TECHNISCHE UNIVERSITÄT DARMSTADT

Master-Thesis

Hardware-Assisted Two-Party Secure Computation on Mobile Devices

Daniel Demmler

July 2013

Technische Universität Darmstadt
CASED
EC SPRIDE

Supervisors: Dr. Thomas Schneider
M.Sc. Michael Zohner

Declaration of Authorship

I certify that the work presented here is, to the best of my knowledge and belief, original and the result of my own investigations, except as acknowledged, and has not been submitted, either in part or whole, for a degree at this or any other University.

Darmstadt, July 5, 2013

_____

Daniel Demmler

Abstract

This thesis focuses on the practical realization of general two-party Secure Function Evaluation in a mobile environment and the possibility of enhancing these techniques by the use of a trusted hardware token.

Secure function evaluation allows multiple mutually distrusting parties to jointly compute a function on their private inputs without revealing anything but the function output. This technique is particularly interesting in the context of mobile electronics, such as smartphones, where typically highly sensitive user data is stored and processed. The protection of this data is desirable but very costly, due to the high complexity of secure computation protocols. Implementing Secure Function Evaluation schemes on smartphones is a challenging task due to their limitations in processing power, memory and battery-life.

To address these issues, we extended an existing two-party secure function evaluation scheme by a trusted hardware token that allows to securely pre-generate data, used in the actual function evaluation phase for masking sensitive values. For the purpose of securely distributing data generated by the token, we designed and implemented a communication protocol based on TLS on the smart card. We present working demonstrators for managing the hardware token and running secure two-party function evaluation on Android smart phones making use of a microSD smart card. The use cases we implemented are private set intersection to find shared contacts and securely scheduling a meeting. Our implementation is benchmarked and its performance is analyzed.

Acknowledgments

First, I would like to thank my supervisors Thomas Schneider and Michael Zohner for giving me the opportunity to work in their group and for introducing me to the fascinating topic of secure computation. I am grateful for their kind supervision in the last months and their inspiring input and directions, while at the same time giving me the freedom to work freely. I am thankful for all their explanations and helpful comments on my ideas – of which so many urgently required the feedback. It was a privilege and a pleasure to work with them and I am looking forward to continuing our work together.

I am very grateful to my parents for supporting me during the whole time of my studies in so many ways.

Many thanks to my friends for the great times we had. I would especially like to thank David Meier and Daniel Steinmetzer for proof-reading this thesis and their valuable feedback. Thanks as well to Patrick Lieser for the time studying together as well as the Fachschaft iST for giving me an occupation next to writing this thesis.

Last but not least, I would like to thank Lisa for the wonderful time together and her understanding during the last months. Thanks for always encouraging me to carry on and for proof-reading early drafts of this work. I would not be where I am today, if it wasn't for you.

Contents

1 Introduction

Today, communication is more important than ever. The amount of exchanged data is constantly growing and electronic communication devices are an integral part of our everyday lives. With the hype of smartphones during the last years a lot of people are constantly online and have their private data digitally available at almost all times. However, many questions regarding confidentiality and privacy arise with the high amount of information that is processed. *Secure two-party computation* is a rapidly growing field of research that offers solutions to privacy preserving computation and communication problems. Especially in the context of recent mass electronic surveillance programs, the use of Secure Function Evaluation (SFE) techniques becomes more relevant than ever, and could potentially play an important role in protecting sensitive data.

In general, secure two-party computation allows two mutually distrusting parties to jointly evaluate functions on their private input data without revealing anything but the final result of the calculation.

While first approaches to securely evaluating functions were rather impractical and computationally expensive, the technological development in processing speed and the massive improvement of communication bandwidth enabled practical implementations of secure computation protocols. Recent research has improved protocols in a way that even mobile devices are nowadays capable of using optimized SFE protocols to a certain extent. This allows for many interesting use cases due to the wide spread of smartphones and the personal data steadily accessible with those devices.

1.1 Motivation

SFE on mobile devices such as smartphones is still a challenging problem due to the limited available computation power, memory and energy on the battery-powered devices. On the other hand, smartphones are a very promising environment for secure computation. They are steadily available for the user and typically hold a multitude of privacy critical information. This data is interesting to process and to share, but needs to be properly protected by various cryptographic measures. Therefore, implementing fully functional SFE applications on recent smartphones is an intriguing task.

Improving SFE schemes by using trusted Hardware (HW) tokens and offloading work to them has already been researched, e.g. [JKSS10] and shown to be a promising approach to improving performance issues.

Using tamper-proof and trusted HW tokens, such as Smart Cards (SCs), we are able to execute trusted pre-calculations and secure data distribution. With this securely generated data the secure computation's performance can be improved.

While simply sending the private input data of all participants to a trusted third party that evaluates the desired function would be a trivial solution to all SFE problems, the token considered here is very restricted in its resources, acts solely as an extension to the protocol and does not get access to any private data.

1.2 Contributions

We implemented a generic SFE scheme based on the two-party protocol of Goldreich, Micali and Wigderson [GMW87]. In our implementation we make use of a SC that has limited computational power but is partially trusted by all participants and hardened against tampering, which allows us to use it in a setup phase to pre-generate data in a trusted way. With the help of this data, we are able to improve the performance of the secure computation. The SC is only used to securely generate and distribute auxiliary data to the parties involved and does not get access to sensitive user data. In contrast to many custom-tailored protocols our generic solution allows the execution of arbitrary functionality that can be modeled as a Boolean circuit.

We base our work on a semi-honest threat model that assumes that all participants are following the protocol as specified, but may try to obtain additional information from data they observe during communication.

A proof-of-concept implementation has been created for our setting: A Java Card (JC) applet running on the SC as well as an Android service managing the SC connection and the pre-generation of data. We developed an additional demonstrator app that executes secure computation use cases on Android smart phones. It relies on a local Wi-Fi Direct connection that is established between the parties. Our protocol is solely based on direct connections between the users and at no point private data is shared with the HW token or sent over the internet. The participants in our scenario can be completely unknown to each other before the protocol execution and do not have to exchange the SC itself. This allows performing ad-hoc secure computation as soon as the required software is available and one party has access to a supported SC.

1.3 Outline

In Chapter 2 basic notation and the theoretical background to this thesis are introduced. Chapter 3 gives an overview over related work in the field of secure computation and the concepts this thesis relies on are shown. The design ideas for our SFE protocol are explained in Chapter 4. Next, Chapter 5 gives a detailed overview of our proof-of-concept implementation, presents relevant performance benchmarks and analyzes them. Finally, Chapter 6 concludes this thesis and gives an outlook to future work.

2 Fundamentals

This chapter introduces fundamental concepts and common notations used later in the context of secure computation. It is intended to explain definitions used in this thesis and to serve as a short introduction for readers not completely familiar with the topic.

2.1 Notation

Numbers and Digital Information We are using standard prefixes from the international system of units (SI) based on powers of 10, such as milli, kilo, mega and giga and we denote them with their corresponding symbols m, k, M and G ($10^{-3}, 10^3, 10^6, 10^9$).

For digital information we employ binary prefixes according to ISO/IEC 80000 such as kibi, mebi, gibi and their corresponding symbols Ki, Mi, Gi ($2^{10}, 2^{20}, 2^{30}$). We use the capital letter B for byte and the lower case b for bit, if the units are not written out.

Bit Strings and Binary Operations The set of binary strings of length n are denoted as $\{0,1\}^n$, while arbitrary length bit strings are denoted by $\{0,1\}^*$. $a||b$ denotes the concatenation of two (bit-)strings a and b. $a \wedge b$ denotes the bitwise AND operation, that can also be written as arithmetic product ab without any operator. $a \oplus b$ denotes the bitwise exclusive-or (XOR) operation and $a \vee b$ denotes the bitwise OR operation of $a, b \in \{0,1\}^n$. A randomly selected bit x is denoted as $x \in_R \{0,1\}$.

Byte vectors are given in big-endian form. In a n byte vector, the most significant byte is placed left and has the index 0, while the least significant byte is placed right and has index $(n-1)$.

Encryption $c = \mathrm{Enc}_k^A(m)$ denotes the symmetric encryption of message m with key k under cipher A to generate ciphertext c. The according decryption is denoted as $m = \mathrm{Dec}_k^A(c)$. Asymmetric encryption under cipher A with $\mathcal{B}$'s public key $k_{\mathrm{Pub},\mathcal{B}}$ is denoted as $\mathrm{Enc}^A_{k_{\mathrm{Pub},\mathcal{B}}}$, while the decryption with $\mathcal{B}$'s private key $k_{\mathrm{Priv},\mathcal{B}}$ is denoted as $\mathrm{Dec}^A_{k_{\mathrm{Priv},\mathcal{B}}}$.

2.2 Cryptographic Primitives and Protocols

The following primitives and the TLS protocol handshake are fundamental to our work and are therefore briefly explained.

Pseudo-Random Functions Pseudo-Random Functions (PRFs) are efficiently-computable deterministic functions that create a pseudo-random output based on its input variable x and a given key k. They are denoted as $\text{PRF}_k(x)$ and can be implemented based on cryptographic hash functions or block ciphers.

Message Authentication Codes and HMAC Message Authentication Codes (MACs) are constructions that can be generated from a message m with a symmetric key k in order to allow verification of authenticity and integrity of m. They are denoted as $\text{MAC}_k(m)$. One instance is the keyed-hash MAC (HMAC) [KBC97] that is based on hash functions.

2.2.1 Transport Layer Security Handshake

Transport Layer Security (TLS) is a cryptographic protocol that is widely used to secure and authenticate internet communication between a client $\mathcal{C}$ and a server $\mathcal{S}$. TLS and its predecessor Secure Sockets Layer (SSL) have undergone thorough security analysis, e.g. [MSW08; WS96], that led to several refinements up to the most recent version TLS 1.2 [DR08]. This section gives an overview of the basic definitions in the standard.

A basic version of the TLS 1.2 handshake with authentication of server $\mathcal{S}$ and an unauthenticated client $\mathcal{C}$ using RSA encryption is shown in Protocol 2.1. TLS offers a lot of opportunities and can be applied to different application scenarios. Most of its functionality is optional and can be omitted, depending on the intended use. We leave out all messages that are not required for our later use. The removed messages were used to agree upon used ciphers and message digests. They are not necessary in our use case, since we can assume these values as predefined. Naturally, we also left out the client certificate messages, since $\mathcal{C}$ is not authenticated.

$$\mathcal{C} \to \mathcal{S} : R_{\mathcal{C}} \,||\, T_{\mathcal{C}} \tag{1}$$
$$\mathcal{S} \to \mathcal{C} : R_{\mathcal{S}} \,||\, T_{\mathcal{S}} \tag{2}$$
$$\mathcal{S} \to \mathcal{C} : \text{Certificate} \tag{3}$$
$$\mathcal{C} \to \mathcal{S} : \text{Enc}^{RSA}_{k_{\text{Pub},\mathcal{S}}}(PMS) \tag{4}$$
$$\mathcal{C} \to \mathcal{S} : \text{PRF}_{MS}\big(\text{“client finished”} \,||\, \text{Hash}\big((1) \,||\, (2) \,||\, (3) \,||\, (4)\big)\big) \tag{5}$$
$$\mathcal{S} \to \mathcal{C} : \text{PRF}_{MS}\big(\text{“server finished”} \,||\, \text{Hash}\big((1) \,||\, (2) \,||\, (3) \,||\, (4) \,||\, (5)\big)\big) \tag{6}$$

Protocol 2.1: TLS RSA Handshake without Client Authentication

The first two messages (1) and (2) include the 28 byte random values $R_{\mathcal{C}}$ and $R_{\mathcal{S}}$ and 4 byte timestamps $T_{\mathcal{C}}$ and $T_{\mathcal{S}}$, generated from $\mathcal{C}$ and $\mathcal{S}$ respectively. These values are later used to generate the final symmetric key material. They aim to prevent replay attacks by making every session unique. Next, the public key certificate of $\mathcal{S}$ is sent to $\mathcal{C}$ in (3). The fourth message (4) transports a 48 byte random premaster secret PMS from $\mathcal{C}$ to $\mathcal{S}$, encrypted with

$\mathcal{S}$'s RSA public key. The next message (5) contains a 12 byte PRF output that authenticates the encrypted premaster secret and all previous messages, binding the random values, timestamps and the certificate to this specific session. This is done using the master secret MS, which is derived as shown in Equation (2.3) as key. In the final message (6) $\mathcal{S}$ calculates a similar 12 byte PRF output over the previous messages, including $\mathcal{C}$'s PRF output from message (4). When the PRF verification succeeds on both sides, it can be assumed that the key-agreement was successful and both parties share the same session keys.

Multiple TLS messages can be combined together into one packet while a single message can also be fragmented into several packets. This happens because TLS is typically implemented on a packet-oriented TCP/IP connection.

The key derivation makes use of the defined TLS PRF shown in Equation (2.1) and Equation (2.2) that is based on a data expansion function to transform a given secret and seed data into an arbitrary amount of output data (cf. [DR08, Sec. 5]). Note that our notation of the PRF differs slightly from the one used in the standard. We are directly concatenating the values label and seed to our parameter data.

$$\begin{aligned} \mathrm{PRF}_k(\text{data}) = &\mathrm{HMAC}_k(A(1) \,||\, \text{data}) \,||\, \swarrow \\ &\mathrm{HMAC}_k(A(2) \,||\, \text{data}) \,||\, \swarrow \\ &\dots \end{aligned} \tag{2.1}$$

$$\begin{aligned} A(0) &= k \\ A(i) &= \mathrm{HMAC}_k\left(A(i-1)\right) \end{aligned} \tag{2.2}$$

The client's and the server's random values R and timestamps T as well as the constant label "master secret" are used as inputs for the PRF together with PMS as the key. The nested HMACs are evaluated as specified to generate the master secret MS.

$$MS = \mathrm{PRF}_{PMS}\left(\text{"master secret"} \,||\, R_{\mathcal{C}} \,||\, T_{\mathcal{C}} \,||\, R_{\mathcal{S}} \,||\, T_{\mathcal{S}}\right) \tag{2.3}$$

2.3 Boolean Circuits

Boolean circuits are a standard representation of (Boolean) functions. Every boolean circuit has n inputs, m outputs and contains g gates, which correspond to a certain Boolean function. An example is given in Figure 2.1 where $n = 4, m = 3$ and $g = 3$.

Boolean circuits are used to describe the function that is evaluated in secure computation protocols. Arbitrary Boolean functions can be represented by Boolean circuits consisting only of XOR and AND gates.

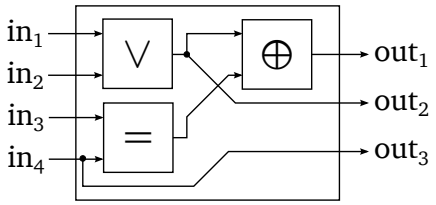


Figure 2.1: A Boolean Circuit Example with $n = 4$ input wires, $m = 3$ output wires and $g = 3$ gates.

2.4 Smart Cards

The term Smart Card (SC) in general refers to small embedded integrated circuits, typically portable and in a small sized plastic housing in the form of a SIM card or a credit card that holds a secure element. However, the actual integrated circuit can also be directly wired to an electronic device and offer similar functionality. It comes with limited computational capabilities and a small amount of memory that is separated in transient Random Access Memory (RAM) and persistent Electrically Erasable Programmable Read-Only Memory (EEPROM). The number of clear or write operations to the EEPROM is limited and therefore write access during the regular runtime of a program should be restricted to a minimum or avoided, if possible. This also benefits the overall speed of the program since RAM access is typically significantly faster than accessing the non-volatile memory.

SCs are hardened against physical attacks. This is typically achieved by a magnitude of sensors that aim to detect physical or electrical tampering and trigger a hardware-reset if certain thresholds are exceeded. Furthermore, several software mechanisms aim to guarantee additional protection. SCs can only be programmed through a secure channel by the issuer that holds a set of keys for authentication. The issuer also takes care of personalization to deploy unique identification numbers or card specific keys. This usually takes place during the one-time setup of the Software (SW). After the cards are deployed to the end user, the communication is limited to the installed SW on the SC and its interface. The user can only communicate over this pre-defined set of instructions. Manipulating the installed SW without the issuer's keys is prevented by various security measures.

The SC is externally powered by the SC reader. Therefore, the reader has full control over the SC and can reset it anytime, invalidating the volatile memory. Communication with the SC is always initiated by the reader and every reply from the SC has to be requested.

When assuming SCs as tamper-proof, we refer to their measures against physical attacks. In fact, there is no perfect security and every system can certainly be broken given sufficient resources. The hardening of the SC aims to make attacks as costly as possible, in a way that the reward of successfully breaking the SC's security is far less than the investment that has to be made to do so. There are several certifications, such as the Federal Information Processing

Standards (FIPS) 140-2 or the Common Criteria for Information Technology Security Evaluation (CC) Evaluation Assurance Level (EAL) that specify the degree of robustness against attacks.

2.5 The Android Operating System

The first commercially available version of the Android operating system was released by Google in 2008. It is based on a modified version of the Linux kernel. Android is by far the most popular mobile operating system today [Int12]. The most recent version is 4.2.2 (Jelly Bean) that was publicly released in February 2013. Applications (apps) for Android can be developed in the Java programming language in a convenient way, since a lot of features are available through a well-documented Application Programming Interface (API). Amongst these features are high-level control of the smartphone hardware and access to the stored data, such as contacts or calendars. An app has to request permission in order to access restricted functionalities. This happens during the app's setup and must actively be acknowledged by the user.

Android apps can be built from so called *activities* that refer to a single user interface containing content and possibilities for the user to interact. In contrast, *services* provide their functionality usually without a Graphical User Interface (GUI). Inter-process communication between applications and/or services can be done, amongst others, with so called *intents* that transport information either to a distinct target or broadcast it to everyone.

