	212.869	224.898
	14	1.114	23.050	24.164	10.578	240.136	250.714
Degree 4	10	1.073	16.578	17.651	13.457	195.541	208.998
	11	1.075	17.796	18.871	13.461	212.631	226.092
	12	1.076	19.636	20.711	13.614	230.040	243.654
	13	1.104	21.431	22.535	13.132	247.291	260.424
	14	1.180	23.996	25.176	13.695	270.099	283.794
Degree 5	10	1.080	16.226	17.305	12.359	178.381	190.740
	11	1.059	18.062	19.120	12.403	194.733	207.136
	12	1.099	19.553	20.653	12.383	211.233	223.616
	13	1.153	21.304	22.456	12.418	228.086	240.504
	14	1.184	22.701	23.886	12.401	245.322	257.723
Degree 6	10	1.089	16.013	17.102	24.732	356.870	381.606
	11	1.225	17.835	19.060	24.750	389.417	414.167
	12	1.222	19.312	20.534	24.792	400.086	424.877
	13	1.126	20.736	21.862	21.985	409.101	431.086
	14	1.158	22.761	23.919	22.843	440.192	463.035
Degree 7	10	1.116	16.235	17.351	23.306	345.303	368.609
	11	1.097	18.150	19.247	23.330	368.040	391.370
	12	1.168	19.265	20.433	22.492	406.781	429.274
	13	1.106	21.729	22.835	23.326	441.871	465.196
	14	1.060	22.816	23.876	24.750	475.521	500.270

Tab. 5.6: Average time in ms that takes to the authentication and verification phases of the ShSSAKA procedure on PC and RPi based on the polynomial degree t (in this table degree 3 to 7). "#" stands for the number of used devices, "Auth" stands for the average time of the authentication phase, "Ver" stands for the average time of the verification phase and "Time" stands for the average time of the whole procedure.

		PC			RPi		
	#	Auth [ms]	Ver [ms]	Time [ms]	Auth [ms]	Ver [ms]	Time [ms]
Degree 8	10	1.147	16.753	17.900	24.711	357.192	381.904
	11	1.087	17.510	18.597	24.718	389.650	414.367
	12	1.062	19.793	20.855	24.782	422.523	447.305
	13	1.124	21.282	22.406	24.770	456.907	481.677
	14	1.125	23.017	24.142	24.762	491.155	515.917
Degree 9	10	1.070	16.664	17.733	24.878	359.426	384.304
	11	1.167	18.161	19.327	24.884	392.332	417.216
	12	1.116	20.651	21.767	24.895	425.588	450.482
	13	1.069	21.076	22.146	24.940	459.358	484.297
	14	1.136	22.776	23.912	24.902	493.750	518.653
Degree 10	10	1.138	16.248	17.387	24.984	360.286	385.270
	11	1.147	18.393	19.540	24.928	392.820	417.748
	12	1.092	19.705	20.796	24.971	426.130	451.102
	13	1.152	21.243	22.395	24.935	460.221	485.155
	14	1.085	23.310	24.395	24.978	494.655	519.633

Tab. 5.7: Average time in *ms* that takes to the authentication and verification phases of the ShSSAKA procedure on PC and RPi based on the polynomial degree t (in this table degree 8 to 10). "#" stands for the number of used devices, "Auth" stands for the average time of the authentication phase, "Ver" stands for the average time of the verification phase and "Time" stands for the average time of the whole procedure.

Conclusion

This Master thesis proposes an extension of SSAKA protocol [9] by combining it with Shamir's Secret Sharing scheme. This allows multi-device and multi-factor authentication during the session key agreement where not all involved devices but only a threshold of them is needed to authenticate. The use of the SSS protocol increases the security of the proposed protocol since the client's secret is distributed between more parties. These parties can be computationally constrained devices such as embedded microcontrollers, wearables, smartphones and many others. The possible attacker would have to acquire more than just one secret key of any involved device to corrupt the system. Our proposal is based on the homomorphic property of the Paillier cryptosystem in the computation of the secret keys of the sharing parties.

Cryptographic library of the proposed ShSSAKA scheme is implemented in C language and also implements a Graphical User Interface. The GUI can be seen as a form of a use-case demo, that also shows the functionality of the protocol.

The implementation uses the OpenSSL and GTK widget libraries. The OpenSSL library's main purpose is to provide open-source implementation of SSL and TLS protocols. Moreover, it implements the Big Numbers data type that also includes the implementation of basic mathematical operations such as addition, multiplication, division and modulo. This library also implements a use-friendly EC library that is used in the implementation.

As a part of this work, a bench-marking of the implemented ShSSAKA protocol library and the optimised Paillier scheme was done. The main focus in the ShSSAKA was given to the change of the average time over changing the number of devices n or the degree of the polynomial t. The experimental results were run on a PC and a RPi3 (for the specifications see Section 5.4).

Gained results over the ShSSAKA show, that the time of the verification procedure raises with both n and t. The average time of the authentication procedure remains constant in all cases.

The gained values on the Paillier scheme show, that the pre-computation of certain values can speed up the encryption drastically, especially if the noise value pre-computation is used. Other pre-computed values such as the message pre-computation or the n^2 value contribute only slightly to the acceleration. Experimental results of the Paillier scheme were presented at EEICT Student Conference 2022 [37].

Bibliography

[1] Mahmoud Ammar, Giovanni Russello, and Bruno Crispo. "Internet of Things: A survey on the security of IoT frameworks". In: *Journal of Information Security and Applications* 38 (2018). [cited 18-11-2021], pp. 8–27. ISSN: 2214-2126. DOI: https://doi.org/10.1016/j.jisa.2017.11.002. URL: https://www.sciencedirect.com/science/article/pii/S2214212617302934.