3 Related Work

The developments in the area of generic SFE can be traced back to two approaches: Yao's Garbled Circuit (GC) protocol [Yao86] and the protocol of Goldreich, Micali and Wigderson [GMW87].

There are further approaches in realizing secure computation, such as *homomorphic encryption* that allows operating directly on ciphertexts, due to the nature of the underlying encryption schemes. Unlike the circuit based approaches, these schemes are usually custom-tailored to a specific problem and can often not be used for arbitrary computations. Homomorphic encryption is a distinct field of research itself and outside the focus of this thesis.

3.1 Oblivious Transfer

Oblivious Transfer (OT) is a two-party protocol, originally introduced by Rabin [Rab81], where a sender inputs multiple bit strings and a chooser obliviously picks a subset of these inputs. The sender does not learn which bit strings have been chosen while the chooser does not gain any information about the inputs he didn't select. An OT protocol with n input values and one selection is referred to as 1-out-of-n OT. A widely used protocol by Naor and Pinkas [NP05] is secure in the semi-honest adversary model (cf. Section 4.1.1) and requires $\mathcal{O}(m)$ modular exponentiations by both parties, where m is the total number of oblivious selections made.

The performance of OTs can be improved by several techniques, such as pre-computing OTs on random values and using them later as one-time pads to mask the actual inputs [Bea95]. *OT extensions* [IKNP03] require the one-time evaluation of t base OTs on t-bit keys, where t is a security parameter. The base OTs are extended by evaluating a small amount of iterations of a cryptographic hash function and a PRF, thus reducing the large amount of computationally expensive modular exponentiations.

3.2 Yao's Garbled Circuit Protocol

Yao's GC protocol takes place between a circuit *creator* and an *evaluator*. The basic idea is for the creator to select two secret keys that represent an encryption of the wire values 0 and 1

for each wire in a Boolean circuit, in order to hide its actual value. For each gate a permuted garbled table is computed by the creator to represent the function of the gate on all possible encrypted input values. The creator's garbled inputs are directly transferred to the evaluator, while the evaluator receives his garbled inputs through an OT protocol from the creator. The evaluator can then iteratively decrypt the gate's garbled output value until he obtains the circuit's garbled output. Depending on which party then should obtain the output, the creator either sends a mapping from garbled outputs to real output values to the evaluator or the evaluator sends the circuit's garbled output values to the creator.

Since GCs run in a constant number of rounds and only symmetric cryptography is used, a lot of research was focused on optimizing and realizing practical examples of SFE scenarios based on GCs, e.g. [MNPS04; KS08; HEKM11].

3.3 Multiplication Triples

One alternative to using OT for evaluating AND gates in the GMW protocol (cf. Section 3.4.2) are Beaver's Multiplication Triples (MTs) [Bea91]. MTs are random-looking shares $a_i, b_i, c_i,\ i \in \{\mathcal{S}, \mathcal{C}\}$ that satisfy Equation (3.1):

$$c_\mathcal{S} \oplus c_\mathcal{C} = (a_\mathcal{S} \oplus a_\mathcal{C})(b_\mathcal{S} \oplus b_\mathcal{C}) \tag{3.1}$$

MTs themselves are independent from private input and can therefore be pre-generated before the actual secure computation. During the SFE phase they are used to mask private values.

MTs can only be used once and have to be discarded afterwards, since they would otherwise leak information about the masked data. Furthermore, MTs cannot be generated by one of the SFE participants alone, as knowing all shares of a MT would leak information about processed values. $a_\mathcal{S}, b_\mathcal{S}$ and $c_\mathcal{S}$ must only be known by $\mathcal{S}$, while $a_\mathcal{C}, b_\mathcal{C}$ and $c_\mathcal{C}$ must only be known by $\mathcal{C}$. MTs can securely be generated for instance by obtaining them using a 1-out-of-4 OT protocol in the setup phase or via a trusted third party that generates and distributes them to the participants in a secure way.

3.4 GMW Protocol

The protocol by Goldwasser, Micali and Wigderson [GMW87] enables two mutually distrusting parties $\mathcal{S}$ and $\mathcal{C}$ to interactively compute a function $f(x, y)$ represented as a Boolean circuit on their respective private inputs x and y, where $x, y \in \{0,1\}^\sigma$ for a certain input size σ, while keeping the privacy of the input data. In general, each value v is shared between the two participants in a way that each party $i \in \{\mathcal{S}, \mathcal{C}\}$ holds a random-looking share v_i for which $v = v_\mathcal{C} \oplus v_\mathcal{S}$ holds. The GMW protocol can be extended to a multi-party setting.

Recent publications have shown that GMW constructions can be as efficient as schemes that rely on additively homomorphic encryption for multiple parties [CHK+12] and even more efficient than Yao's GCs in the two-party case [SZ13], even though an interactive evaluation of AND gates and a large amount of OTs is necessary.

The GMW protocol is split into a *setup phase* that can be pre-calculated offline and an *online phase* where the input sharing, function evaluation and output reconstruction takes place in an interactive fashion between the parties. Recent work has shown that the setup phase is typically much more complex than the online phase (cf. [SZ13, Sect. 4]).

3.4.1 Secret Input Sharing

This stage hides the private inputs in random-looking shares and divides them among the participants.

To secretly share his inputs, $\mathcal{S}$ selects a random bit string $x_\mathcal{S} \in_R \{0,1\}^\sigma$. Then, he calculates $x_\mathcal{C} = x \oplus x_\mathcal{S}$ and sends $x_\mathcal{C}$ to $\mathcal{C}$. For the private input y, $\mathcal{C}$ will proceed similarly, picking a random $y_\mathcal{C} \in_R \{0,1\}^\sigma$ and then sending $y_\mathcal{S} = y \oplus y_\mathcal{C}$ to $\mathcal{S}$. This is depicted in Protocol 3.1.

$$\mathcal{S} \rightarrow \mathcal{C} : x_\mathcal{C} = x \oplus x_\mathcal{S} \tag{1}$$
$$\mathcal{C} \rightarrow \mathcal{S} : y_\mathcal{S} = y \oplus y_\mathcal{C} \tag{2}$$

Protocol 3.1: GMW Secret Input Sharing

After the inputs have been secret shared, $\mathcal{S}$ knows $x_\mathcal{S}$ and $y_\mathcal{S}$, while $\mathcal{C}$ knows $x_\mathcal{C}$ and $y_\mathcal{C}$. Single bits out of these shares are then assigned to the gates in the Boolean circuit. This step does not require any encryption and needs only one communication step. It is therefore uncritical to the overall performance.

3.4.2 Circuit Evaluation

Evaluating XOR gates does not require communication between the parties, since the operation is associative and only the local shares have to be XORed. Thus, we consider XOR gates as essentially free.

AND gates have to be evaluated interactively, which is the bottleneck of this protocol. However, AND gates on one circuit layer can be evaluated in parallel. Thus, optimizing the depth of a circuit is crucial for a good overall performance.

The evaluation of AND gates can be realized with the use of a 1-out-of-4 OT protocol to obliviously obtain the gate's output. Alternatively the parties can make use of their respective

MT shares a_i and b_i to mask their gate inputs x_i and y_i for $i \in \{\mathcal{SC}\}$. The input values are XORed with the MT shares and exchanged as shown in Protocol 3.2.

$$\mathcal{S} \rightarrow \mathcal{C} : d_\mathcal{S} = x_\mathcal{S} \oplus a_\mathcal{S},\ e_\mathcal{S} = y_\mathcal{S} \oplus b_\mathcal{S} \quad (1)$$
$$\mathcal{C} \rightarrow \mathcal{S} : d_\mathcal{C} = x_\mathcal{C} \oplus a_\mathcal{C},\ e_\mathcal{C} = y_\mathcal{C} \oplus b_\mathcal{C} \quad (2)$$

Protocol 3.2: AND Gate Input Sharing

After exchanging the masked input shares the variables d and e are calculated locally by both parties. Note that $x = x_\mathcal{C} \oplus x_\mathcal{S}$, $y = y_\mathcal{C} \oplus y_\mathcal{S}$, $a = a_\mathcal{C} \oplus a_\mathcal{S}$ and $b = b_\mathcal{C} \oplus b_\mathcal{S}$.

$$\begin{aligned} d &= d_\mathcal{S} \oplus d_\mathcal{C} = x_\mathcal{S} \oplus a_\mathcal{S} \oplus x_\mathcal{C} \oplus a_\mathcal{C} = x \oplus a \\ e &= e_\mathcal{S} \oplus e_\mathcal{C} = y_\mathcal{S} \oplus b_\mathcal{S} \oplus y_\mathcal{C} \oplus b_\mathcal{C} = y \oplus b \end{aligned} \quad (3.2)$$

The respective output shares of the AND gate are calculated with the MT shares $c_\mathcal{S}$ and $c_\mathcal{C}$ as shown in Equation (3.3).

$$\begin{aligned} \mathcal{S} :\ z_\mathcal{S} &= de \oplus db_\mathcal{S} \oplus ea_\mathcal{S} \oplus c_\mathcal{S} \\ \mathcal{C} :\ z_\mathcal{C} &= db_\mathcal{C} \oplus ea_\mathcal{C} \oplus c_\mathcal{C} \end{aligned} \quad (3.3)$$

By calculating $z_\mathcal{S} \oplus z_\mathcal{C}$ the AND gate is evaluated as shown in Equation (3.4) and the product of the input values is reconstructed. Note that $c = c_\mathcal{C} \oplus c_\mathcal{S} = ab$, according to the definition of MT. Thus $ab \oplus c = 0$, which allows for the removal in the last step.

$$\begin{aligned} z_\mathcal{S} \oplus z_\mathcal{C} &= de \oplus dy_\mathcal{S} \oplus ex_\mathcal{S} \oplus c_\mathcal{S} \oplus dy_\mathcal{C} \oplus ex_\mathcal{C} \oplus c_\mathcal{C} \\ &= (x \oplus a)(y \oplus b) \oplus (x \oplus a)y_\mathcal{S} \oplus (y \oplus b)x_\mathcal{S} \oplus c_\mathcal{S} \oplus (x \oplus a)y_\mathcal{C} \oplus (y \oplus b)x_\mathcal{C} \oplus c_\mathcal{C} \\ &= (x \oplus a)(y \oplus b) \oplus (x \oplus a)y \oplus (y \oplus b)x \oplus c \\ &= (x \oplus a)(y \oplus b) \oplus (xy \oplus ay) \oplus (yx \oplus bx) \oplus c \\ &= xy \oplus ay \oplus xb \oplus ab \oplus xy \oplus ay \oplus yx \oplus bx \oplus c \\ &= xy \oplus ab \oplus c \\ &= xy \end{aligned} \quad (3.4)$$

It is sufficient to have the ability to evaluate AND and XOR gates, since all other Boolean gates can be constructed from these two functions.

NOT gates are essentially free, since they can be realized with a free XOR gate and the Boolean constant 1 by having one party locally calculate $\bar{x} = 1 \oplus x$.

OR gates are constructed from two XOR gates and a single AND gate since $a \vee b = a \oplus b \oplus (a \wedge b)$. They have to be evaluated interactively and are therefore as costly as AND gates.

3.4.3 Output Reconstruction

The final step of the GMW protocol is reconstructing the actual function outputs by removing the random masks. To get the result of the circuit, the secret shares z_i of the final output wires of the circuit are exchanged and each party locally calculates $z = z_C \oplus z_S$ to retrieve the final function output.

3.5 Boolean Circuits for Private Set Intersection

Secure computation such as private set intersection can be based on a number of generic Boolean circuits that will be explained in the following section. The figures in Section 3.5 are drawn based on the work in [HEK12].

3.5.1 Private Set Intersection

Two parties are holding the sets $S = \{s_1, s_2, \ldots, s_n\}$ and $S' = \{s'_1, s'_2, \ldots, s'_n\}$ of a known size n where $s_i, s'_i \in \{0,1\}^\sigma$. Each element is unique inside its set and has a length of σ bits, which can be achieved using padding for different sized elements up to a certain bound.

The goal is to find all intersecting elements privately, i.e. without revealing anything about the elements that are not contained in both sets – except their amount.

3.5.2 Sort-Compare-Shuffle Circuits

The *sort-compare-shuffle* design is used for private set intersection and explained in detail for multiple versions in [HEK12]. Assuming the input sets' sizes are a power of 2 and that the sets are sorted, it is possible to implement this design efficiently, requiring only $\mathcal{O}(n \log_2 n)$ element comparisons for the private set intersection of $2n$ inputs.

The parties interested in executing a private set intersection are first required to locally sort their input data. These sorted sets are then obliviously merged in the circuit. The merged list of entries is then filtered for duplicates and the intersecting elements are found. The list of intersections has to be shuffled in order to hide the position of the intersecting elements. Therefore $\mathcal{S}$ choses the permutation bits and puts them into his circuit. He sends his shuffled shares to $\mathcal{C}$, who removes the zero-bit contacts (misses) and sends back only the matching shared contacts.

In the following paragraphs we give a short overview about the circuits that are used.

Switching Blocks

Switching blocks are gates that flip the order of their inputs depending on the status of a control signal. This functionality is shown in Figure 3.1. They can be realized by the circuit depicted in Figure 3.2. If the control signal s is set, the input's order is switched, otherwise the inputs are directly passed through the gate. To switch two σ-bit values, σ `CondSwap` gates can be executed in parallel receiving the same selection signal s. This can be implemented to require the evaluation of only one non-free gate.

(a) Reset State **(b)** Set State

Figure 3.1: Switching Block

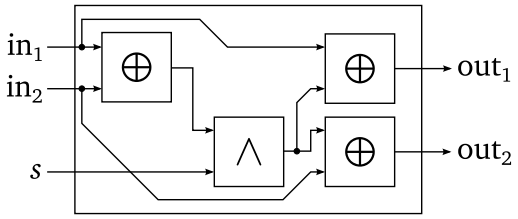


Figure 3.2: Conditional Switch Block `CondSwap`

Sorting Blocks

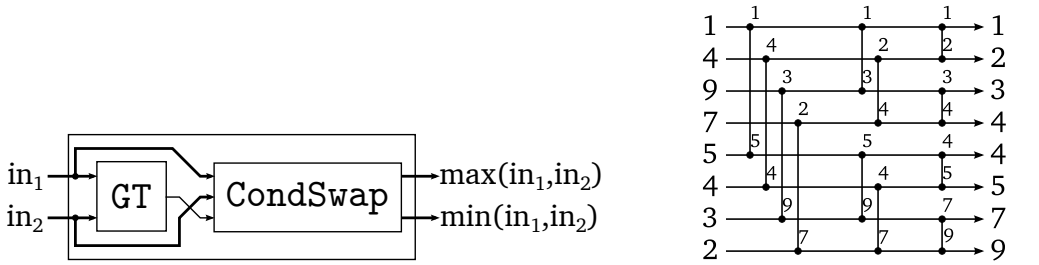
An efficient 2-input sorting block can be built from the `CondSwap` block with one additional greater-than comparison `GT` as depicted in Figure 3.3a. Note that the bold wires represent multi-bit values, while the thin wire is a 1-bit control signal.

This block can be used to implement the merging of a bitonic sequence as shown in Figure 3.3b, where every vertical connection represents one 2-input sorting operation. A sequence is called bitonic if it satisfies Eq. (3.5).

$$x_1 \leq \ldots \leq x_k \geq \ldots \geq x_n, \quad \text{for } 1 \leq k \leq n \tag{3.5}$$

With this construction we can efficiently exploit the condition, that the input sets are already sorted. By concatenating the two input sets, such that one set is sorted ascending while the other one is sorted in descending order, a bitonic sequence is generated.

To merge $2n$ bitonic inputs, $n \log_2(2n)$ sorting blocks are required, if n is a power of 2. The circuit to merge two sets of n sorted σ-bit elements is built of $2\sigma n \log_2(2n)$ non-free gates.



(a) 2-Input Sorting **(b)** Merging a Bitonic Sequence

Figure 3.3: Sorting Block and Bitonic Merging

Duplicate Finding

In order to find duplicate entries, we make use of the fact that those elements are now adjacent. The fully sorted sequence is input into a comparison-based filtering circuit as depicted in Figure 3.4, This circuit consists of multiple `3-Dup Select` blocks and one `2-Dup Select` block as shown in Figure 3.5. This is due to the number of inputs, which is a power of two. The `Dup Select` blocks output either an intersecting element or 0^σ if the inputs do not match. This is realized by a multiplex block `MUX` as depicted in Figure 3.6 that outputs the first input in_1 if its control signal s is not set and in_2 otherwise. `MUX` blocks can be used in parallel for σ-bit values, just like `CondSwap`.

Exploiting the condition that there are no duplicates inside each separate input-set, the cost of this step can be optimized by using as many `3-Dup Select` blocks as possible. In total $(3n - 1)\sigma - n$ non-free gates are required to find all intersecting elements.

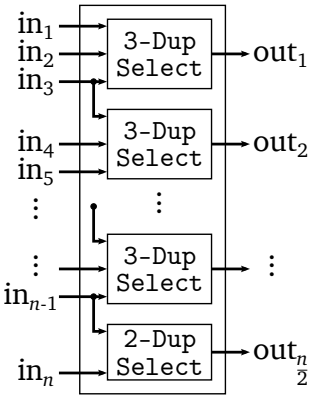
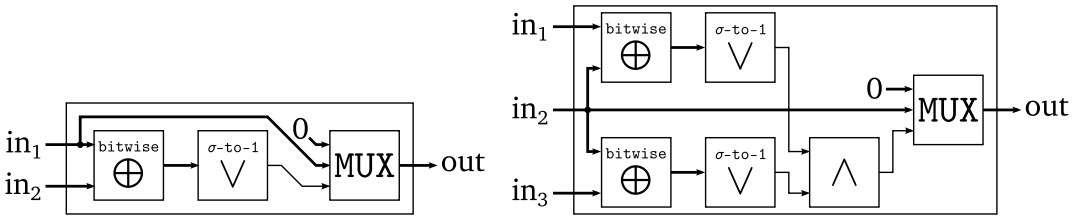


Figure 3.4: Comparison-based Filtering



(a) 2-Dup Select **(b)** 3-Dup Select

Figure 3.5: Duplicate Select Circuits Dup Select

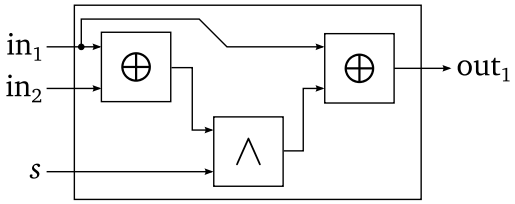


Figure 3.6: Multiplex Block MUX

Waksman Permutation Networks

The Waksman permutation network [Wak68] is a switching network with n inputs and outputs that generates a permutation of the inputs. It is a recursive construction as shown in Figure 3.7 that consists of $\frac{n}{2}$ input CondSwap blocks, two $\frac{n}{2}$-input Waksman networks P_0 and P_1 and $\frac{n}{2}-1$ output CondSwap blocks. If n is a power of 2, it is built from a total number of $n\lceil \log_2 n \rceil - n + 1$ switching blocks.

We use Waksman's construction to shuffle the sorted values of intersecting elements in order to hide their position inside the set. This has to be done since otherwise the position might leak information about the other party's private inputs, such as the minimum or maximum value.

We want to point out that selecting random control bits for the selection blocks does not yield a random permutation of the inputs. Instead, Waksman gives an efficient algorithm that generates the selection bits for an arbitrary given permutation of the inputs in a recursive way.

3.5.3 Bit-wise AND

When the element space is relatively small (e.g. $\leq 2^{16}$), the possession of a set element can be represented by setting a bit in a bit-vector of length 2^σ. A bit-wise AND circuit has to be evaluated on both parties input vectors to output a bit-vector of the intersections. This can be done evaluating 2^σ AND gates in parallel, which is very efficient.

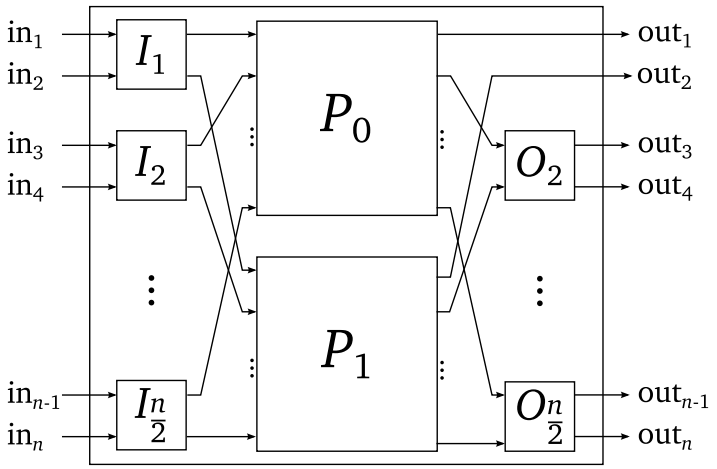


Figure 3.7: Recursive Construction of a Waksman Permutation Network

However, this scheme leaks information about the party's input values and could be abused by setting all input bits to 1. An attacker in the malicious adversary model could use this property to reveal all information given by the other party. Therefore, auditing mechanisms should be included when this scheme is used in practice. This could, for instance, be realized by an extra addition circuit that checks the number of set input bits and only reveals the result if a certain threshold is not exceeded.

3.6 Secure Computation and Applications

Secure computation is a growing field of research that developed rapidly in the recent years. We want to point out some work that is focused on practical SFE applications.

Fairplay [MNPS04] is a two-party SFE framework based on Yao's GCs that was one of the first to demonstrate the practical feasibility of generic secure computation protocols. It is the foundation for many implementations as high-level descriptions of algorithms can be compiled into Boolean circuits by the framework.

Next to the details on the aforementioned circuits, [HEK12] point out that carefully designed generic SFE protocols can be as efficient as custom-tailored protocols. Generic protocols have the general advantage of being modular and extendable. The authors also give a performance evaluation of their implementation. They implemented bit-wise AND circuits and their sort-compare-shuffle circuit using Yao's GC protocol and the OT extension technique on desktop computers. Their work is based on the FastGC framework [HEKM11]. FastGC is an optimized version of Fairplay that significantly improved performance using optimizations such as pipelining.

In [HCE11] the authors explore the possibility of running secure computation schemes on smartphones. They demonstrate the feasibility on Google Nexus One smartphones implementing several schemes. They identify the processing power instead of the bandwidth as the main limiting factor to a fast execution of SFE on mobile devices. With our results we are able to show that this isn't universally true and highly depends on the underlying protocol instance.

Another Fairplay-based example is [CMSA12], where the authors proposed applications for the context of a social network on mobile devices and implemented a mobile interest cast. All previous work of in the field of generic SFE on mobile phones that we know is based on Yao's GC protocol whereas our approach is based on the GMW protocol.

[CHK+12] shows an implementation of the protocol of Goldreich, Micali and Wigderson based on both Boolean and arithmetic circuits using on-line marketplaces as an example in a desktop environment.

In [SZ13], the authors significantly improved the performance of the GMW protocol and pointed out several advantages over Yao's GCs.

Token-based SFE is the setting of the work of Järvinen et al. [JKSS10], who implemented a scheme that allows for the creation of a GC on a HW token that is held by a client in order to reduce the workload on a server.

A realization of an anonymous credential protocol based on Java Card SCs is given in [BCGS09].

There is further related work that focuses on computing OT on HW tokens [GT08; Kol10; DSV10]. Note, however, that these works require the evaluation of symmetric cryptographic primitives per oblivious transfer to be evaluated, which cannot be pre-computed without interaction between both participants.

4 Protocol Design

This chapter explains the design decisions we made for our SFE setting. We base this chapter on the information gained from related research and extend it by our own ideas.

4.1 Secure Function Evaluation on Mobile Devices

The privacy of the user's inputs in SFE schemes comes with the cost of high complexity of the calculations in order to protect the sensitive data. Yao's GC protocol was the first work that was practically realized after it was historically viewed as too inefficient for practical applications, even on powerful desktop computers. The GMW protocol became an alternative and was shown to be truly practical. However, both protocols rely on the complex evaluation of a large number of symmetric cryptographic operations, such as AES or SHA. In case of Yao's GCs they stem from the generation of the garbles tables, while in the GMW protocol they are due to the large number of OTs that have be evaluated. In the following we focus on the GMW protocol, since it holds several advantages over Yao's GCs for deployment on mobile devices: It enables us to equally distribute the workload among the protocol participants. It also requires less memory per gate in the Boolean circuit and allows us to shift all usage of symmetric primitives into a setup phase that can be pre-calculated [SZ13].

However, this pre-calculation has to be done in a trusted way, since it would otherwise break the security of the protocol. Therefore, our main idea is to outsource this expensive setup phase to a partially trusted HW token. This token guarantees that the calculations are done securely, as it is tamper-proof. The setup phase can be executed at any time before the SFE starts. This allows us to pre-compute trusted data, which is then immediately available during the SFE phase, thus making it faster.

This HW token cannot be used as generic trusted third party to simply execute the secure computation on both parties' inputs as its storage capacity is too limited to hold typical inputs, let alone process them. With our design, we offer a generic approach that allows the evaluation of arbitrary functions that can be represented as Boolean circuits. This feature cannot easily be realized on a SC with only a few kB of RAM and very limited non-volatile storage. It would require external memory that could be secured by using symmetric cryptography, thus introducing a certain overhead.

In the following section, we give details on our SFE setting, explain the security assumptions we made and the protocol we based on these assumptions.

4.1.1 Adversary Model

Adversaries in a SFE context refer to the protocol participants themselves rather than to external entities like man-in-the-middle attackers. Those standard threats are prevented by common cryptographic techniques, while internal adversaries have to be considered during the design and construction of SFE protocols.

This thesis focuses on the *semi-honest* adversary model that is also called *honest-but-curious* or *passive* adversary model. It refers to participants that follow the given protocols but try to learn additional information from observed messages. This security model is relatively weak but allows designing very efficient schemes and is widely used.

Malicious or *active* adversaries are stronger adversaries that can deviate from the protocol steps without restrictions in order to learn secret information or influence the computation results. Protocols that are secure against these adversaries are naturally harder to construct and significantly more expensive to compute.

The *covert* adversary can also actively deviate from the protocol, but is detected by honest participants with a certain probability (e.g. $\frac{1}{2}$) and is therefore weaker than the malicious one. This model can be applied to a lot of practical contexts where parties consider cheating, but can't afford being detected since the loss of trust and the embarrassment would overweigh the profits of cheating.

Another problem that has to be considered in multi-party SFE scenarios are colluding participants that are working together and therefore having an advantage over honest parties.

4.1.2 Setting and Requirements

This thesis focuses on a two-party SFE scenario with the two participants denoted as client $\mathcal{C}$ and server $\mathcal{S}$. This setting is extended by a HW token under direct control of $\mathcal{S}$ that is referred to as SC. $\mathcal{S}$ is participating in the protocol with private input x while $\mathcal{C}$ provides y as shown in Figure 4.1. If both participants have access to a SC the roles of $\mathcal{S}$ and $\mathcal{C}$ can be assigned arbitrarily.

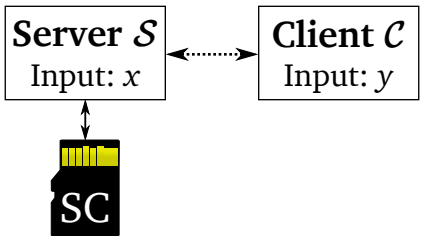


Figure 4.1: SFE Scenario

Computation

Server $\mathcal{S}$ and client $\mathcal{C}$ are mobile devices, such as smartphones that are approximately equally powerful. Their computational power is weaker than the performance of typical desktop computer systems but significantly stronger than the SC. Complex long-term calculations should be avoided since $\mathcal{S}$ and $\mathcal{C}$ are battery-powered. Both have access to a sufficient amount of RAM and persistent storage. The only notable difference is $\mathcal{S}$'s direct access to the SC during the protocol evaluation.

The SC has very limited computational power and only a small amount of persistent and non-persistent memory that is protected against tampering. While arbitrary computations would be possible, we limit the functionality of the SC to the single purpose of generating MTs (cf. Section 3.3) and securely distributing them. We want to point out that the SC does not take part in the actual secure computation, it only supplies additional data.

All parties, including the SC, are capable of calculating symmetric, as well as asymmetric cryptographic functions up to certain size limitations.

Communication

$\mathcal{S}$ and $\mathcal{C}$ communicate through a high-bandwidth channel, such as wireless or cellular networks. Depending on the distance and connection quality, the delay typically varies between less than 10 ms (Wi-Fi / Wi-Fi Direct) and more than 100 ms (trans-atlantic internet).

The SC and $\mathcal{S}$ have a direct physical connection with a low latency but also a low bandwidth. There is no immediate connection between $\mathcal{C}$ and the SC; all communication must be forwarded by $\mathcal{S}$.

Trust

All participants trust the SC as a third party to work as intended, generate data as specified and not collude with other parties. However, the users do not want to share their private data with the SC. Therefore we call the SC *partially trusted*. Since it is hardened against attacks, the participants can assume it to be non-manipulated. The SC can provide a signed certificate to authenticate itself. This certificate has been issued by a trusted Certificate Authority (CA).

$\mathcal{C}$ and $\mathcal{S}$ mistrust each other with regard to private data, however $\mathcal{C}$ relies on $\mathcal{S}$ to properly communicate with the SC, as specified in the adversary model (cf. Section 4.1.1). Neither $\mathcal{S}$ nor $\mathcal{C}$ can authenticate themselves. $\mathcal{S}$ and $\mathcal{C}$ are not required to know each other before the protocol evaluation.

Requirements

It is crucial that the overall runtime of our protocol is short enough to be actually used in practice. Users do not want to wait more than a few moments before they receive a computation result. Therefore, we aim to achieve runtimes of less than 10 seconds for typical input sizes. This is only possible if the implementation adapts to the restrictions in memory and processing power on the mobile devices and intelligent pre-calculation is done. This also addresses the issue of limited power supply, since mobile hardware is typically battery-driven.

We also intend to focus on regular off-the-shelf HW equipped with out-of-the-box software that should run without making modifications to the operating system.

4.2 Multiplication Triples

The previously mentioned pre-calculation on the SC is realized by calculating MTs and securely distributing them amongst the protocol participants.

The theoretical concept and the application of MTs is explained in Section 3.3 and 3.4.2. Depending on the number of AND gates that have to be evaluated during the secure computation a large amount of MTs has to be generated. This task is outsourced to a partially trusted HW token with limited storage capacity, limited processing power and limited amount of write-accesses to the persistent memory. Thus, we have to come up with a solution that is adapted to the restrictions of our HW.

The amount of MTs to generate is not known until the secure computation starts where $\mathcal{S}$ and $\mathcal{C}$ agree on the functionality to compute as well as the sizes of their input data. Due to the slow computational speed of the SC it is not possible to generate large amounts of MTs upon request. Instead they are pre-calculated and must be stored to be distributed later. Storing big amounts of MTs, however, is not possible, as the SC's memory is very small in comparison to the memory of the protocol participants. Therefore, we make use of two keyed PRFs that generate pseudo-random bits from starting values, the seeds. We need two separate PRFs, since the seeds distributed to $\mathcal{C}$ and $\mathcal{S}$ have to be independent and both parties must only learn their own MT shares, derived from their seeds.

Only the PRF seeds have to remain on the SC and can later be queried, allowing re-calculation of the same MTs by $\mathcal{S}$ or $\mathcal{C}$. Since the MTs are not entirely random, but follow Equation (3.1), only five of the six shares can be taken from the PRF while the sixth share $c_{\mathcal{S}}$ has to be calculated according to Equation (4.1).

$$c_{\mathcal{S}} = \big((a_{\mathcal{S}} \oplus a_{\mathcal{C}})(b_{\mathcal{S}} \oplus b_{\mathcal{C}})\big) \oplus c_{\mathcal{C}} \tag{4.1}$$

This calculation is done on the SC, but the storage capacity is still not large enough to hold sufficient amounts of $c_{\mathcal{S}}$ shares. Therefore, the $c_{\mathcal{S}}$ shares are sent to $\mathcal{S}$ that controls the MT generation process. $\mathcal{S}$ receives segments of concatenated $c_{\mathcal{S}}$ bits to store them locally in a sufficiently sized long-term memory. Note that $\mathcal{S}$ only receives shares that are intended for him to use and none of $\mathcal{C}$'s shares are seen or stored by $\mathcal{S}$.

4.2.1 Multiplication Triple Sets

Each PRF can create an arbitrary amount of random bits to generate MTs, however, the MTs can only be used once to mask private data and have to be discarded after they have been used. Therefore, we create certain amounts of MTs in so called MT sets. Those have a specific size of 2^n MTs, where n is chosen in accordance to the HW restrictions. MT sets are generated from unique seeds to ensure independence from other MT sets. $\mathcal{S}$ controls the generation of MT sets, assigns a unique ID to each set and also choses their size by starting a certain amount of PRF iterations on the SC and storing the $c_{\mathcal{S}}$ shares he receives as response. This process can be seen as an overview in Figure 4.2, where an ID is input into `SeedGen` once to calculate the MT set seeds. Those are then transferred into `MTSetGen` that is called multiple times with the same seeds to create the desired output length.

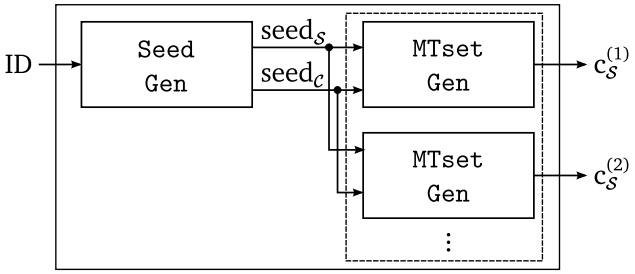


Figure 4.2: Overall MT set Generation

The overall required amount of MTs is created by concatenating multiple MT sets. Since MT sets come in fixed sizes of a power of 2, an efficient creation of arbitrary amounts of MTs is possible, without wasting pre-calculated MTs. This can be achieved efficiently by looking at the binary representation of the number of required MTs N and concatenating MT sets of size 2^i for every set bit at index i in N. The number of required MT sets corresponds to the Hamming weight d_H of N. An Example is given in Equation (4.2) for an amount of 2336 MTs that can be constructed from three MT sets of sizes $2048, 256$ and 32.

$$2336_{\text{dec}} = 100100100000_{\text{bin}} = 2^{11} + 2^8 + 2^5 \tag{4.2}$$

SeedGen

To further reduce the required storage capacity and to assure independence of the MT sets we use a block cipher like AES in CTR mode to derive the PRF seeds from two fixed master secrets, one for $\mathcal{S}$ and $\mathcal{C}$ each. The master secrets and the initial value of the seed counter are chosen randomly during the setup phase of the applet. This setup occurs once when the card issuer installs the applet on the SC, as depicted in Figure 4.3. For each new MT set the monotonic seed_ctr value is increased by one and then encrypted with the corresponding master secrets as keys to generate the MT set seeds as shown in Figure 4.4. The supplied ID is used as an address to store the used seed_ctr value in the persistent SC memory and to allow querying the value at a later point in time. When `SeedGen` is executed, the MT set counter is always reset to zero to ensure a defined starting point for the PRF.

The SC has to store two values per MT set: A unique ID that might be re-used as soon as a MT set has been consumed and the seed counter value seed_ctr to allow re-calculation of the PRF's seeds at a later point in time. The number of MT sets that can be stored like that has to be chosen in accordance to the available HW.