[2] Shifa Manaruliesya Anggriane, Surya Michrandi Nasution, and Fairuz Azmi. "Advanced e-voting system using Paillier homomorphic encryption algorithm". In: *2016 International Conference on Informatics and Computing (ICIC)*. [cited 17-05-2022]. IEEE. 2016, pp. 338–342.

[3] IEEE Standard Association. *IEEE standards activities in the internet of things (IOT)*. [online]. [cited 22-11-2021]. Mar. 2020. URL: https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/other/iot.pdf.

[4] Mihir Bellare and Oded Goldreich. "On defining proofs of knowledge". In: *Annual International Cryptology Conference*. [cited 16-05-2022]. Springer. 1992, pp. 390–420.

[5] Mihir Bellare, David Pointcheval, and Phillip Rogaway. "Authenticated Key Exchange Secure against Dictionary Attacks". In: *Advances in Cryptology — EUROCRYPT 2000*. Ed. by Bart Preneel. [cited 10-12-2021]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 139–155. ISBN: 978-3-540-45539-4.

[6] Raphael Bost et al. "Machine learning classification over encrypted data". In: *Cryptology ePrint Archive* (2014). [cited 17-05-2022].

[7] Ernest F. Brickell. "Some Ideal Secret Sharing Schemes". In: *Advances in Cryptology — EUROCRYPT '89*. Ed. by Jean-Jacques Quisquater and Joos Vandewalle. [cited 30-11-2021]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 468–475. ISBN: 978-3-540-46885-1.

[8] Microsoft Corporation. *Azure IoT – Internet of Things Platform | Microsoft Azure*. [online]. [cited 27-11-2021]. URL: https://azure.microsoft.com/en-us/overview/iot/.

[9] Petr Dzurenda et al. "Secret Sharing-Based Authenticated Key Agreement Protocol". In: *The 16th International Conference on Availability, Reliability and Security*. ARES 2021. [cited 30-11-2021]. New York, NY, USA and Vienna, Austria: Association for Computing Machinery, 2021. ISBN: 9781450390514. DOI: 10.1145/3465481.3470057. URL: https://doi.org/10.1145/3465481.3470057.

[10] Alexander S. Gillis. *What is internet of things (IoT)?* [online]. [cited 17-11-2021]. Aug. 2021. URL: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT.

[11] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. "The Knowledge Complexity of Interactive Proof Systems". In: *SIAM Journal on Computing* 18.1 (1989). [cited 10-12-2021], pp. 186–208. DOI: 10.1137/0218012. eprint: https://doi.org/10.1137/0218012. URL: https://doi.org/10.1137/0218012.

[12] Carsten Gregersen. *A Complete Guide to IoT Protocols & Standards In 2021.* [online]. [cited 27-11-2021]. Dec. 2020. URL: https://www.nabto.com/guide-iot-protocols-standards/.

[13] Jayavardhana Gubbi et al. "Internet of Things (IoT): A vision, architectural elements, and future directions". In: *Future Generation Computer Systems* 29.7 (2013). Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications — Big Data, Scalable Analytics, and Beyond. [cited 27-11-2021], pp. 1645–1660. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2013.01.010. URL: https://www.sciencedirect.com/science/article/pii/S0167739X13000241.

[14] Software Testing Help. *10 Best IoT Platforms To Watch Out In 2020.* [online]. [cited 27-11-2021]. Nov. 2021. URL: https://www.softwaretestinghelp.com/best-iot-platforms/.

[15] Amazon.com Inc. *AWS IoT Core.* [online]. [cited 27-11-2021]. 2021. URL: https://aws.amazon.com/iot-core/.

[16] Mitsuru Ito, Akira Saito, and Takao Nishizeki. "Secret sharing scheme realizing general access structure". eng. In: *Electronics & communications in Japan. Part 3, Fundamental electronic science* 72.9 (1989). [cited 30-11-2021], pp. 56–64. ISSN: 1042-0967.

[17] Christine Jost et al. *Encryption Performance Improvements of the Paillier Cryptosystem.* Cryptology ePrint Archive, Report 2015/864. [cited 17-05-2022]. 2015. URL: https://ia.cr/2015/864.

[18] Anders Karlsson, Masato Koashi, and Nobuyuki Imoto. "Quantum entanglement for secret sharing and secret splitting". In: *Phys. Rev. A* 59 (1 Jan. 1999). [cited 16-05-2022], pp. 162–168. DOI: 10.1103/PhysRevA.59.162. URL: https://link.aps.org/doi/10.1103/PhysRevA.59.162.

[19] Sjoerd Langkemper. *The Most Important Security Problems with IoT Devices.* [online]. [cited 28-11-2021]. URL: https://www.eurofins-cybersecurity.com/news/security-problems-iot-devices/.

[20] Laurie Law et al. “An Efficient Protocol for Authenticated Key Agreement”. In: *Designs, Codes and Cryptography* 28.2 (Mar. 2003). [cited 16-05-2022], pp. 119–134. ISSN: 1573-7586. DOI: 10.1023/A:1022595222606. URL: https://doi.org/10.1023/A:1022595222606.

[21] Libelium. *50 Sensor Applications for a Smarter World*. [online]. [cited 18-11-2021]. Sept. 2020. URL: https://www.libelium.com/libeliumworld/top_50_iot_sensor_applications_ranking/.

[22] Libelium. *Libelium " connecting sensors to the cloud: Powering the IOT revolution*. [online]. [cited 09-12-2021]. 2021. URL: https://www.libelium.com/.

[23] Google LLC. *Cloud IoT Core*. [online]. [cited 27-11-2021]. Nov. 2021. URL: https://cloud.google.com/iot-core?hl=cs.