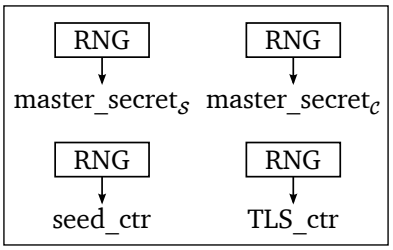


Figure 4.3: JC Applet Setup

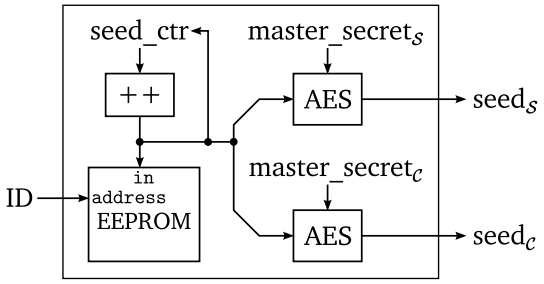


Figure 4.4: `SeedGen`

Additional reduction of the memory requirement could be achieved if a shorter seed counter is used. The required block size can be filled up with a fixed random prefix. However, it has to be evaluated if this would introduce the possibility for related-key attacks or similar weaknesses.

MTSetGen

The MT set seeds are used in `MTSetGen` as AES keys that encrypt a separate 16 byte MT set counter. This counter is independent from the seed counter seed_ctr used in `SeedGen` and starts from zero for each new MT set. This is necessary in order to have a fixed starting point and allow re-calculation of the PRF and does not weaken the security, since different seeds are used as keys for the ciphers. The querying application can theoretically request MT sets of arbitrary length. The upper limit is reached when the counter would overflow back to zero. This happens after 2^{128} iterations. The same MT set counter is used for both PRFs, which is allowed due to their different and unrelated keys. Figure 4.5 depicts how c_S is calculated using five random shares generated from two separate PRFs. Note that $\texttt{PRF}_S$ and $\texttt{PRF}_C$ have a different number of inputs and outputs as shown in Figure 4.6.

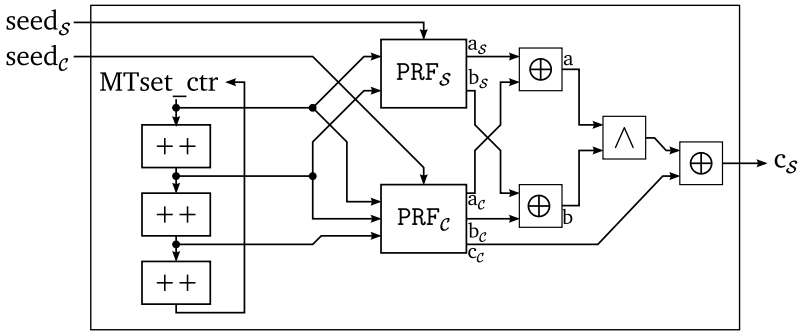


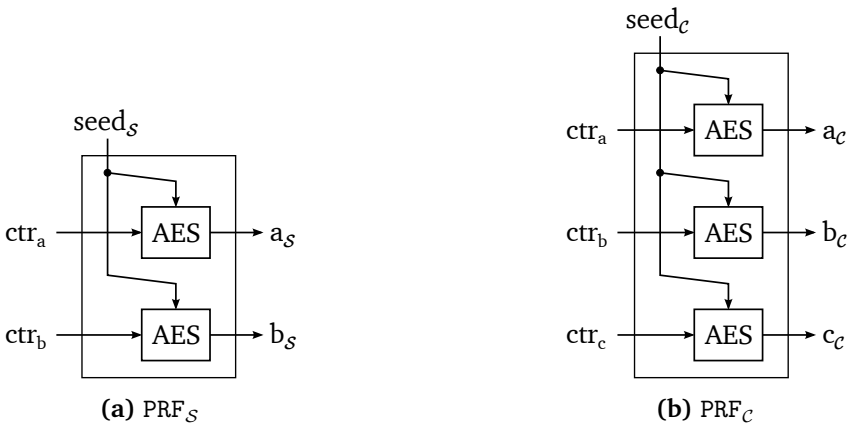
Figure 4.5: `MTSetGen`

Since a block cipher such as AES is used in the PRF, shares should not be generated in a bitwise fashion, but instead larger segments of one type of shares, e.g. a_S, can be generated at once. The actual size of these share segments has to be a multiple of the cipher's block size and can be chosen to fit to the available hardware. Thus, unnecessary bit-operations can be avoided and the performance is maximized. We give details on that part in Section 5.3.4.

Querying MT Sets

To retrieve the seeds and re-generate the MT sets at a later point in time, they have to be queried with their corresponding ID by $\mathcal{S}$. Then the previously stored seed counter is read from memory and used to generate the seeds for either $\texttt{PRF}_S$ or $\texttt{PRF}_C$. The query of seeds for $\mathcal{S}$ is depicted in Figure 4.7. The seeds for $\mathcal{S}$ are transferred in plain text.

The seed values intended for $\mathcal{C}$ have to be encrypted in order to be unreadable for $\mathcal{S}$. This is done using a block cipher like AES in CBC mode. Before they leave the SC to be forwarded from $\mathcal{S}$ to $\mathcal{C}$, both the encrypted seeds and the IDs are authenticated with a MAC as shown in Figure 4.8. The key k_{MAC} is used for authentication while the key k_s and the initialization



(a) $\mathrm{PRF}_{\mathcal{S}}$ **(b)** $\mathrm{PRF}_{\mathcal{C}}$

Figure 4.6: MT PRFs

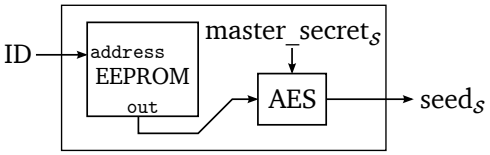


Figure 4.7: Query of Seeds for $\mathcal{S}$

vector IV are used for encryption. Details on the encrypted and authenticated seed transport follow in Section 4.3. Once a seed for $\mathcal{C}$ has been queried and was sent out through an encrypted connection it is ensured that the corresponding seed_ctr value is erased from memory as it is overwritten with 16 zero bytes.

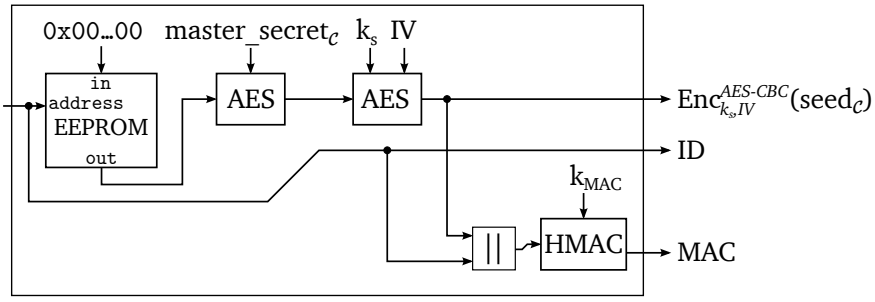


Figure 4.8: Query of authenticated and encrypted Seeds for $\mathcal{C}$

4.3 TLS-based Seed Exchange Protocol

We need a way to securely transmit the MT set seeds from the SC to $\mathcal{C}$ via $\mathcal{S}$. Thus, we need a protocol to securely transport a small amount of data between two parties through an unsafe channel.

In our scenario it is clear that $\mathcal{S}$ has all capabilities of an active man-in-the-middle attacker: He can read, alter, delay, replay or discard messages between $\mathcal{C}$ and the SC or even send new ones. For our setting we assume that $\mathcal{S}$ does not discard messages, since he is interested in successfully finishing the protocol with $\mathcal{C}$. All other attacks are possible and should be detected and therefore be prevented by the protocol. This assumption is stronger than the *semi-honest adversary model* that we use in the SFE part. Therefore this protocol for seed exchange could potentially also be used in the *malicious adversary model* (cf. Section 4.1.1).

The SC holds an asymmetric key pair whose public key is signed by a trusted CA. We assume that all potential users know and trust the CA and therefore are able to verify the validity of the SC's public key. Users are not certified by the CA and do not own a signed public key. Therefore, we require a key exchange protocol that allows one-way authentication of the SC and has low complexity in order to be calculated on the SC.

After carefully evaluating several protocols for key agreement and key exchange based on information from [BM10], we decided to use TLS in its most recent version 1.2 [DR08]. This is due to the fact that TLS and its predecessor SSL are widely used and have gone through an enormous amount of security analysis. There are some recent attacks on TLS 1.2, e.g. [DR11; AP13] that usually affect certain types of ciphers or paddings. These attacks have been evaluated and should not affect our specific instance of the protocol.

4.3.1 TLS-based Key Exchange

For the background of the TLS protocol and the standard definitions see Section 2.2.1.

We implemented a slightly modified version of the TLS handshake in Protocol 4.1, that is specifically designed for our scenario without weakening the security of the protocol. The main limiting factor is the SC, since its computing and storage capacities are limited. The SC acts as the TLS server, while $\mathcal{C}$ has the role of a TLS client. We are making use of server authentication while $\mathcal{C}$ is not authenticated, since he does not own a public key certificate. We omitted all optional messages as they are not applicable in our context. We also chose fixed parameters, ciphers and hash functions in order to remove overhead from parameter agreement messages.

The used ciphers are 1536 bit RSA with Optimal Asymmetric Encryption Padding (OAEP) and 128 bit AES in CBC-mode using an implicit Initialization Vector (IV) that is generated along with the session key k_s. The HMACs use SHA-256 as hash function according to the

recommendations in [DR08, Sec. 5]. The encryption key sizes have been chosen in accordance to recommendations from [ECR12].

We specify the variables, parameter sizes, ciphers and message digests used in the following protocols and equations as follows:

- R_C - 28 random bytes, chosen by C
- T_C - 4 byte GMT unix timestamp, chosen by C
- R_{SC} - 28 random bytes, chosen by the SC
- N_{SC} - 4 byte nonce, chosen by the SC
- $k_{Pub,SC}$ - Public RSA 1536 key of the SC signed by a trusted CA
- RSA - RSA 1536 with OAEP padding
- PMS - Premaster secret, 48 random Bytes, chosen by C
- MS - Master secret, 48 Bytes, derived from PMS, R_C, T_C, R_{SC} and N_{SC}
- HMAC - SHA-256 HMAC
- k_{MAC} - final 32 Byte HMAC key, derived from MS
- k_s - AES-128 session key, derived from MS
- IV - 16 Byte IV for AES in CBC mode, derived from MS

$$C \rightarrow \text{SC} : R_C \,||\, T_C \tag{1}$$
$$\text{SC} \rightarrow C : R_{SC} \,||\, N_{SC} \tag{2}$$
$$\text{SC} \rightarrow C : \text{Certificate} \tag{3}$$
$$C \rightarrow \text{SC} : \text{Enc}^{RSA}_{k_{Pub,SC}}(PMS) \,||\, \swarrow \tag{4}$$
$$\text{PRF}_{MS}\Big(\text{"client finished"} \,||\, \text{SHA256}\Big((1) \,||\, (2) \,||\, (3) \,||\, \text{Enc}^{RSA}_{k_{Pub,SC}}(PMS)\Big)\Big)$$
$$\text{SC} \rightarrow C : \text{PRF}_{MS}\big(\text{"server finished"} \,||\, \text{SHA256}\big((1) \,||\, (2) \,||\, (3) \,||\, (4)\big)\big) \tag{5}$$

Protocol 4.1: SC TLS RSA Handshake without Client Authentication

There is only one minor modification to the standard in the second message (2) in Protocol 4.1. It lacks the timestamp in our version due to the missing access to a timing device on the SC. We replaced the 32 bit unix timestamp with a 32 bit nonce N_{SC}. The initial value of this nonce is randomly chosen during the installation of the applet. For every handshake the nonce acts as a counter that is monotonically increased by one, allowing for 2^{32} handshakes before its value repeats. The first message (1) from C to the SC contains the timestamp as specified in the RFC, even though its validity cannot be verified by the SC. Checking the validity of

the timestamp T_C, i.e. verifying it is larger than the previously received timestamps, would allow denial-of-service attacks. An attacker could send a very large $\hat{T}_C$ value, thus blocking all legitimate requests that follow and contain valid timestamps that are smaller than $\hat{T}_C$. Ignoring the timestamps and replacing one with a nonce does not affect the overall security of the protocol for two reasons. First, the TLS RFC does not enforce a validity check for the timestamp in the basic TLS protocol and even warns about incorrectly set clocks. Second, the intention of including timestamps is prevention of replay attacks that target repetition of the random numbers R. This protection is still guaranteed by a nonce that is monotonically increased.

However, replay attacks should not apply to our scenario. Replaying messages from C does not bring any advantage for an attacker, since C is intentionally not authenticated and anyone can start his own valid session. If messages are replayed anyway, they will fail due to the mismatch in the PRF outputs in messages (4) and (5).

The SC's certificate is sent separately in message (3). Since it is typically larger than the maximum payload size of one message from the SC, it will be sent in a block-wise fashion in multiple messages. The messages to and from the SC are referred to as Application Protocol Data Units (APDUs). The RSA-encrypted premaster secret and the finished message are combined into a single APDU (4). Similar effects can occur in a TCP-based TLS implementation, however, we are enforcing this behavior in our implementation to use the SC's APDU format optimally. The last two messages' content is fully following the TLS 1.2 specification.

The PRF used to generate the master secret and to authenticate the last two key agreement messages is fully complying with the specification. Arbitrary length PRF outputs can be generated in TLS, concatenating multiple 32 byte outputs of the SHA-256-HMACs. However, we limit our usage to two instances of 32 and 64 byte outputs respectively. To achieve an output length of 64 byte the calculations shown in Equation (4.3) are done.

$$
\begin{aligned}
\text{64 B output} =& PRF_k(\text{label} \,||\, \text{seed}) \\
=& \text{HMAC}_k(A(1) \,||\, \text{label} \,||\, \text{seed}) \,||\, \text{HMAC}_k(A(2) \,||\, \text{label} \,||\, \text{seed}) \\
=& \text{HMAC}_k(\text{HMAC}_k(A(0)) \,||\, \text{label} \,||\, \text{seed}) \,||\, \swarrow \\
& \text{HMAC}_k(\text{HMAC}_k(A(1)) \,||\, \text{label} \,||\, \text{seed}) \\
=& \text{HMAC}_k(\text{HMAC}_k(\text{label} \,||\, \text{seed}) \,||\, \text{label} \,||\, \text{seed}) \,||\, \swarrow \\
& \text{HMAC}_k(\text{HMAC}_k(\text{HMAC}_k(A(0))) \,||\, \text{label} \,||\, \text{seed}) \\
=& \text{HMAC}_k(\text{HMAC}_k(\text{label} \,||\, \text{seed}) \,||\, \text{label} \,||\, \text{seed}) \,||\, \swarrow \\
& \text{HMAC}_k(\text{HMAC}_k(\text{HMAC}_k(\text{label} \,||\, \text{seed})) \,||\, \text{label} \,||\, \text{seed})
\end{aligned}
\tag{4.3}
$$

We use two instances of the above mentioned 64 byte PRF: to generate the 48 byte master secret MS as depicted in Equation (4.4) and to expand the master secret MS in order to obtain the final key material denoted as keyblock as shown in Equation (4.5). The 32 byte PRF is used in the last two handshake messages, however, the result is truncated to 12 byte according to specification before it is sent to the other party.

$$MS = \text{PRF}_{PMS}(\text{"master secret"} \,||\, R_C \,||\, T_C \,||\, R_{SC} \,||\, N_{SC}) \tag{4.4}$$

$$\text{keyblock} = \text{PRF}_{MS}(\text{"key expansion"} \,||\, R_{SC} \,||\, N_{SC} \,||\, R_C \,||\, T_C) \tag{4.5}$$

The calculated 64 byte keyblock is then partitioned and the final MAC key k_{MAC}, symmetric session key k_s and initialization vector IV are selected as shown in Equation (4.6).

$$\begin{aligned} k_{MAC} &= \text{keyblock}[0:31] \\ k_s &= \text{keyblock}[32:47] \\ IV &= \text{keyblock}[48:63] \end{aligned} \tag{4.6}$$

4.3.2 Seed Transport Protocol

Once the key agreement is successfully finished we are able to transport the needed MT set seeds as shown in Protocol 4.2. We are able to send up to 14 MT set seeds from the SC to C in a single APDU. The seeds are encrypted with AES-CBC using the session key k_s and initialization vector IV from the key agreement protocol. The IDs are determined in the local MTSetService that runs under control of S. Therefore all IDs are transmitted unencrypted. The whole message, including the plaintext IDs, is authenticated by the SC with a HMAC using the previously derived HMAC key k_{MAC}. Since our maximum APDU size is limited to 256 bytes, we only use the first half of the SHA-256 HMAC, i.e. we truncate the output to the most significant 16 bytes. This allows us to maximize the number of seeds that can be transported in a single message. Truncation of the HMAC output to half of its original length can be done without significantly weakening security according to [KBC97].

$$S \rightarrow \text{SC} : \text{ID}_1 \,||\, \ldots \,||\, \text{ID}_n \tag{1}$$

$$\begin{aligned} \text{SC} \rightarrow C : \text{Enc}^{AES-CBC}_{k_s,IV}(\text{seed}_1 \,||\, \ldots \,||\, \text{seed}_n) \,||\, \text{ID}_1 \,||\, \ldots \,||\, \text{ID}_n \,||\, \swarrow \\ \text{HMAC}_{k_{MAC}}\left(\text{Enc}^{AES-CBC}_{k_s,IV}(\text{seed}_1 \,||\, \ldots \,||\, \text{seed}_n) \,||\, \text{ID}_1 \,||\, \ldots \,||\, \text{ID}_n\right) \end{aligned} \tag{2}$$

Protocol 4.2: MT Set Seed Transport

We want to stress that we use an encrypt-then-MAC strategy in contrast to MAC-then-encrypt as specified in the TLS record protocol. This does not weaken security as shown in [Kra01], but helps keeping our protocol simple. We can completely eliminate the use of padding for AES, since the encrypted plaintext only consists of MT set seeds and is therefore always a multiple of the AES block size of 16 bytes. This allows us to keep the code complexity as low as possible, since the SC does not natively support any kind of padding for AES

ciphers. Furthermore padding-oracle attacks, such as Lucky13 [AP13] are not applicable to our protocol.