[24] Google LLC. *Pricing | Cloud IoT Core*. [online]. [cited 27-11-2021]. Nov. 2021. URL: https://cloud.google.com/iot/pricing?hl=cs.

[25] Microsoft. *C/C++ Extension Pack - Visual Studio Marketplace*. [cited 10-12-2021]. URL: https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools-extension-pack.

[26] Zeyad Mohammad, Ahmad Abusukhon, and Thaer Abu Qattam. “A Survey of Authenticated Key Agreement Protocols for Securing IoT”. In: *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. [cited 1-12-2021]. 2019, pp. 425–430. DOI: 10.1109/JEEIT.2019.8717529.

[27] Tatsuaki Okamoto. “Provably Secure and Practical Identification Schemes and Corresponding Signature Schemes”. In: *Advances in Cryptology — CRYPTO' 92*. Ed. by Ernest F. Brickell. [cited 30-11-2021]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 31–53. ISBN: 978-3-540-48071-6.

[28] Inc. OpenSSL Foundation. *OpenSSL*. [online]. [cited 16-05-2022]. URL: https://www.openssl.org/docs/manmaster/man7/.

[29] Inc. OpenSSL Foundation. *OpenSSL – Cryptography and SSL/TSL Toolkit*. [online]. [cited 09-12-2021]. 2021. URL: https://www.openssl.org/.

[30] Oracle. *What is an IoT Gateway?* [online]. [cited 17-11-2021]. May 2017. URL: https://openautomationsoftware.com/open-automation-systems-blog/what-is-an-iot-gateway/.

[31] Oracle. *What is the Internet of Things (IoT)?* [online]. [cited 17-11-2021]. Nov. 2021. URL: https://www.oracle.com/cz/internet-of-things/what-is-iot/.

[32] Pascal Paillier. “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In: *Advances in Cryptology — EUROCRYPT ’99*. Ed. by Jacques Stern. [cited 10-12-2021]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-48910-8.

[33] Pascal Paillier. “Public-key cryptosystems based on composite degree residuosity classes”. In: *International conference on the theory and applications of cryptographic techniques*. [cited 1-12-2021]. Springer. 1999, pp. 223–238.

[34] G2 Bussiness Software Reviews. *AWS IoT Core reviews*. [online]. [cited 27-11-2021]. URL: https://www.g2.com/products/aws-iot-core/reviews.

[35] G2 Bussiness Software Reviews. *Azure Iot Central Reviews*. [online]. [cited 27-11-2021]. URL: https://www.g2.com/products/azure-iot-central/reviews.

[36] G2 Bussiness Software Reviews. *Google Cloud IoT Core reviews*. [online]. [cited 27-11-2021]. URL: https://www.g2.com/products/google-cloud-iot-core/reviews.

[37] Sara Ricci and Pavla Ryšavá. “Paillier Cryptosystem Optimisations for Homomorphic Computation”. In: Accepted in the 28th Conference STUDENT EEICT 2022. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií. 2022.

[38] C. P. Schnorr. “Efficient Identification and Signatures for Smart Cards”. In: *Advances in Cryptology — CRYPTO’ 89 Proceedings*. Ed. by Gilles Brassard. [cited 16-05-2022]. New York, NY: Springer New York, 1990, pp. 239–252. ISBN: 978-0-387-34805-6.

[39] C. P. Schnorr. “Efficient Identification and Signatures for Smart Cards”. In: *Advances in Cryptology — CRYPTO’ 89 Proceedings*. Ed. by Gilles Brassard. [cited 10-12-2021]. New York, NY: Springer New York, 1990, pp. 239–252. ISBN: 978-0-387-34805-6.

[40] Adi Shamir. “How to Share a Secret”. In: *Commun. ACM* 22.11 (Nov. 1979). [cited 1-12-2021], 612–613. ISSN: 0001-0782. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176.

[41] Eldar Sultanow and Alina Chircu. “A Review of IoT Technologies, Standards, Tools, Frameworks and Platforms”. In: *The Internet of Things in the Industrial Sector: Security and Device Connectivity, Smart Environments, and Industry 4.0*. Ed. by Zaigham Mahmood. [cited 18-11-2021]. Cham: Springer International Publishing, 2019, pp. 3–34. ISBN: 978-3-030-24892-5. DOI: 10.1007/978-3-030-24892-5_1. URL: https://doi.org/10.1007/978-3-030-24892-5_1.

[42] The GTK Team. *Building GTK*. [online]. [cited 28-11-2021]. URL: https://docs.gtk.org/gtk4/building.html.

[43] The GTK Team. *The GTK Project - A free and open-source cross-platform widget toolkit*. [online]. [cited 28-11-2021]. URL: https://www.gtk.org/.

Symbols and abbreviations

2D	2-Dimensional
3G	Third Generation of broadband cellular network
4G	Fourth Generation of broadband cellular network
5G	Fifth Generation of broadband cellular network
AES	Advanced Encryption Standard
AI	Artificial Intelligence
AKA	Authentication Key Agreement
AKE	Authentication Key Exchange
AWS	Amazon Web Services
BDS	Berkeley Software (or Standard) Distribution
BN	Big Numbers
CID	Client Identification
COTS	Commercial of the Shelf
CSS	Cascading Style Sheet
DH	Diffie-Hellman
DSA	Digital Signature Algorithm
DES	Data Encryption Standard
EC	Elliptic Curves
GB	Giga Byte
GCD	Greatest Common Divisor
GNU	recursive acronym for "GNU's Not Unix!"
GPU	Graphical Processing Unit
GTK	GIMP ToolKit
GUI	Graphical User Interface