4.4 Protocol Overview

The following section summarizes the protocol steps. A graphical representation is found in Figure 4.9. A solid line represents a direct physical connection, while a dotted line is used for wireless communication. The dashed line symbolizes an authenticated and encrypted connection in both domains.

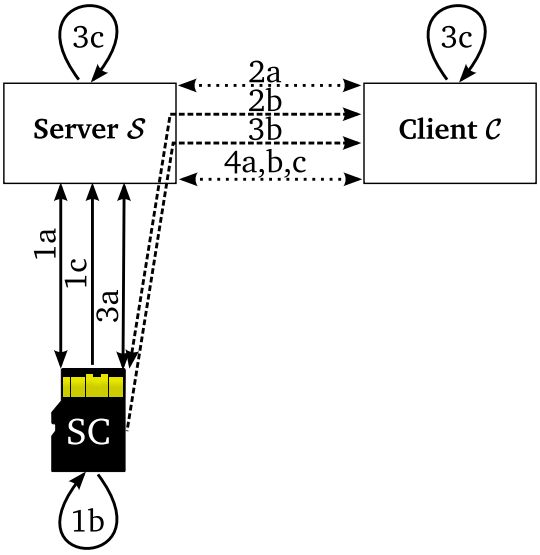


Figure 4.9: Protocol Steps

The setup phase (1*) takes place entirely between the SC and $\mathcal{S}$. In that step, $\mathcal{S}$ connects to the SC (1a) and requests MTs that are generated on the SC (1b). $\mathcal{S}$ stores the $c_{\mathcal{S}}$ shares he receives on his local storage (1c). This phase can happen independently from the actual SFE and can therefore be pre-computed at any time.

The online phase (2*) starts with the connection establishment between $\mathcal{S}$ and $\mathcal{C}$ (2a). The next step is the handshake between the SC and $\mathcal{C}$ (2b) that establishes a secure channel and validates the trustworthiness of the SC.

The next step in the online phase is the transmission and expansion of the seeds (3*). The seeds are requested by $\mathcal{S}$ from the SC depending on the desired amount of MTs. The shares for $\mathcal{S}$ are sent from the SC in plaintext (3a). The shares that are sent to $\mathcal{C}$ are encrypted and authenticated with the keys established during the handshake (3b). $\mathcal{S}$ and $\mathcal{C}$ then expand the received seeds and re-generate the MT sets (3c).

After that, the secure computation starts (4*). The parties mask their private inputs with random shares and exchange these values (4a). The circuit evaluation follows, where the circuit's gates are evaluated iteratively (4b). XOR gates can be evaluated locally, while AND gates require a single message exchange from either party. The final step is the reconstruction of the circuit outputs by exchanging the shares of the final wires of the circuit (4c).

The SC is only involved in the first three phases, while the circuit evaluation phase takes place only between the protocol participants.

4.5 Attacks

We want to consider some generic attacks on our design and outline how they are prevented.

Due to our adversary model, maliciously manipulated messages can be excluded from the list of possible attacks. Passive attacks, however, have to be considered. $\mathcal{C}$ will only get data from $\mathcal{S}$. Thus, $\mathcal{S}$ is in control of the information $\mathcal{C}$ learns. Private data is masked before it is transferred and it is guaranteed by the SFE protocol, that it is not leaked. $\mathcal{S}$ can spy on the data exchanged between the SC and $\mathcal{C}$, but sensitive data is protected by either the SC's private key or the established TLS session keys.

The SC stores a lot of critical information and supplies the protocol participants with sensitive data that affects the masking of private values. It also keeps track of the data from the TLS handshake and the keys derived from it. Therefore it is crucial, that its storage is secure and no information is leaked. This is taken care of by the hardening of the HW against attacks and by the trusted issuer that provided the software of the SC and certified it. Thus, SCs with a valid certificate can be assumed to be trustworthy if the issuing CA is.

Since communication partners of the SC are not authenticated, anyone can initiate a TLS handshake. Also, after a valid SFE session $\mathcal{S}$ can pretend to be $\mathcal{C}$, connect to the SC and request the seeds of the same IDs that the real $\mathcal{C}$ received during the previous session. By learning the same seeds, $\mathcal{S}$ would be able to unmask $\mathcal{C}$'s private values, thus breaking the security of the protocol. This is prevented by the SC that securely erases $\mathcal{C}$'s seeds after they have been queried once through an encrypted channel.

Collusion attacks between $\mathcal{S}$ and $\mathcal{C}$ can also be ruled out. $\mathcal{S}$ and $\mathcal{C}$ naturally do not collude, since otherwise there would be no point in secure computation in the first place. Collusion with the SC is also prevented by the trust in the issuer's certification and software. Communication is only possible through the defined interfaces and commands.

Attacks on the public key certificate of the SC should also not succeed. The certificate is stored inside the secure element and is directly included in the TLS-based handshake. A certificate could be manipulated or replaced with a version from a different SC before it is sent to $\mathcal{C}$. In

that case the TLS handshake will fail on the SC, since either the certificate verification will fail or the PMS value cannot be decrypted correctly.

A possible attack vector is collusion between $\mathcal{S}$ and the HW issuer. This issuer has full access to the SC and can potentially alter the software or knows of backdoors or debug functionality that might expose sensitive data. If $\mathcal{C}$ receives a valid certificate it will consider the SC as trustworthy and rely on the received seeds to generate the MT sets. $\mathcal{S}$ however, could also be able to query the same seeds through a backdoor, thus revealing $\mathcal{C}$'s private data. Collusion between $\mathcal{C}$ and the issuer is harder to realize, since $\mathcal{S}$ controls the channel and knows the content and length of the transferred messages. However, if $\mathcal{S}$ does not keep track of message sizes, it would be possible to trigger a rogue handshake with a certain pre-defined $R_{\mathcal{C}}$ value. As a result the SC could leak $\mathcal{S}$'s seeds within the certificate message that can vary depending on the used HW. This message is also rather long so that a small amount of seed values can be included in an inconspicuous way. This scenario is not impossible, since governments or intelligence agencies might have the power to influence a HW issuer to enable such backdoor functionality and a widely trusted CA to sign it.

5 Implementation

Based on our design decisions from the previous chapter we want to give a detailed insight into our implementation. First, we present the HW and SW we used, explain some useful features and evaluate the limitations we had to deal with. Second, our realization on the given systems is shown and finally we measure the performance of our implementation and analyze it.

5.1 Hardware

The HW we used to develop and benchmark our implementation are state-of-the-art electronics that are publicly available and in case of the Samsung Galaxy S III smartphone also commonly used today.

5.1.1 Samsung Galaxy S III

The Samsung Galaxy S III (GT-I9300) was released in May 2012 and was one of the fastest Android smartphones at that time. It is powered by a 1.4 GHz Samsung Exynos 4412 ARM-Cortex-A9 Quad-Core CPU, has 1 GB of RAM and 16 GB of internal flash memory in the smallest version. It was shipped with the mobile operating system Android 4.0 (Ice Cream Sandwich) and later upgraded to Android 4.1.2 (Jelly Bean). It provides a microSD card slot, Near Field Communication (NFC), Wi-Fi Direct and a lot of additional state-of-the-art smartphone technology. The Galaxy S III was one of the most popular smartphones in 2012 [Maw12] and is still widely used today. Note that a clock of 1.4 GHz on a smartphone CPU cannot be directly compared to desktop CPUs, since power consumption limitations, slower bus speeds and smaller caches limit its actual computational power drastically.

5.1.2 Giesecke & Devrient Mobile Security Card SE 1.0

The G&D Mobile Security Card SE 1.0 (MSC) is a secure flash element released and made available for public use in early 2010. It is embedded into a microSD card that additionally contains 2 GB of separate flash memory. We want to point out that the flash memory is isolated from the SC and cannot be directly accessed by it. Both can be accessed from the outside over a controller. Depending on the incoming command it communicates with either

the secure element or the flash memory. Communication with the SC runs over a protocol according to ISO 7816 (cf. Section 5.2.2).

The secure element is based on a NXP SmartMX P5CD080 microcontroller running with a maximum frequency of 10 MHz. It is equipped with 6 kB of RAM whereof 1750 B are available for transient storage of applet data. The secure element has access to a maximum amount of 80 kB of persistent EEPROM that allows up to 250,000 write or erase cycles and a data retention time of 10 years. The operating system is stored in 200 kB of Read-Only Memory (ROM). The secure element also contains co-processors for 3DES, AES and RSA as well as Elliptic Curve Cryptography (ECC).

The MSC runs the G&D Sm@rtCafe Expert 5.0 operating system that is Common Criteria EAL 4+ certified while the secure element itself conforms to the Common Criteria EAL 5+ certification.

In order to program the MSC, a subset of the functionality of Java Card (JC) 2.2.2 [Sun06] can be used. SW can be installed on the MSC as one or multiple JC applets. It implements the vendor-neutral Global Platform 2.1.1 specification that regulates communication and specifies commands that must be supported. The MSC works with a wide variety of platforms such as Windows, Unix, Blackberry and Android. It is designed to offer various cryptographic and security functions. Amongst others, AES, RSA with up to 2048 bit, DES and DSA are available. The hash functions MD5, SHA-1, SHA-256 and RIPE-MD160 are integrated as well.

While the MSC was available and actively supported for almost two years, its manufacturer Giesecke & Devrient Secure Flash Solutions, a joint venture of G&D and the Phison Electronics Corp. does no longer exist since 2013. However, our results can also be applied to similar embedded security solutions on the market.

5.2 Communication

The communication between the parties involved in our setting takes place in two different channels. A wireless Wi-Fi Direct connection between the participants $\mathcal{S}$ and $\mathcal{C}$ with high bandwidth of several MBits/s and a direct physical connection between the SC and $\mathcal{S}$ with a very limited throughput of a few kBits/s.

5.2.1 Wi-Fi Direct

Wi-Fi Direct is a standard for ad-hoc data exchange through IEEE 802.11 Wi-Fi. It is decentralized and does not require a wireless access point to connect multiple devices, making it especially interesting for mobile use where a stationary infrastructure is not always available. It has similar characteristics as regular router-based 802.11 Wi-Fi and supports a connection with typical Wi-Fi properties: a high bandwidth in the order of multiple Mbit/s, low latency

of less than 10 ms and high range of up to 100 m under good conditions. The connection between multiple parties is provided by one device that runs a SW access point and acts as the so called group owner.

The connection between parties is established using Wi-Fi Protected Setup (WPS) to automatically agree on a key to encrypt all traffic using the standard Wi-Fi Protected Access (WPA) encryption based on AES.

5.2.2 Smart Card Communication Protocol

Communication with SCs is defined in the ISO 7816 standard in different parts, specifying amongst others the physical characteristics, message formats and security measures. Messages to and from a SC are referred to as APDUs. The SC is a passive element, meaning that it never initiates communication and all data from it has to be queried by the SC reader. Therefore there are two categories of APDUs: command APDUs to control the SC or request data and response APDUs to reply to a SC reader. Both types have a specific format as defined in ISO 7816-4 and are shown in Table 5.1 and Table 5.2.

Table 5.1: ISO 7816-4 Command APDU Format

Header				Body		
`CLA`	`INS`	`P1`	`P2`	`Lc`	`Data`	`Le`
1 B	1 B	1 B	1 B	0 … 1 B	0 … 255 B	0 … 1 B

Table 5.2: ISO 7816-4 Response APDU Format

Body	Trailer	
`Data`	`SW1`	`SW2`
0 … 256 B	1 B	1 B

The command APDU header contains four bytes defining the exact command meaning. The `CLA` byte is the command class and indicates whether secure messaging is used and if the command is proprietary or standardized. `INS` encodes the instruction number for the given command class. `P1` and `P2` are parameters for the instruction. The APDU body is used to transfer data of up to 255 byte to the smartcard and indicates if a payload is expected as response. However, it is optional and can be left out. When data should be transferred to the SC the `Lc` byte specifies the length of the payload `Data` in bytes. `Le` indicates the expected length of the SC's response.

The response APDU can start with an optional `Data` part of up to 256 byte that is always followed by two status bytes `SW1` and `SW2` that indicate whether the command was successfully evaluated or an error occurred.

There is an extension to the APDU format that allows the `Lc` and `Le` byte to have a length of up to 3 byte, therefore enabling a maximum payload of $2^{24} - 1$ byte for the command and 2^{24} byte for the response APDU. However, the support for this extension depends on the specific SC and the SW used to communicate with it.

5.3 Smart Card Implementation

We created a JC applet based on our design decisions and requirements from the previous chapter. We now elaborate on our implementation on actual hardware. After introducing the available software environment with its restrictions, we explain how we handled these challenges and realized our protocol design on an off-the-shelf SC.

The biggest challenges of implementing SW on a SC are the tight memory restrictions of 1750 B RAM and 75 kB of persistent storage. Additionally the number of write operations to the EEPROM has to be kept to a minimum in order to preserve its limited amount of available write cycles before wearing out. We addressed these issues by following the JC best practice of creating several generic buffers in the RAM and re-using them for different purposes throughout the entire program.

5.3.1 Java Card

The Java Card (JC) standard [Sun06] offers the ability to write software in form of applets for SCs and other small memory devices. JC uses the syntax of the widely used Java programming language and offers a subset of the features of Java. There are several memory and size restrictions as well as the lack of support for multiple threads. Due to the typical SC processor architecture the types integer, float and double are also not available. Garbage collection is very costly and not automatically available, but can be triggered manually. Typically, memory allocation is done during the one-time setup of an applet, whereas dynamic allocation should be avoided. A JC SC can hold multiple applets depending on the available memory.

One advantage of JC is the availability of several high-level functions in a well-defined API. JC was designed for the purpose of managing sensitive data on SCs. Therefore, user data such as encryption keys are stored separately from the operating system and are strictly isolated from other applets. Several cryptographic functions are pre-defined and can be used if the HW supports them. However, low-level access to the underlying hardware, such as cryptographic co-processors, is not possible.

JC offers a Pseudo-Random Number Generator (PRNG) as well as a cryptographically secure random number generator. The PRNG implementation differs from its regular Java equivalent because it does not produce the same deterministic output if it is seeded with the same input.

5.3.2 Certificates

To ensure that a SC is trustworthy and not manipulated, it holds a certificate that is signed by the trusted authority that also issued the card itself and specified its SW. Obviously the card issuer acts as a CA and has to be trusted by anyone that wants to interact with the SC.

The signed certificate can be requested by an application directly from the JC applet on the SC and contains the public key of the SC. This public key is used by external parties in a key agreement protocol based on TLS in order to establish a secure channel. The certificate itself is also input to the key agreement. For details of the implementation we refer to Section 4.3.1. Alternatively, a copy of the certificate could also be placed on the flash memory of the MSC, which would allow easier access to it. This should not introduce any weaknesses in practice, as long as the original stays inside the secure element.

In our implementation, each SC is deployed with its own RSA key pair with a modulus size of 1536 bit and a public key certificate issued by the CA. The keys of the CA that we created for testing purposes have a modulus size of 2048 bit. Those key sizes were chosen based on the recommendations of [ECR12] and because of the limited processing power and messaging restrictions of both the SC and the mobile device.

For performance reasons the RSA decryption is based on the Chinese remainder theorem. Instead of the regular RSA decryption operation with one large exponent and modulus, two modular exponentiations on smaller moduli and smaller exponents are executed.

5.3.3 Seed Exchange Protocol

Our TLS based seed exchange protocol is explained in Section 4.3. In order to make it work on the MSC, we first had to implement the basic features that were not natively supported by the HW token we used.

The HMAC functionality [KBC97] was added to the JC standard with version 2.2.2 [Sun06], which our SC is also based on, but unfortunately there is no native support for it. Luckily our HW supports the SHA-256 message digest, so realizing the HMAC on our desired hash function was straight forward due to its low complexity. However, we want to stress that a software implementation on a generic processor in Java will certainly not reach the performance of a native, optimized C or assembler implementation on a co-processor. Using a native HMAC implementation would yield a much faster execution of the TLS handshake.

5.3.4 Multiplication Triple Generation

We base our implementation of the MT generation on both the SC and the protocol participants on our design decisions in Section 4.2.

We specified that MT sets are sets with sizes of 2^n MTs. We chose $n = 11$ as a lower bound for the size of a MT set since this amount of c_S bits corresponds to the maximum amount of data that can be transferred in one response APDU from the SC. Furthermore, the computation speed of the SC (cf. Section 5.5.1) is sufficient so that this amount of MTs can be generated in well below 1 second. The minimum set size is also a question of overhead that is required for querying, transferring and storing the corresponding MT set data. We chose $n = 24$ as upper bound for the set size in our implementation, which corresponds to 2 MiB of c_S shares. This parameter can easily be increased independently from the SC.

Since we need to store the seed counters on the smartcard for a long time, i.e. independent from the SC's power supply, we have to write them to the EEPROM. We decided on a maximum number of 256 seed counters that can be stored, occupying 4096 byte of persistent memory. This also allows indexing with a single byte value as address. The capacity can be increased if larger HW resources are available.