HTTPS Hypertext Transfer Protocol Secure

IBM International Business Machines Corporation

IEEE Institute of Electrical and Electronics Engineers

IoT Internet of Things

IIoT Industrial Internet of Things

IP Internet Protocol

JSON JavaScript Object Notation

LCM Least Common Multiple

LLC Limited liability company

LoRaWAN Long Range Wide Area Network

MB Mega Byte

MD Message-Digest algorithm

MQTT Message Queue Telemetry Transport

NFC Near Field Communication

OS Operation System

RC4 Rivest Cipher 4

RPi Raspberry Pi

RSA Rivest-Shamir-Adleman algorithm

SHA Secure Hash Algorithm

SSAKA Secret Sharing-based Authentication Key Agreement

SSL Secure Sockets Layer

SSS Shamir's Secret Sharing

ShSSAKA Shamir's Secret Sharing-based Authentication Key Agreement

TB Tera Byte

TLS Transport Layer Security

UI	User Interface
VS Code	Microsoft Visual Studio Code
WSS	Web Soil Survey
WSL	Windows Subsystem for Linux

List of appendices

A Source code Directory Tree

For better navigation through the implemented code is in this chapter viewed the directory tree of the whole ShSSAKA application. The source code of the application was created in the Microsoft Visual Studio Code editor (version 1.62.3) with installed extension pack for the C language code development, the **C/C++ Extension Pack** (version 1.0.0) developed by *Microsoft*. This VS Code package is described in 5.

Along with the `src` directory, there is also included the `.vscode` directory, that contains the JSON files, which setts the VS Code environment for the compilation, launch and debugging of the C code. The ShSSAKA algorithm itself is implemented in the `ShSSAKA.c` file. Implementation also include a `Makefile` for the code compilation and a shell script for Linux OS.

- `/` ..... the root directory of the attached archive
  - `.vscode` ..... VS Code directory with `*.json` configuration files
    - `c_cpp_properties.json`
    - `launch.json`
    - `settings.json`
    - `tasks.json`
  - `src` ..... main source code directory
    - `c_files` ..... C files directory with the algorithm and support libraries
      - `AKA.c` ..... AKA library
      - `paillier_scheme.c` ..... Paillier library (Scheme 1 implementation)
      - `paishamir.c` ..... SSS protocol modified by Paillier scheme
      - `schnorrs_signature.c` ..... Schnorr's signature protocol modified to EC
      - `ShSSAKA.c` ..... ShSSAKA library
      - `support_functions.c` ..... library containing various functions for the protocols
    - `headers` ..... directory with C header files
      - `AKA.h`
      - `globals.h` ..... header file containing common and global parameters
      - `openssl_bn.h`
      - `paillier_scheme.h`
      - `paishamir.h`
      - `schnorrs_signature.h`
      - `ShSSAKA.h`
    - `precomputed_values`
      - `precomputed_message.json` ..... pre-computed message values
      - `precomputed_noise.json` ..... pre-computed noise values
    - `GUI.c` ..... GUI source file
    - `icon.jpg` ..... application icon file
    - `keychain.json` ..... saved Paillier key-chain used for pre-computations
    - `Linux-install.sh` . a `bash` script for the application installation on Linux OS
    - `main_test.c` ..... file containing testing blocks of each implemented library

- `Makefile` ..... Makefile for code compilation by `make`
- `style.css` ..... CSS file for GUI visualisation customisation
- `.gitignore` ..... GIT file defining files to ignore during upload to GitHub
- `README.md` ..... GIT file containing essential information about the application

B Libraries description

B.1 Globals

Macros

- NUM_THREADS – number of threads for pre-computations
- G_NUMOFDEVICES – total number of devices
- G_POLYDEGREE – minimum of devices to reconstruct the secret (polynomial degree)
- BUFFER – size of a buffer for strings
- BITS – bit size of generated values
- MAXITER – maximal number of iterations
- RANGE – range of pre-computed values from 0

Dependent libraries stdio.h, stdlib.h, time.h, math.h, limits.h, string.h, unistd.h, openssl/bn.h, openssl/sha.h, openssl/ec.h, openssl/obj_mac.h, cjson/cJSON.h

Defined structures

- **Paillier structures**:
  - *struct paillier_PrivateKey* – P, Q, λ and μ values
  - *struct paillier_PublicKey* – N, N_SQ (n^2) and G values
  - *struct paillier_Keychain* – Paillier PK and SK structures
- *struct ServerSign* – TAU_S (τ_s) value and KAPPA (κ) point
- *struct ClientProof* – TAU_C (τ_c) value, KAPPA (κ) point and Schnorr's Signature structure
- *struct DeviceProof* – signature S_I (σ_i) and point KAPPA_I (κ_i)
- **Schnorr structures**:
  - *struct schnorr_Keychain* – EC GROUP variable and EC key-pair for Schnorr's Signature
  - *struct schnorr_Signature* – hash, signature (σ), random r and point C_PRIME (C')
- *struct aka_Keychain* – Schnorr key-chain and ID value
- *struct shssaka_Keychain* – SK, PK, kappa (κ) and ID value
- *struct globals* – Schnorr's key-chain and ID counter value

Defined global variables:

- struct globals *g_globals* – holds global values (especially the EC group and idCounter)
- global key-chains:

  - struct shssaka_Keychain *g_shssaka_devicesKeys*[G_NUMOFDEVICES] – ShSSAKA keychain for client (0) and devices (1 to G_NUMOFDEVICES)
  - struct aka_Keychain *g_serverKeys* – server keychain (both for AKA and ShSSAKA)
  - struct aka_Keychain *g_aka_clientKeys* – client keychain for AKA
- unsigned int *currentNumberOfDevices* – holds the current number of used devices
- struct paillier_Keychain *g_paiKeys* – global Paillier key-chain
- EC_POINT **pk_c* – ShSSAKA client's public key
- pthread_t *threads*[NUM_THREADS] – threads field used for pre-computation
- BIGNUM **g_range* – global range value for pre-computation (created from RANGE macro)
- unsigned int *paillier_inited* – value 0 or 1, marks, whether was the Paillier key-chain initiated
- unsigned int *pre_noise* – value 0 or 1, marks, whether the pre-computed value of noise should be used
- unsigned int *pre_message* – value 0 or 1, marks, whether the pre-computed value of message should be used
- const char *restrict *file_keychain* – holds the path to the saved Paillier key-chain used for the pre-computations
- const char *restrict *file_precomputed_noise* – holds the path to the pre-computed noise values
- const char *restrict *file_precomputed_message* – holds the path to the pre-computed message values
- cJSON **json_noise* – holds the parsed noise JSON file after the application start
- cJSON **json_message* – holds the parsed message JSON file after the application start

B.2 AKA file

Dependent libraries: support_functions.h, schnorrs_signature.h, globals.h

aka_setup

- input: none
- output: unsigned int

Initialise the AKA protocol by allocating the memory and initialising the key-chains of server and client (both uses struct aka_Keychain). Returns 1 on success, 0 otherwise.

aka_serverSignVerify

- input:
  - BIGNUM *Y – numeric message value
  - struct ServerSign **server* – structure holding the server's point κ and σ (signature) value
- output: unsigned int

Simulates the communication between the server and client. Computes the server's Schnorr's signature and "sends" the message Y and its signature σ_S to the client (aka_clientProofVerify is carried out). After the client device "sends" the value of τ_C and point κ_C back, carries out the verification of the server's signature σ_S. If the verification proceeds ie. τ_S is set to 1 by Schnorr's verification procedure, the protocol ends successfully. Returns 1 on success, 0 otherwise.

aka_clientProofVerify

- input:
  - BIGNUM *Y – numeric message value
  - struct schnorr_Signature *server_signature – structure holding the values created during the server's signing procedure; holds the created hash, signature σ_S, generated random r and point C'
  - struct ClientProof **client* – structure holding the τ_C value, κ_C point and client's Schnorr's Signature structure
- output: unsigned int

Carries out the client's computation part of the communication simulation. First is verifying the server's signature. If the verification proceeds, the client's signature of the message is created and the client's part of the computation ends. Returns 1 on success, 0 otherwise.

init_aka_mem

- input: struct aka_Keychain **keychain*
- output: unsigned int

Initialize the given aka_Keychain. This procedure sets the ID of the device with the help of the idCounter (in the globals structure), allocates the memory for the keys and generates the Schnorr's key-chain. Returns 1 on success, 0 otherwise.

free_aka_mem

- input: struct aka_Keychain **keychain*
- output: none

Frees the memory used by the given key-chain.

aka_keyPrinter

- input: struct aka_Keychain *key
- output: none

Prints out the given key-chain into the console.

B.3 Paillier scheme file

Dependent libraries: support_functions.h, globals.h

paillier_generate_keypair

- input: struct paillier_Keychain *$keychain$
- output: unsigned int

Generates random P and Q primes and computes generator G with the procedure named `gen_pqg_params`. Also pre-computes the μ value used in the Paillier decryption procedure.

paillier_encrypt

- input:
  - struct paillier_PublicKey *pk – public key structure used for the encryption
  - BIGNUM *$plain$ – plaintext for the encryption
  - BIGNUM *$cipher$ – final ciphertext
  - BIGNUM *$precomp\_message$ – holds the value of the pre-computed message (if a global pre_message is set to 1)
  - BIGNUM *$precomp\_noise$ – holds the value of the pre-computed noise (if a global pre_noise is set to 1)
- output:unsigned int

First is checked whether the length of the message is smaller than the length of the N value. Then the Paillier encryption is performed. If the pre-computed values are not set (are set to 0) then the values are computed.

paillier_decrypt

- input:
  - struct paillier_Keychain *$keychain$ – Paillier key-chain with the public and private values
  - BIGNUM *$cipher$ – ciphertext for decryption
  - BIGNUM *$plain$ – decrypted plaintext
- output: unsigned int

Performs the Paillier decryption procedure. Returns 1 on success, 0 otherwise.

init_paillier_keychain

- input: struct paillier_Keychain **keychain*
- output: none

Initialise the given paillier_Keychain. This procedure allocates memory for the public key structure (paillier_PublicKey) – generator G, prime N, and N_SQ. This procedure allocates memory for the private key structure (paillier_PrivateKey) – λ and μ value, prime P and prime Q.

free_paillier_keychain

- input: struct paillier_Keychain **keychain*
- output: none

Frees the memory used by the given key-chain.

homomorphy_add

- input:
  - struct paillier_PublicKey **pk* – Paillier public key values
  - BIGNUM **enc_1* – first encrypted value to be summed
  - BIGNUM **enc_2* – second encrypted value to be summed
  - BIGNUM **res* – resulting value
- output: unsigned int

Procedure implementing the addition of two encrypted values using the additive homomorphic property of Paillier. Returns 1 on success, 0 otherwise.

homomorphy_add_const

- input:
  - struct paillier_PublicKey **pk* – Paillier public key values
  - BIGNUM **enc_value* – encrypted value
  - BIGNUM **constant* – constant which will be added to the encrypted value
  - BIGNUM **res* – resulting value
- output: unsigned int

Procedure implementing the addition of a constant to an encrypted value using the additive homomorphic property of Paillier. Returns 1 on success, 0 otherwise.

homomorphy_mul_const

- input:
  - struct paillier_PublicKey **pk* – Paillier public key values
  - BIGNUM **enc_value* – encrypted value

  - BIGNUM *$constant$ – constant which will be multiplied with the encrypted value
  - BIGNUM *res – resulting value
- output:unsigned int

Procedure implementing the multiplication of a constant to an encrypted value using the additive homomorphic property of Paillier. Returns 1 on success, 0 otherwise.