For the PRFs we chose AES with a key size of 128 bit in CTR mode. The CTR mode as a mode of operation for block ciphers is not natively available on JC 2.2.2 SCs. We created a large buffer that holds 24 counter values of 16 byte each. These values are encrypted as one continuous block in order to create the PRF output. This is done for performance reasons, since encrypting larger blocks of data results in a better performance than crypto operations on multiple smaller inputs. We tested our implementation against a Python script for multiple sizes to ensure the counter was correctly increased and every counter value was only encrypted once under the same key – which is crucial for the security of the CTR mode. Since the encrypted blocks are unique counters the underlying encryption can be realized in ECB mode. This mode of encryption is also used to derive the seeds from the master secrets as shown in Figure 5.1.

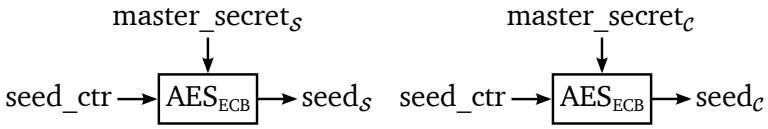


Figure 5.1: Seed generation from the monotonic seed counter. The master secrets are used as AES keys. All values are 128 bit.

The PRFs that generate the MTs are depicted in Figure 5.2. The MT set counter is reset to zero for each new MT set and then monotonically increased for each AES block. For each party the MT set seed is set as the AES key to encrypt the counter blocks. For performance reasons we have chosen blocks with a size of 128 byte. This construction creates a stream of pseudorandom data and can easily be reconstructed by supplying the seed values. Each party receives half of the shares of each MT. Note that the c_S shares are pre-calculated and are

already stored in $\mathcal{S}$'s memory. Therefore, we can skip one counter block during the evaluation of the PRF of $\mathcal{S}$.

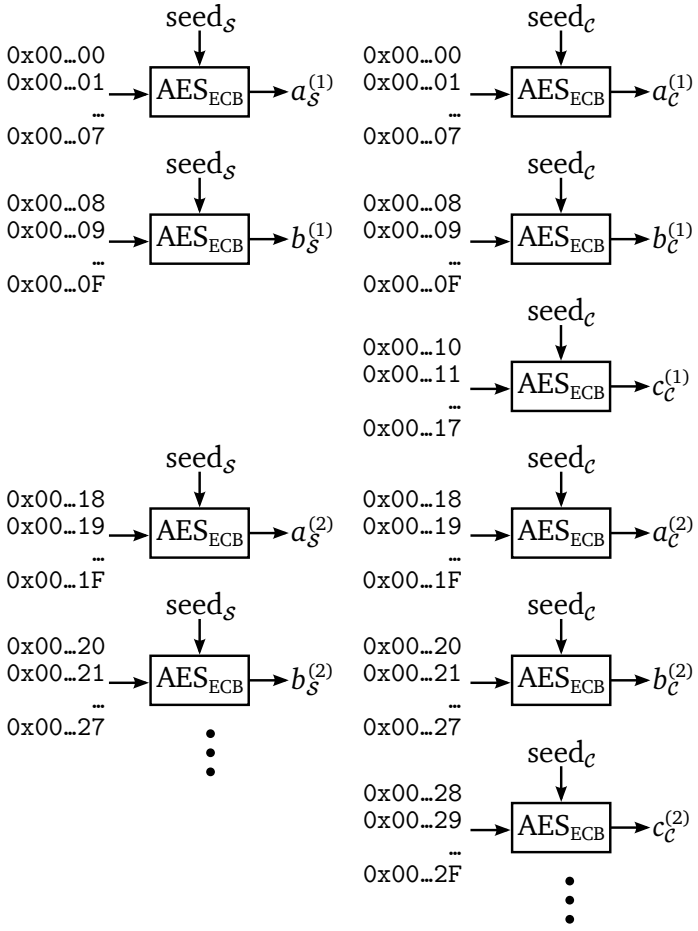


Figure 5.2: MT Set Generation for $\mathcal{S}$ and $\mathcal{C}$. Every input line denotes a 16 byte counter value. a, b and c are generated in blocks of 128 byte each. The seed values are used as AES keys.

The sizes of the master secrets and the MT set counter have been chosen as 16 byte. This matches the AES 128 block size and guarantees that the counter does not repeat until after 2^{128} iterations, which is certainly longer than the SC's lifetime.

5.3.5 APDU Commands for the Java Card Applet

There is a fixed set of APDUs for instructions that can be used to communicate with the JC applet. We want to explain every instruction and give some details to the implementation. We are explaining the instruction syntax in accordance to Section 5.2.2.

We defined the `CLA` byte as set to `0x90` for all instructions. This value specifies that all APDUs are following the structure defined in ISO 7816 and that the commands exchanged are proprietary and can therefore be defined freely. The parameter byte`P1` is the ID of the MT set in certain commands and ignored otherwise. `P2` is unused and must therefore be set to `0x00`.

The following list describes the available instructions ordered by their `INS` byte.

`0x10` Starts the generation of a new MT-Set. The MT set counter is reset to zero before the PRF evaluation and 256 bytes of c_S shares are returned. The parameter `P1` must be set to the ID of the MT set.

`0x12` Continues the previously generated MT-Set that has been started with `INS 0x10` and also returns 256 bytes of c_S shares. Note that this command must follow to a `0x10` or `0x12` instruction.

`0x14` Returns a single seed_S for the ID specified in parameter `P1`.

`0x16` Returns multiple seed_S values for the IDs specified in the body of the APDU. The field `Lc` must specify the number of IDs requested.

`0x20` Initiates the TLS key exchange. The `Lc` byte must be set to `0x20` since the payload contains 28 byte R_C and the 4 byte gmt_unix timestamp R_C. This command returns 28 byte R_{SC}, the 4 byte nonce N_{SC} and a 2 byte `short` value specifying the length of the public key certificate in bytes.

`0x22` Finishes the TLS key exchange. The payload is a 192 byte RSA-1536-OAEP encrypted 48 byte PMS and the 12 byte TLS client finished PRF output. Thus, `Lc` must be set to `0xCC`. The instruction returns the 12 byte server finished PRF output. This command must immediately follow to either `INS 0x20` or `INS 0x32`.

`0x24` Returns up to 14 encrypted and authenticated seed_S values. The keys used for encryption and authentication must have been established in a successful key exchange in instructions `0x20` and `0x22`. This request overwrites the internal storage of the seed counter for the requested IDs to ensure one time query. The `Lc` field has to specify the number of IDs requested, which are contained in the APDU body.

`0x30` Returns the length of the stored RSA public key certificate in bytes as a `short` value.

`0x32` Returns up to 256 bytes of the public key certificate or an empty APDU if the end is reached. Either `0x20` or `0x30` must be called before the first block is queried, in order to reset the current read position of the certificate and to learn its length.

`0xFE` Returns the version number of the installed applet. In our implementation this value corresponds to the build date of the applet in the format `0xMMDD` in the year 2013.

5.4 Android Apps

This section gives an overview of our Android applications and explains our implementation decisions. The components are depicted in Figure 5.3. Both $\mathcal{S}$ and $\mathcal{C}$ run the same app that offers the SFE functionality and the use cases built upon it, as well as the communication through Wi-Fi Direct. $\mathcal{S}$ runs an additional MT Set Service that manages the creation and request of MTs from the SC, so the high-level application does not require implementing SC specific commands. This also allows for the MT Set Service to be replaced with a different solution. The MT Set Service communicates with the G&D MSC SmartcardService that has been provided by the SC manufacturer and allows connecting to the SC and communicating with it on a low level.

All three components are regular Android apps that could be installed from the market or manually by the user and do require neither root access to the devices nor manipulation of the system image.

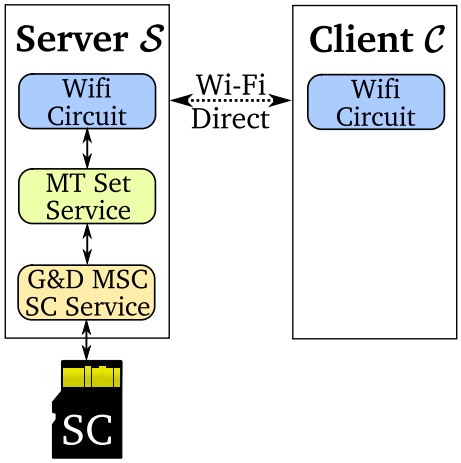


Figure 5.3: Android Software Components

5.4.1 G&D MSC SmartcardService

The MSC's manufacturer Giesecke & Devrient published and supported the MSC Smartcard-Service [SEE13] along with the HW. It is a stand-alone version of the SmartCard API extended by manufacturer drivers for the MSC. The service is deployed as .apk file and installed as a regular app by the user. It supports all commands of the SmartCard API, which is a reference implementation of the SIMalliance Open Mobile API specification. This specification allows communication with a variety of secure elements. However, the regular Smartcard API has to be integrated into the system image, which has to be flashed on the device before it can be used. This procedure should not be done by an average smartphone user, which is why we rely on this separate service.

5.4.2 MT Set Service

We implemented a stand-alone service that can run in the background, taking care of communicating with the SC and managing the creation, deployment and local storage of MTs. It offers convenient, intent-based access for other apps that want to work with MTs. Commands to the service are sent as a single intent that is then translated into one or multiple APDUs to the SC. This simplifies communicating with the secure element for applications. The user has to grant permission for other apps to interact with the MT Set Service during their installation.

The MT Set Service then acts as a layer of abstraction between the SFE application and the SC for several commands. It offers the possibility to query specific amounts of MTs and puts them together from multiple MT sets, taking care of managing already generated MT sets and optionally generating new sets on the fly. Upon request it sends the seeds from the SC to the requesting application and adds information about the set sizes to each seed. Depending on the query it adds either the locally stored $c_{\mathcal{S}}$ shares to its response for $\mathcal{S}$ or queries encrypted seeds for $\mathcal{C}$.

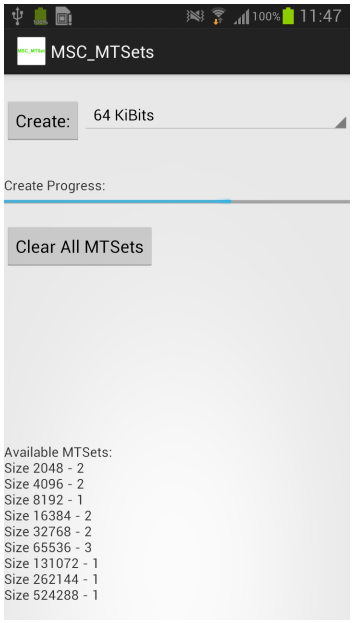


Figure 5.4: MT Set Service GUI Screenshot

Generating new MT sets can be triggered manually by the user or in an automated way. The automatic generation task is started periodically and makes sure that a certain list of MT sets is available. During the generation it saves the received $c_{\mathcal{S}}$ blocks into a file for each MT set. These files are stored in the internal app directory, which is only readable for the MT Set Service. Note that a user or application with root rights can access all internal app directories. Root access is per default not activated on android phones, but can be activated

with little effort on most devices. Then additional encryption of the locally stored shares would be necessary. This could be realized by using authenticated encryption, such as AES in OCB mode with a key that is stored on the SC.

The MT Set Service is also responsible for forwarding the handshake messages between $\mathcal{C}$ and the SC according to the defined protocol, combining several SC APDUs to one response.

While the service does not need user interaction for its main tasks, it offers a simple GUI that allows the user to manually trigger the generation of MTs, get an overview of the available MTs or delete all generated MTs. The GUI is depicted in Figure 5.4. Several settings allow customization of the service's behavior.

We chose to divide the MT set functionality into a distinct service for reasons of modularity and in order to be able to exchange the process of generating MTs. This could be realized, for example, by a trusted SW solution if no SC is available among two users intending to execute a secure computation.

5.4.3 Circuit Evaluation Demo

The circuit evaluation app has several functions, from managing the Wi-Fi Direct connection, to communicating with the MT Set Service and the actual SFE.

The first step is establishing and managing the connection via Wi-Fi Direct. Therefore, both participants have to start the peer discovery, i.e. actively searching for other devices offering a connection via Wi-Fi Direct. Available communication partners are listed as shown in Figure 5.6a. The direct connection is then requested by one user and has to be accepted by the partner within a time limit of 30 s. This is due to the Wi-Fi Direct standard but is actually rather useful in our scenario, since no SFE calculation can be started without the user actively acknowledging it. This part of our application is based on Google's Wi-Fi Direct Demo that is available with the Android Development Tools.

After the connection is established either of the two parties selects a circuit to evaluate, thus fixing the use case and the number of inputs and sends his choice to the other user.

The user with direct access to a SC from now on takes the role of $\mathcal{S}$ while the other one acts as $\mathcal{C}$. If both users have a SC, the roles can be assigned randomly. $\mathcal{C}$ then starts initiating the handshake with the SC in order to agree on session keys. During that phase $\mathcal{S}$ has to act as a gateway between $\mathcal{C}$ and the SC and forward every message to the MT Set Service accordingly.

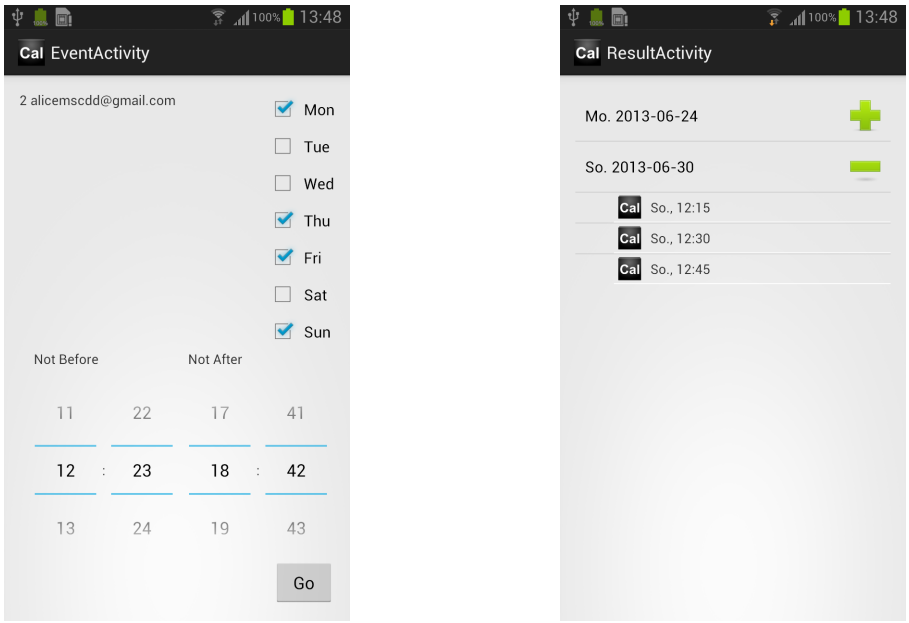
As soon as the session keys between $\mathcal{C}$ and the SC are established, the amount of required MTs is requested from the SC. The sizes of the MT sets and the seeds are sent to $\mathcal{S}$ by the MT Set Service in plain text and to $\mathcal{C}$ encrypted and authenticated with the previously established keys. $\mathcal{S}$ and $\mathcal{C}$ then locally re-generate the MT sets to use them later on in the interactive evaluation of AND gates.

After that the SFE phase starts between $\mathcal{S}$ and $\mathcal{C}$. Both parties prepare their inputs and mask them with random shares. The next step is interactively evaluating the selected circuit as described below. Therefore a pre-generated circuit file is loaded from memory and processed. The participant's inputs are masked with random shares before they are sent to the other party and input into the circuit. The final step after the circuit evaluation is the reconstruction of the results from the circuit outputs.

Our use cases are securely scheduling a meeting and finding common contacts as instances of private set intersection, however, we are not limited to these schemes as the evaluation of arbitrary circuits is possible in our implementation.

Securely Scheduling a Meeting

One of the use cases we implemented is finding a shared free slot in the participant's calendar with a bit-wise AND circuit in order to securely schedule a meeting without leaking information about their schedule. This is realized by dividing one week into time slots of 15 minutes each and finding all the slots that happen to be available in both party's calendars. Each free time slot is represented as a 1 bit, while unavailable slots are set to 0, totaling to 672 bits of information about the availability of the user in the following week. The possible meeting times are found by sending the participant's inputs into a bit-wise AND circuit built from 672 AND gates that will return a logical 1 for the time slots, where both $\mathcal{S}$ and $\mathcal{C}$ are available for scheduling a meeting.



(a) Filtering Input Dates **(b)** The Shared Free Slots

Figure 5.5: Securely Scheduling a Meeting Screenshots

Our app also allows restricting the input to certain days of the week and set a limit for the minimum and maximum time of the day, where meetings can be scheduled. Those restricted time slots are then forced to 0 bits. The filter options are depicted in Figure 5.5a.

The resulting free slots are displayed as a list of dates that is grouped by days as shown in Figure 5.5b. One user can select a time slot, which is then sent to the other user and can be directly added to both calendars.

Unfortunately the construction of using only AND gates would allow a malicious party to set all of his input bits to 1, receiving the complete input of the other participant. This problem is addressed by filtering one's inputs with our app and could further be improved by adding a counter to the circuit. This counter could then be checked against a certain threshold that is set to the maximum number of set bits allowed to input. If a party inputs more free slots than the threshold we can assume it's not a legitimate input and the circuit result will not be revealed. Another possibility to address this problem is evaluating the circuit bit by bit until the first shared free slot is found. Then, an interactive decision can be made to stop the calculation or continue for the next possible date.