B.4 PaiShamir file

Dependent libraries: paillier_scheme.h, support_functions.h, globals.h

paiShamir_distribution

- input: struct paillier_Keychain *$paikeys$
- output: unsigned int

Carries out the distribution of the client's secret through out all of the devices (uses g_shssaka_devices[] field). Uses the Shamir's Secret Sharing principles modified with Paillier. Returns 1 on success, 0 otherwise.

paiShamir_get_ci

- input:
  - struct paillier_Keychain *$paikeys$ – Paillier key-chain used during the Shamir's Secret Sharing
  - BIGNUM *$kappa\_i$ – secret value (private key of a device)
  - BIGNUM *d[G_POLYDEGREE] – randomly generated values
  - BIGNUM *xs[G_POLYDEGREE] – pre-computed powers of the public key of a device
  - BIGNUM *mod – used modulo value
  - BIGNUM *ci – resulting value
- output: unsigned int

Procedure implementing the computation of encrypted device polynomial which is one part for the Shamir's Secret Sharing. Returns 1 on success, 0 otherwise.

paiShamir_get_cN_prime

- input:
  - struct paillier_Keychain *$paikeys$ – Paillier key-chain used during the Shamir's Secret Sharing
  - BIGNUM *pre_cN – resulting CI of the previous device
  - BIGNUM *cN – resulting CI of the current device
  - BIGNUM *cN_prime – resulting value

- output: unsigned int

Adds the result of the polynomial of the previous device to the current result. Returns 1 on success, 0 otherwise.

paiShamir_get_c

- input:
  - BIGNUM *$kappa$ – secret value (private key of a device)
  - BIGNUM *mod – used modulo value
  - BIGNUM *xs[G_POLYDEGREE] – pre-computed powers of the public key of a device
  - BIGNUM *d[G_POLYDEGREE] – randomly generated values
  - BIGNUM *c – resulting value
- output: unsigned int

Computes the polynomial of the device to which is the secret computed. Returns 1 on success, 0 otherwise.

paiShamir_get_share

- input:
  - struct paillier_Keychain *$paikeys$ – Paillier key-chain used for the decryption of the CN_PRIME (C'_N)
  - BIGNUM *cN_prime – encrypted value of summed encrypted polynomial results
  - BIGNUM *c – un-encrypted result of a polynomial of current device
  - BIGNUM *mod – used modulo value
  - BIGNUM *$share$ – resulting value
- output: unsigned int

Gets the part of the distributed secret. Decrypts the CN_PRIME (C'_N) value and adds the C. Result is then saved to SHARE value. Returns 1 on success, 0 otherwise.

paiShamir_interpolation

- input:
  - unsigned int *$devices\_list$ – list of the used devices for the secret reconstruction
  - unsigned int $size\_of\_list$ – size of the list with the used devices
  - BIGNUM *mod – used modulo value
  - BIGNUM *$secret$ – reconstructed secret
- output: unsigned int

Procedure for reconstructing the previously distributed secret. In the beginning is checked, if there are enough devices to reconstruct the secret value. Returns 1 on success, 0 otherwise.

part_interpolation

- input:
  - unsigned int **devices_list* – list of the used devices for the secret reconstruction
  - unsigned int *size_of_list* – size of the list with the used devices
  - unsigned int *current_device* – index from the list containing the current device
  - BIGNUM **mod* – used modulo value
  - BIGNUM **sk_i* —- gained part of the reconstructed secret value
- output: unsigned int

Computes one part of the interpolation for each used device. Returns 1 on success, 0 otherwise.

B.5 ShSSAKA file

Dependent libraries: support_functions.h, schnorrs_signature.h, paishamir.h, paillier_scheme.h, AKA.h, globals.h

shssaka_setup

- input: none
- output: unsigned int

Initialise the key-chains (g_serverKeys, g_deviceKeys[]). Allocates the memory and generate the public keys, IDs, distributes the secret and computes the clients public key. Returns 1 on success, 0 otherwise.

shssaka_KeyGeneration

- input: struct shssaka_Keychain **shssaka_Keychain*
- output: unsigned int

Procedure generating random values as a public key of a given device keychain and assignment of the ID (based on the idCounter value from the globals structure). Returns 1 on success, 0 otherwise.

shssaka_ClientAddShare

- input: unsigned int *num_of_new_devices*
- output: unsigned int

Generates given number of new devices, generates their values and saves them to the g_shssaka_devices[] field. Afterwards is recomputed the client's public key. Returns 1 on success, 0 otherwise.

shssaka_ClientRevShare

- input:
  - unsigned int **rev_devices_list* – list of devices containing the index values from the g_shssaka_devices[], that should be revoked
  - unsigned int *list_size* – size of the revocation list
- output: unsigned int

Revokes the given devices from the devices list, redistributes the client's secret and recomputes the client's public key. Returns 1 on success, 0 otherwise.

shssaka_akaServerSignVerify

- input:
  - unsigned int **list_of_used_devs* – list of devices containing the index values from the g_shssaka_devices[], that should be used during the signing procedure
  - unsigned int *size* – size of the used devices list
  - BIGNUM **Y* – numeric message value
  - struct ServerSign **server* – structure holding the server's point κ and σ (signature) value
- output: unsigned int

Procedure simulates the communication between the server, client and devices. Computes the server's Schnorr's signature and "sends" the message Y with it's signature σ_S to the client (`shssaka_clientProofVerify` is carried out). After the client's device "sends" the computed value τ_C and point κ_C back, carries out the verification of the client's signature σ_C. If the verification proceeds ie. τ_S is set to 1 by the Schnorr's verification procedure, the protocol ends successfully. Returns 1 on success, 0 otherwise.

shssaka_clientProofVerify

- input:
  - unsigned int **list_of_used_devs* – list of devices containing the index values from the g_shssaka_devices[], that should be used during the signing procedure
  - unsigned int *size* – size of the used devices list
  - BIGNUM **Y* – numeric message value

  - struct schnorr_Signature *$server\_signature$ — structure holding the values created during the server's signing procedure; holds the created hash, signature"σ_S, generated random r and point C'
  - struct ClientProof *$client$ – structure holding the τ_C value, point κ_C and client's Schnorr's Signature structure
- output: unsigned int

Carries out the clients computation part of the communication simulation. First is verified the server's signature. If the verification proceeds, the client's signature of the message is created with the help of the other devices, that each creates a part for the κ_C, t (value for the hash computation) and σ_C values with the use of the `paiShamir` library. Than the client's part of the computation ends. Returns 1 on success, 0 otherwise.

init_shssaka_mem

- input: none
- output: none

Initialise the server AKA keychain, client's shssaka_Keychain (the first device in the g_shssaka_devices[] field), all device key-chains and the PK_C point (client's main public key pk_C). Procedure also initialise the Paillier key-chain, if it was not initialised before. Returns 1 on success, 0 otherwise.

free_shssaka_mem

- input: none
- output: none

Frees the memory used by the server AKA keychain, client's shssaka_Keychain (the first device in the g_shssaka_devices[] field) all device key-chains and the PK_C point (client's main public key pk_C). Procedure also frees the Paillier keychain.

shssaka_keyPrinter

- input: struct shssaka_Keychain *key
- output: none

Prints out the given key-chain into the console.

B.6 Schnorr's signature file

Dependent libraries: support_functions.h, ShSSAKA.h, globals.h

gen_schnorr_keychain

- input:

  - const EC_GROUP *_group_ – group of the EC
  - struct schnorr_Keychain *_keychain_ – contains EC group (given by the group parameter of the EC) and corresponding keys
- output: unsigned int

Generates the EC keychain. Returns 1 on success, 0 otherwise.

schnorr_sign

- input:
  - EC_GROUP *_group_ – group of the EC
  - const BIGNUM *_sk_ – private key used for the signing
  - BIGNUM *_message_ – message to be signed
  - EC_POINT *_kappa_ – generated κ point used in the second verification process
  - struct schnorr_Signature *_signature_ – structure, where the computed values are stored
- output: unsigned int

Carries out the Schnorr's signing process in the EC modification. Returns 1 on success, 0 otherwise.

schnorr_verify

- input:
  - EC_GROUP *_group_ – group of the EC
  - const EC_POINT *_pk_ – public key used for verification of the signature
  - BIGNUM *_message_ – signed message
  - EC_POINT *_kappa_ – generated κ point used in the second verification process
  - struct schnorr_Signature *_signature_ – structure, where the computed values are stored
- output: unsigned int

Carries out Schnorr's verification process in the EC modification. Returns 1 on success, 0 otherwise.

free_schnorr_keychain

- input: struct schnorr_Keychain *_keys_
- output: none

Frees the used memory used by the Schnorr's key-chain.

init_schnorr_signature

- input:

  - const EC_GROUP *$group$
  - struct schnorr_Signature *$signature$
- output: none

Initialises and allocates the memory for the Schnorr's signature structure.

free_schnorr_signature

- input: struct schnorr_Signature *$signature$
- output: none

Frees the used memory used by the Schnorr's signature structure.

B.7 Support functions file

Dependent libraries: schnorrs_signature.h, ShSSAKA.h, globals.h

gen_pqg_params

- input:
  - BIGNUM *p – generated prime P
  - BIGNUM *q – generated prime Q
  - BIGNUM *$lambda$ – computed λ value
  - struct paillier_PublicKey *pk – Paillier public key structure
- output: unsigned int

Generates random P and Q prime number of a defined bit-length (used BITS macro). If the $GCD(P--1, Q--1) \neq 1$ in defined number of iterations (macro MAXITER) the procedure ends. If not, then are computed the N, N_SQ (n^2) and the λ value.

After that is computed the generator G. Again, if $GCD(G, N_SQ) \neq 1$, then the G value is recomputed (goes until the MAXITER number of iterations).

Returns 1 on success, 0 otherwise.

lcm

- input:
  - BIGNUM *a – first value for the LCM computation
  - BIGNUM *b – second value for the LCM computation
  - BIGNUM *res – resulting value
- output: unsigned int

Computes the $LCM(a, b)$. Returns 1 on success, 0 otherwise.

count_mi

- input:
  - BIGNUM *mi – resulting μ value

  - BIGNUM *g – generator value
  - BIGNUM *$lambda$ – pre-computed λ value
  - BIGNUM *n_sq – pre-computed n^2 value (modulo value)
  - BIGNUM *n – n value (divisor value)
- output: unsigned int

Computes the μ value as the $L(g^{\lambda} \mod n^2)^{-1}$. The L function is carried out by the eponymous implemented function. Returns 1 on success, 0 otherwise.