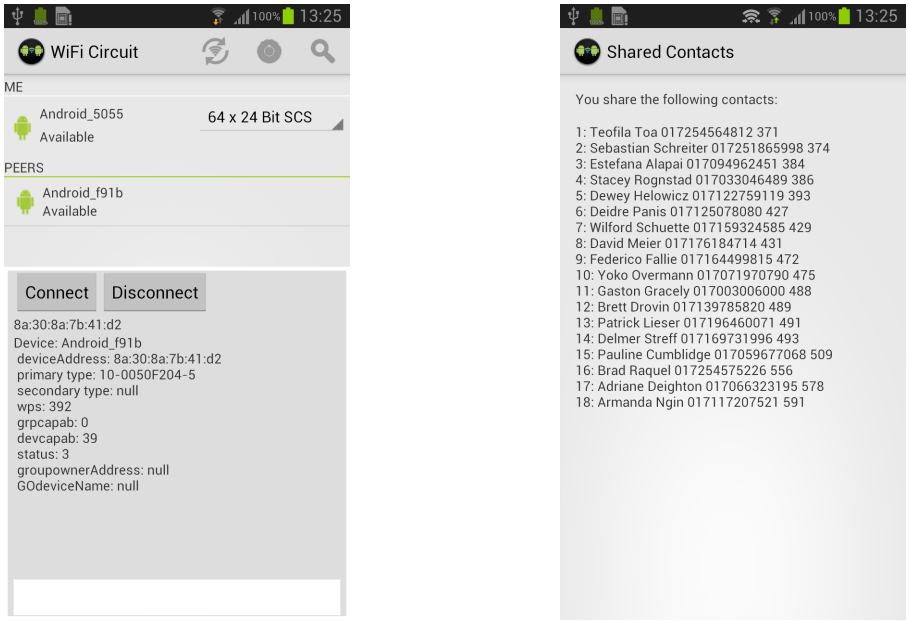
Shared Contacts

The second use case we implemented is finding common contacts as an instance of the private set intersection problem. We based this on a sort-compare-shuffle circuit, similar to the one used in [HEK12] and described in Section 3.5.

First, the parties have to create 24 bit or 128 bit representations of the contacts they want to input, depending on the circuit they chose to evaluate. This can be realized in multiple ways: The calculation of a conventional hash function with the contact's data, such as names, addresses or phone numbers as input can be truncated to the desired size. However, hashing multiple parameters will result in failed comparisons, since not everyone holds the same amount of data about a potentially shared contact. Hashing only the contact's names is prone to spelling errors and might create collisions for very common names. Therefore we chose the mobile telephone number as key to uniquely identify a contact. We didn't make use of a cryptographic hash function since the phone numbers are already a unique identifier. To create the hash value we chose to map the phone numbers to a 24 bit representation by reducing them modulo 2^{24} or representing them as 16 byte number with leading zeros.

The participants have to sort the hashes of the phone numbers before they can be masked and input into the circuit. Additionally, $\mathcal{S}$ has to generate a random permutation and calculate the according bits for the Waksman permutation network.

After evaluating the circuit, $\mathcal{S}$ sends his output shares to $\mathcal{C}$ who is then able to reconstruct the outputs in permuted order. $\mathcal{C}$ removes all non-intersecting contacts (all-zero hashes) from the result, sorts the hashes and shares this final list of matching hashes with $\mathcal{S}$. Both parties can



(a) Wi-Fi Direct Circuit **(b)** List of Shared Contacts

Figure 5.6: Wi-Fi Circuit and Shared Contacts Screenshots

then map the hashes back to their contacts and get a list of the contacts that are contained in both parties' address books as shown in Figure 5.6b.

5.5 Benchmarks

In the following section we benchmark the performance of all our implementations on the hardware introduced in Section 5.1.

We depicted some of the measurements as box plots. In these diagrams the box is used for all values between the first and third quartile, the median value is shown as a horizontal line and the mean value is depicted as $\times$. The whiskers represent the minimum and maximum measured value.

5.5.1 Smart Card Benchmarks

The following measurements have been made on the Samsung Galaxy S III Android smartphones. We measured the time from sending the command APDU until the response APDU was received. Therefore, the values in this section include the transmission time to and from the SC.

We present benchmarks for the MT generation, the TLS handshake and the seed transport as well as generic measurements of the SC's performance.

Generic Smart Card Benchmarks

The communication with the SC is severely limited in terms of delay and throughput. To have a general understanding what that means, we measured the communication characteristics. The values listed here have negligible variance. The shortest possible sequence of APDUs requires 23 ms to be transferred to the SC and back to the reader. This was measured as the average of multiple command APDUs containing only the 4 byte header that is followed by a response APDU consisting of only the two trailer bytes. The maximum throughput that can be achieved, averages at 7.5 KiB/s for large APDUs that are sent to the SC and immediately returned.

We also measured the speed of the block cipher AES that runs on a co-processor on the SC as well as several supported hash functions Therefore, we initialized two byte arrays and ran 200 encryption or hash operations on those arrays, writing the result into same buffer after each iteration. The sizes of the byte arrays were chosen according to the block or input size of the algorithm. Our results are depicted in Table 5.3.

Table 5.3: Crypto and Message Digest Benchmarks on the Smart Card

Algorithm	Input Size	Speed
AES-128-ECB	16 B	2.60 KiB/s
AES-128-ECB	256 B	16.71 KiB/s
MD5	16 B	1.62 KiB/s
SHA1	20 B	1.26 KiB/s
SHA256	32 B	1.22 KiB/s
MD5	256 B	7.33 KiB/s
SHA1	240 B	4.37 KiB/s
SHA256	256 B	2.09 KiB/s

Multiplication Triple Set Generation

The generation of MT sets as described in Section 5.3.4 is split into two different instructions: Creating a new MT set (`INS 0x10`) and continuing the previously generated set (`INS 0x12`) in order to extend its length.

As expected, the create instruction takes slightly longer due to the additional complexity of deriving and storing the set's seed values. The measured values are depicted in Table 5.4 and Figure 5.7. For large MT sets an average generation speed of $\frac{2,048}{0.34453} =$

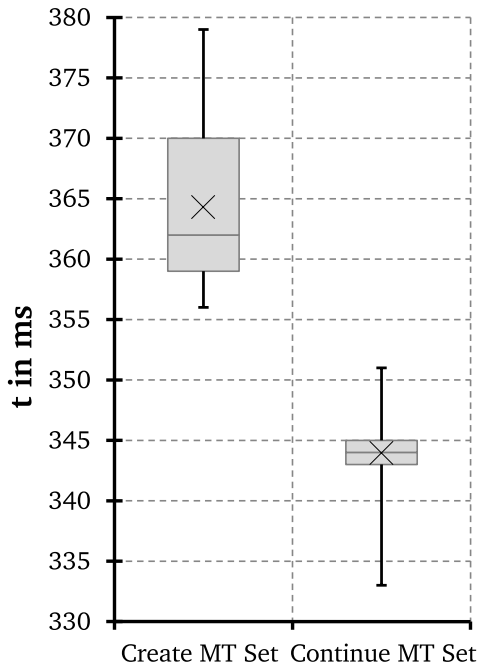


Figure 5.7: SC MT Set Generation Timings, $N = 100$ Measurements

5,944.34 MT sets/s can be achieved as the influence of the slightly slower create instruction amortizes.

Table 5.4: MT Set Generation Instruction Timings, $N = 100$ Measurements

Instruction	Create MT Set	Continue MT Set
Mean ∅	364.29 ms	343.94 ms
Standard Deviation σ	6.14 ms	3.12 ms

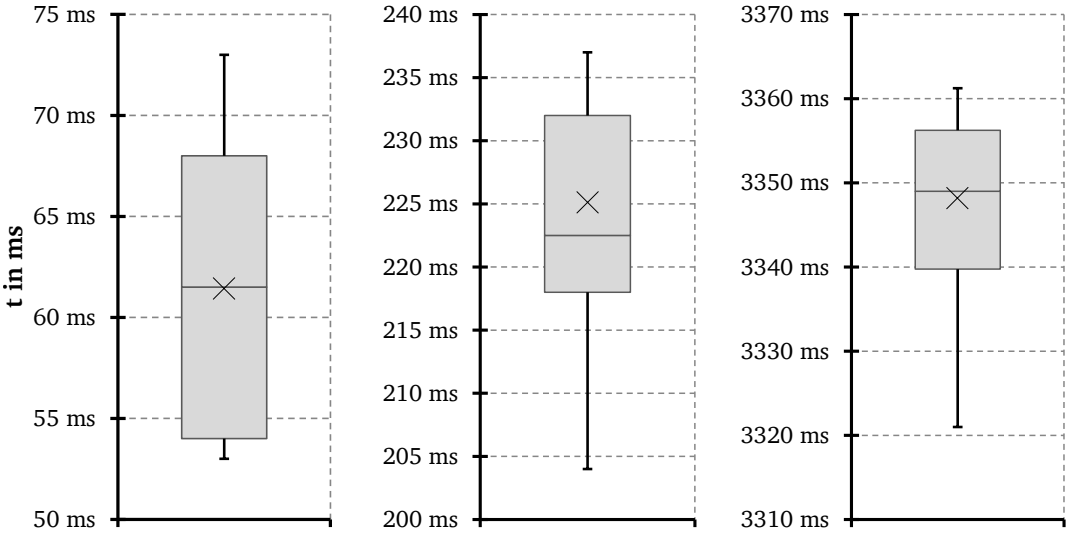
TLS-based Handshake

The TLS-based handshake can be broken down into the three steps shown in Figure 5.8. The first handshake instruction is computationally cheap and requires less than two percent of the total handshake time. The certificate instruction is mainly limited by the communication bandwidth since multiple APDUs are required to transfer the entire certificate. For a future version a copy of the SC's public key certificate could be stored on the MSC's flash memory in order to speedup its query during the handshake and decrease the overall handshake runtime.

More than 92 percent of the handshake time is spent in the final instruction, where the RSA-encrypted PMS value is decrypted, $\mathcal{C}$'s and the SC's PRF outputs are calculated and the session keys are derived.

Table 5.5: SC TLS Handshake Timings, $N = 100$ Measurements

	Handshake 1	Certificate	Handshake 2	Total
Mean ∅	61.42 ms	225.12 ms	3,348.16 ms	3,634.70 ms
Standard Deviation σ	7.02 ms	8.90 ms	11.88 ms	16.54 ms
Runtime %	1.7%	6.2%	92.1%	100%



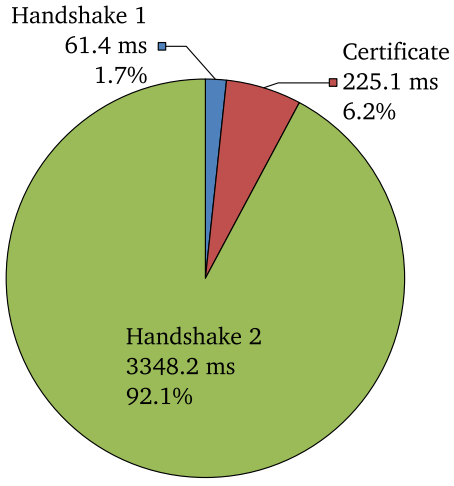
(a) Handshake 1 **(b)** Certificate **(c)** Handshake 2

Figure 5.8: Handshake Message Timings, $N = 100$ Measurements

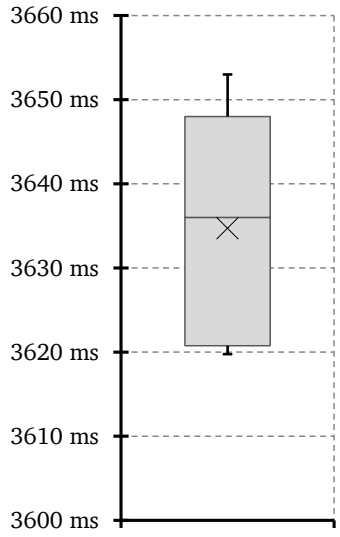
We have analyzed the second handshake instruction in detail to determine what part is the most costly. Therefore, we created three test versions of this instruction where we removed all calls for a certain function that we expected to be costly. All validity checks have been made, but exceptions have been deactivated in order to evaluate the complete command. We created one version without the TLS PRF calls, which removed all HMAC evaluations, but included RSA and the SHA256 evaluation, which would have been used as PRF input. For the second version we only removed the RSA decryption and left PMS at the default value of zero. The third version includes the RSA decryption and a full PRF evaluation but the SHA256 inputs into the PRF have not been calculated. The results are depicted in Table 5.6, where we list the mean values for 20 iterations and calculate the runtime difference by subtracting the new runtime from the regular one. We also listed the runtime percentage of the three components. Note that the missing 9,5% runtime are spent on receiving the encrypted PMS value and transferring data between the buffers and replying with the PRF output.

Table 5.6: Handshake 2 Instruction Components, $N = 20$ Measurements

	without PRF	without RSA	without SHA	Regular
Mean ∅	1,566.01 ms	2,987.15 ms	2,462.17 ms	3,348.16 ms
Standard Deviation σ	6.45 ms	6.98 ms	7.58 ms	11.88 ms
Difference to Regular	1,782.15 ms	361.01ms	885.99 ms	0
Difference Runtime %	53.2%	10.8%	26.5%	100%



(a) Complete SC TLS Handshake, Mean Values



(b) Complete SC TLS Handshake

Figure 5.9: Complete SC TLS Handshake, $N = 100$ Measurements

Seed Transport

We measured the time to query a certain amount of MT set seeds from the SC as specified in instructions `0x14` and `0x24`. The results are shown in Figure 5.10. The values of some characteristic measurements are also depicted in Table 5.7. Note that the plaintext query intended for $\mathcal{S}$ allows up to 16 seeds per APDU while the confidential version for $\mathcal{C}$ can only hold 14 seeds in one message, due to the included IDs and MAC value (cf. Protocol 4.2). The values for 15 and 16 seeds for the encrypted query are the results of requesting two separate APDUs. Thus, two encryptions and MAC calculations have to be executed.

Table 5.7: SC MT Set Seed Query Timings, $N = 20$ Measurements each

Seeds/APDU	$\text{Seed}_{\mathcal{S}}$ $\varnothing$	$\text{Seed}_{\mathcal{S}}$ σ	$\text{Seed}_{\mathcal{C}}$ $\varnothing$	$\text{Seed}_{\mathcal{C}}$ σ
1	24.0 ms	1.5 ms	157.3 ms	4.3 ms
2	38.3 ms	1.7 ms	169.6 ms	2.3 ms
4	53.0 ms	2.0 ms	213.1 ms	2.6 ms
6	67.2 ms	1.5 ms	227.4 ms	1.6 ms
8	82.4 ms	2.2 ms	268.9 ms	6.4 ms
10	96.2 ms	1.8 ms	284.9 ms	1.7 ms
12	111.4 ms	2.0 ms	324.9 ms	4.7 ms
14	126.5 ms	3.0 ms	359.3 ms	2.2 ms
16	157.5 ms	3.3 ms	528.9 ms	3.3 ms

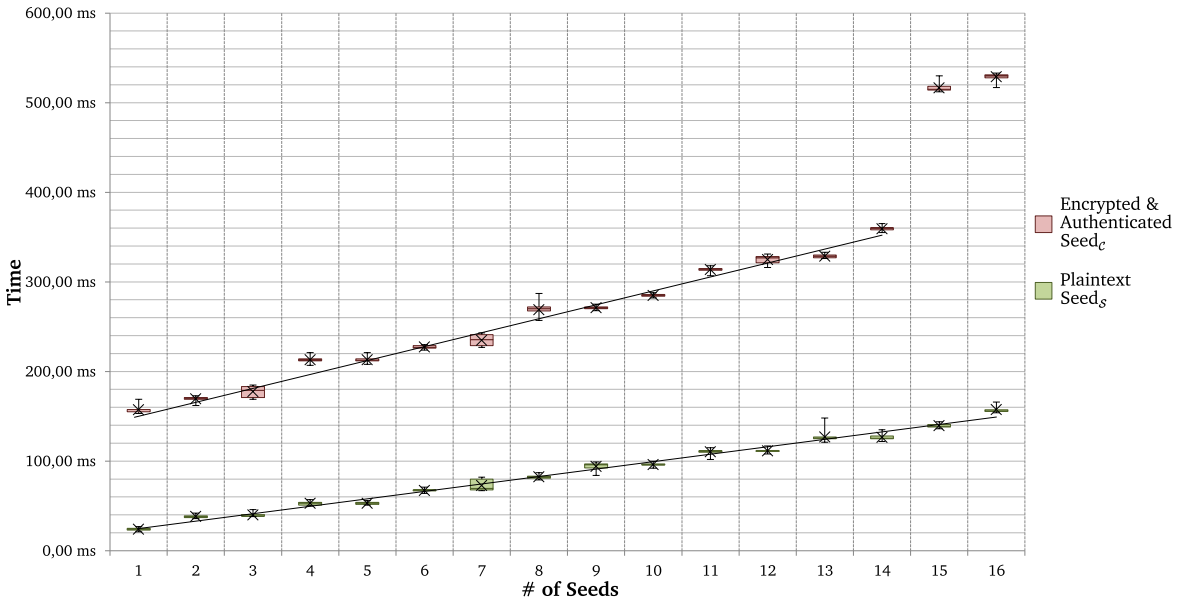


Figure 5.10: SC MT Set Seed Query, $N = 20$ Measurements each

The required time increases linearly with the number of seeds in both cases. However, querying seeds for $\mathcal{C}$ is clearly more expensive. There is a time difference of about 130 ms that is required to initiate the encryption and MAC calculation and the absolute difference increases with the length of the APDU, as more data has to be encrypted and authenticated. This difference is introduced for every encrypted and authenticated APDU.

5.5.2 Android Benchmarks

This section evaluates the performance of the Android applications we developed. We give detailed insight on the re-generation of MT sets, the circuit evaluation and the querying of contact data.