L

- input:
  - BIGNUM *u – given value
  - BIGNUM *n – divisor value
  - BIGNUM *res – resulting value
- output: unsigned int

Computes $(u-1)/n$. Returns 1 on success, 0 otherwise.

lambda_computation

- input:
  - BIGNUM *p – prime P value to compute λ
  - BIGNUM *q – prime Q value to compute λ
  - BIGNUM *$lambda$ – resulting value
- output: unsigned int

Computes $\lambda = LCM(P-1, Q-1)$. Returns 1 on success, 0 otherwise.

generate_rnd_paillier

- input:
  - BIGNUM *bn_range – top value for the generation
  - BIGNUM *gcd_chck – value to check GCD of the generated value with
  - BIGNUM *$random$ – resulting value
- output: unsigned int

Generates random value with $GCD(random, gcd\_chck) = 1$ property. Returns 1 on success, 0 otherwise.

ec_hash

- input:
  - const EC_GROUP *$group$ – EC group variable
  - BIGNUM *res – resulting value
  - BIGNUM *Y – numeric message
  - EC_POINT *t_s – value of t computed by the Schnorr's signature

  - EC_POINT *$kappa$ – κ value computed by the Schnorr's signature
- output: unsigned int

Procedure implementing the concatenation of needed values for the (ShSS)AKA protocol. Concatenates the message, t and κ in this order and creates hash SHA256. Returns 1 on success, 0 otherwise.

rand_range

- input:
  - BIGNUM *rnd – resulting random value
  - const BIGNUM *bn_range – top value for the generation
- output: unsigned int

Generates random value in a range, where the top value is (bn_range – 1). Returns 1 on success, 0 otherwise.

rand_point

- input:
  - const EC_GROUP *$group$ – EC group variable
  - EC_POINT *$point$ – resulting random point
- output: unsigned int

Generate a random point on the elliptic curve given by the EC generator. Returns 1 on success, 0 otherwise.

set_precomps

- input:
  - BIGNUM *$message$ – searched message value
  - BIGNUM *$pre\_message$ – resulting pre-computed message value
  - BIGNUM *pre_noise – resulting pre-computed noise value
- output: unsigned int

Sets the pre-computed values before the Paillier encryption. If the global pre_message or pre_noise is set to 1 given value of the message or randomly generated value is found in the global JSON files. Otherwise, the values are set to 0.

init_serversign

- input:
  - const EC_GROUP *$group$ – EC group variable
  - struct ServerSign *$server\_sign$
- output: none

Initialise the memory for the given Server's signature structure.

free_serversign

- input: struct ServerSign *$server\_sign$*
- output: none

Frees the memory used by the given Server's signature structure.

init_clientproof

- input:
  - const EC_GROUP *$group$ – EC group variable
  - struct ClientProof *$client\_proof$*
- output: none

Initialise the memory for the given Client's proof structure.

free_clientproof

- input: struct ClientProof *$client\_proof$*
- output: none

Frees the memory used by the given Client's proof structure.

init_deviceproof

- input:
  - const EC_GROUP *$group$ – EC group variable
  - struct DeviceProof *$device\_proof$*
- output: none

Initialise the memory for the given Device's proof structure.

free_deviceproof

- input: struct DeviceProof *$device\_proof$*
- output: none

Frees the memory used by the given Device's proof structure.

***thread_creation**

- input: void *$threadid$*
- output: none

Procedure called by the `pthread_create` for the pre-computation of the message and noise values.

threaded_precomputation

- input: none
- output: unsigned int

Procedure calling the thread creation. Returns 0 on success, otherwise exits the application with the -1 code.

parse_JSON

- input: const char *restrict *file_name*
- output: cJSON *

Procedure for parsing the JSON file defined by the given path. Returns cJSON* object with the parsed JSON file.

find_value

- input:
  - cJSON **json* – object with parsed JSON file
  - BIGNUM **search* – searched value
  - BIGNUM **result* – found value
- output: unsigned int

Procedure for finding the given value SEARCH in the given parsed JSON file. Returns 1 on success, 0 otherwise.

read_keys

- input:
  - const char *restrict *file_name* – path to the JSON file
  - struct paillier_Keychain **keychain*
- output: none

Procedure for loading the saved Paillier key-chain into the given structure from the file in the given path.

write_keys

- input:
  - const char *restrict *file_name* – path to the JSON file
  - struct paillier_Keychain **keychain*
- output: int

Saves the given Paillier keychain into the JSON file in the given path. Returns 0 on success, EOF otherwise.

precomputation

- input:
  - const char *restrict *file_name* – path to the JSON file
  - struct paillier_Keychain **keychain*
  - unsigned int *int_range* – top value for the pre-computation

  - unsigned int *type* – decides which type of pre-computation will be carried out
- output: int

Starts the pre-computation procedure based on the given type (type = 1 for message pre-computation, type = 2 for noise pre-computation). Function Is called by the `thread_creation`. Returns 0 on success, EOF otherwise.

message_precomp

- input:
  - unsigned int *int_range* – top value for the message pre-computation
  - BIGNUM **base* – Paillier generator for the message pre-computation
  - BIGNUM **mod* – value for modulo operation (Paillier value of n^2)
- output: cJSON *

Procedure for pre-computation of the $g^m \mod n^2$ for the Paillier encryption acceleration. During the procedure, the achieved values are saved to the cJSON* object which is then returned.

noise_precomp

- input:
  - unsigned int *int_range* – top value for the noise pre-computation
  - BIGNUM **exp_value* – Paillier value of N
  - BIGNUM **mod* – value for modulo operation (Paillier value of n^2)
- output: cJSON *

Procedure for pre-computation of the $r^n \mod n^2$ for the Paillier encryption acceleration. During the procedure, the achieved values are saved to the cJSON* object which is then returned.