MT Re-Generation on Android

Half of the MT set shares are re-calculated from the received seeds by evaluating the respective AES CTR PRF from Figure 5.2 on the smartphones. The performance is depicted in Figure 5.11 and Table 5.8 for different MT set sizes. Note that the scale of Figure 5.11 is logarithmic. For comparison, we also listes the time it took to compute the same functionality on a desktop computer. We benchmarked the same single-threaded Java implementation on a Windows 8 x64 machine running on a AMD Phenom II X4 945 CPU with 3 GHz with 12 GiB RAM. The runtime difference is approximately a factor of 10. The performance of $\mathcal{S}$'s and $\mathcal{C}$'s PRF on Android is almost identical as it is based on a AES cipher object running the CTR mode natively. This has shown to be significantly faster than implementing similar functionality ourselves. Even if we have to encrypt a certain amount of overhead, it is still faster than re-initializing the cipher object with a new CTR value.

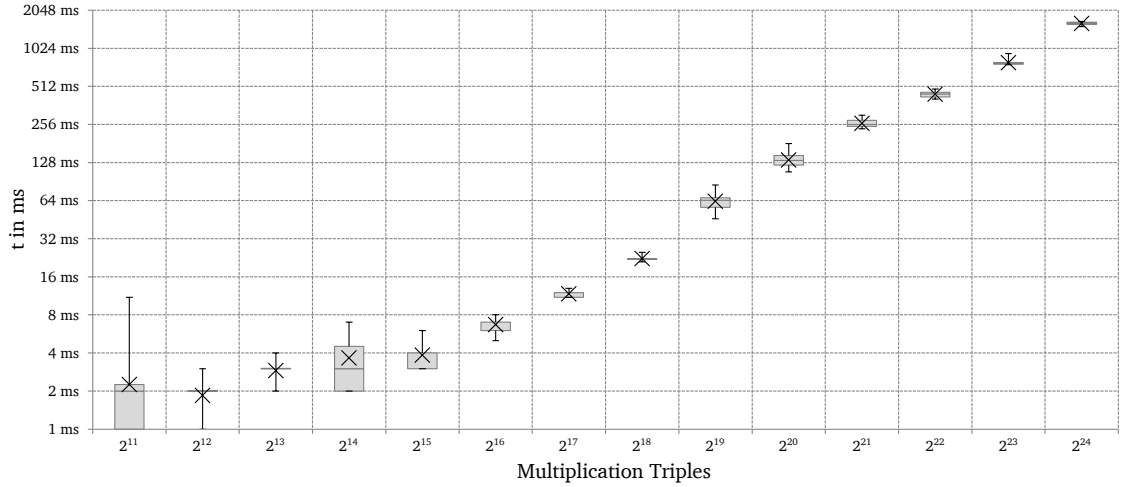


Figure 5.11: Android MT PRF, $N = 20$ Measurements each

Table 5.8: Android MT PRF, SGS3 denotes the Samsung Galaxy S III

MTs	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
SGS3 Mean ∅	1.9 ms	3.7 ms	6.7 ms	22.3 ms	134.1 ms	444.2 ms	1,607.8 ms
SGS3 St.dev. σ	0.5 ms	1.7 ms	0.7 ms	0.9 ms	16.5 ms	21.7 ms	43.7 ms
PC Mean ∅	0.5 ms	1.0 ms	1.0 ms	2.5 ms	10.0 ms	40.2 ms	169.5 ms
PC St.dev. σ	0.5 ms	0.0 ms	0.0 ms	0.5 ms	0.0 ms	0.7 ms	5.7 ms

Hashing and Reconstructing Contacts

In order to create the circuit inputs, the parties are querying contact data from their address book and the phone numbers are used as identifiers. This is a very cheap operation as shown in Table 5.9. The recover operation maps the found shared contacts' identifiers back to the actual contact. Both operations rely on Android's ContentResolver, which is querying a SQLite database of contact information.

Table 5.9: Hashing and Reconstructing Contacts

Contacts	32	64	128	256
Hash Mean ∅	40.20 ms	72.20 ms	85.80 ms	136.60 ms
Hash St.dev. σ	3.11 ms	16.84 ms	12.11 ms	33.87 ms
Recover Mean ∅	10.40 ms	17.00 ms	20.60 ms	33.40 ms
Recover St.dev. σ	0.55 ms	2.45 ms	6.23 ms	9.45 ms

Loading Circuits

We did not dynamically generate the Boolean circuits during runtime, but instead we load pre-generated circuits from memory. Their sizes and gate count is listed in Table 5.10. After the circuits have been read from memory their structure is created in the RAM. The amount of time required to load the circuits is shown in Figure 5.12 and Table 5.11. The total number of gates is the sum of XOR gates, AND gates, input gates and the Boolean constants 0 and 1. In the following, we use the abbreviation SCS to refer to sort-compare-shuffle circuits and BWA for the bit-wise AND circuits.

Evaluating AND Gates

Evaluating AND gates requires communication between the parties and is an expensive operation compared to evaluating XOR gates. Particular iterations took significantly longer than the average runtime, which can be seen from the rather high standard deviation values. Out of the 50 measurements we made for each circuit, approximately 2 values were significant

Table 5.10: Circuit Sizes

Circuit	BWA		24 Bit SCS				128 Bit SCS	
Inputs	672	16,384	32	64	128	256	32	64
Size in KiB	40	1,032	1,827	4,239	9,738	21,954	9,952	23,273
Total Gates	2,018	49,154	68,184	158,808	362,520	814,872	370,752	863,552
AND Gates	672	16,384	22,432	52,096	118,656	266,240	125,216	290,816
XOR Gates	0	0	44,085	103,317	236,949	534,549	237,213	556,029

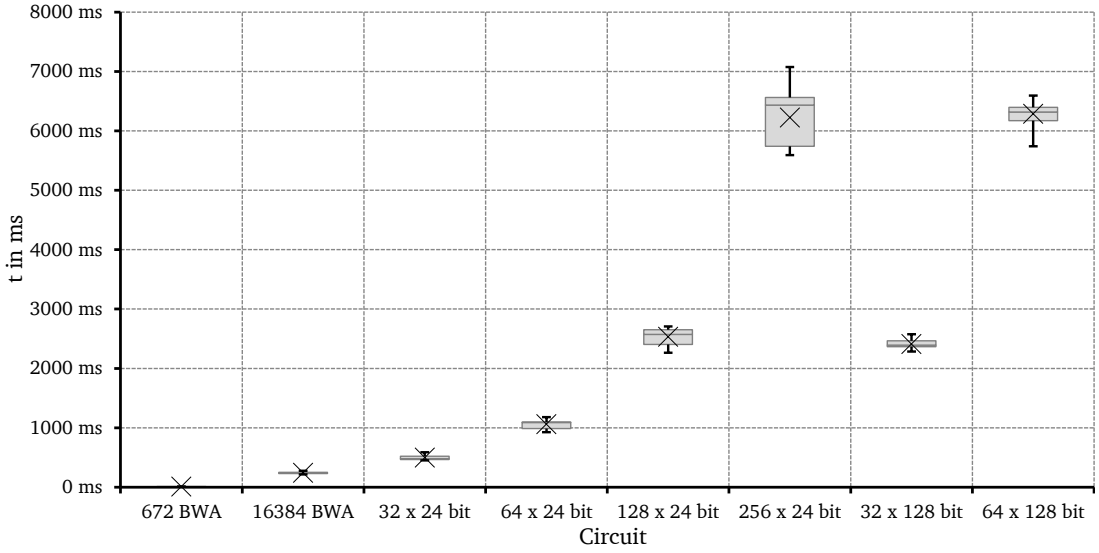


Figure 5.12: Loading Circuits, $N = 20$ Measurements each

Table 5.11: Loading Circuits, $N = 20$ Measurements each

Circuit	BWA		24 Bit SCS				128 Bit SCS	
Inputs	672	16,384	32	64	128	256	32	64
Mean ∅	9 ms	240 ms	495 ms	1,057 ms	2,528 ms	6,221 ms	2,409 ms	6,283 ms
Median	9 ms	239 ms	481 ms	1,091 ms	2,573 ms	6,434 ms	2,393 ms	6,317 ms
St.dev. σ	1 ms	16 ms	37 ms	70 ms	141 ms	449 ms	71 ms	189 ms

outliers. We suppose that this is due to the high amount of small data transactions via Wi-Fi Direct. The network latency is crucial to the overall performance, since the AND gates have to be evaluated iteratively in an interactive fashion. Large delays, like the ones we observed significantly increase the runtime.

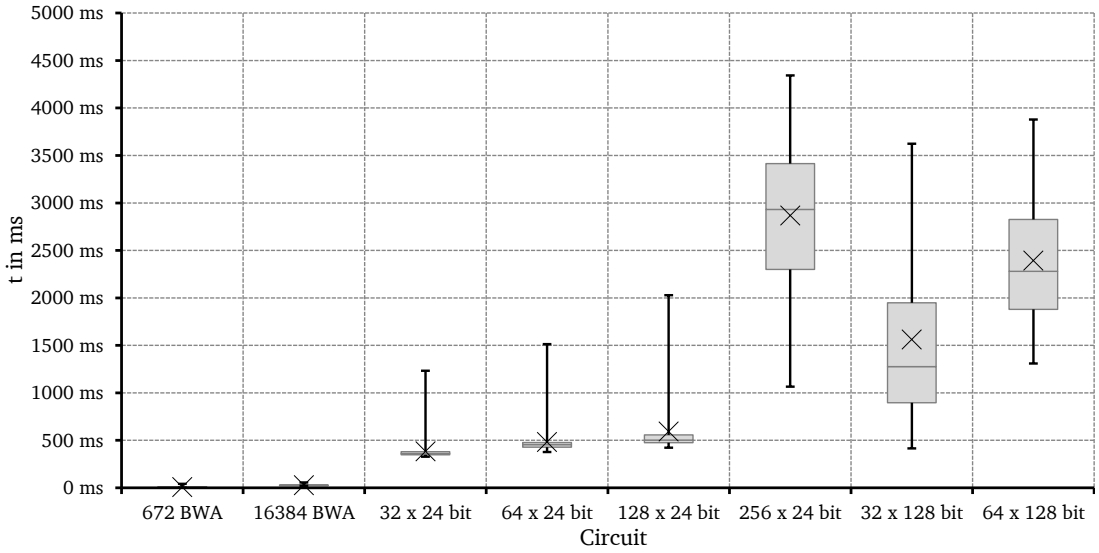


Figure 5.13: Evaluation of AND Gates, $N = 50$ Measurements each

Table 5.12: Evaluation of AND Gates, $N = 50$ Measurements each

Circuit	BWA		24 Bit SCS				128 Bit SCS	
Inputs	672	16,384	32	64	128	256	32	64
Mean ∅	8 ms	28 ms	383 ms	483 ms	592 ms	2,867 ms	1,561 ms	2,393 ms
Median	8 ms	25 ms	360 ms	455 ms	503 ms	2,932 ms	1,277 ms	2,279 ms
St.dev. σ	5 ms	8 ms	124 ms	162 ms	268 ms	797 ms	908 ms	627 ms

We further analyzed the AND gate evaluation for each circuit and depicted the values for the 64 input 24 bit circuit as one example in Table 5.13. The distribution of runtime is shown in Figure 5.14 and is comparable for all other circuits. The biggest part of the runtime is spent waiting for incoming shares. This is an instruction that is blocking until a value is received and shows a high variance due to the access of the network connection. Sending the shares however can be done rather efficiently. The calculation is only a small percentage of the total runtime while the masking of the inputs with the MTs and memory allocations need almost 17% of the time.

Table 5.13: Evaluation of the 64 x 24 Bit SCS Circuit AND Gates

	Mask	Calculate	Receive Shares	Send Shares
Mean ∅	81.56 ms	39.80 ms	316.86 ms	43.34 ms
Median	81.00 ms	40.00 ms	286.00 ms	42.50 ms
St.dev. σ	12.10 ms	5.62 ms	152.67 ms	6.38 ms
%	16.94%	8.26%	65.80%	9.00%

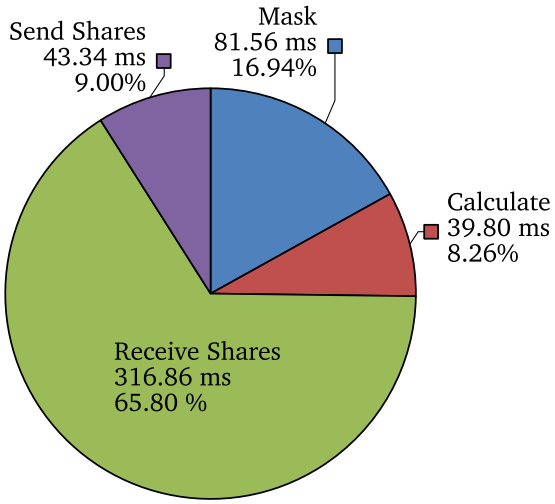


Figure 5.14: Evaluation of the 64 Input 24 bit Circuit AND Gates, $N = 50$ Measurements

Evaluating XOR Gates

The evaluation of XOR gates was considered as free since they can be calculated locally. However, we found that this still requires a non-negligible amount of runtime as depicted in Figure 5.15 and Table 5.14 for all circuits we evaluated that contain XOR gates.

When creating the two largest circuits (256×24 bit and 64×128 bit), we get close to the application's memory limit. We assume this is the explanation for the drastic runtime increase from the second largest circuit (128 inputs) to the largest circuit (256 inputs) with 24 bit hashes. For the three smallest 24 bit circuits the runtime growth for both AND and XOR gates is sub-linear.

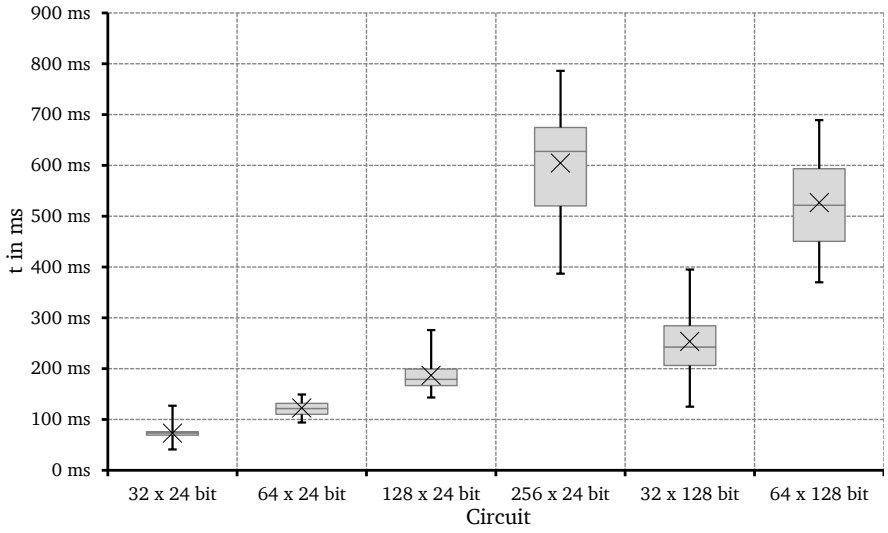


Figure 5.15: Evaluation of XOR Gates, $N = 50$ Measurements each

Table 5.14: Evaluation of XOR Gates, $N = 50$ Measurements each

Circuit	24 Bit SCS				128 Bit SCS	
Inputs	32	64	128	256	32	64
Mean ∅	72.8 ms	121.9 ms	186.5 ms	604.4 ms	253.0 ms	526.4 ms
Median	73.5 ms	121.5 ms	179.0 ms	627.5 ms	242.5 ms	521.5 ms
St.dev. σ	10.7 ms	14.5 ms	30.4 ms	97.8 ms	59.7 ms	86.9 ms

6 Conclusion

In this chapter we summarize the contents of this thesis, discuss our results and give an outlook to possible future work in the field of secure computation.

6.1 Summary and Discussion

While practically realized secure computation schemes on desktop computers are available for several years, they have only recently become possible on mobile devices. This is made possible by the enormous speedup of HW, the spread of smartphones and mainly because of the massive improvements of SFE schemes that have been researched in the past years.

While a lot of research was focused on Yao's GC protocol for a while, it was recently shown that the protocol of Goldreich, Micali and Wigderson can achieve better performance while allowing comparable security.

With our implementation we have shown that generic SFE can be realized on state-of-the-art mobile HW in reasonable time. The use of a trusted HW token allows us to solve the problem of pre-generating MTs in a secure way, thus improving the overall performance. To transport the generated MTs a secure channel is established with a key agreement protocol based on TLS that was implemented on a SC.

Unfortunately, our work is only based on a semi-honest adversary model which does not conform to real-life security requirements where active attacks are possible, applications can be manipulated and malicious data could be sent into the secure computation.

6.2 Future Work

There are several possibilities to continue the work done in this thesis. We want to give some ideas for future research that could contribute further to practically usable secure computation.

6.2.1 Communication

The communication between the participants could also be handled via NFC or Bluetooth. The amount of data transferred and the timing of our implementation should work with both communication protocols. However, the impact of the latency must be carefully evaluated since our used circuits are sensitive to high delays.

To enable new use cases, our scenario could be extended by a central server that connects the parties at different locations and forwards their data, without taking an active role in the computation itself. This would allow secure computation without being in the same location. However, securing the connection is a fundamental requirement.

The seed exchange protocol based on TLS should be modified to implementing a handshake based on the Diffie-Hellman key exchange in order to offer perfect forward secrecy. This requires the SC to support the desired algorithms in order to keep the execution times low.

6.2.2 Multi-party Secure Computation

We restricted our setting to the two-party case while MTs and the GMW protocol in general can also be applied to more than two parties. An extension to multi-party SFE would be interesting as it would allow for more complex use cases.

Settings with multiple SCs might also bring benefits to the overall performance and security since MTs can be generated in parallel and collusion scenarios with multiple SCs at the same time are less likely.

6.2.3 Extension to stronger Adversary Models

An extension to a stronger adversary model is another task that should be investigated as it would enable applications that are secure under realistic conditions. Especially on mobile platforms it will be challenging to realize due to the high computational requirements.

6.2.4 Implementation Improvements

There is a lot of room for innovating practical realizations in the field of secure computation, apart from the two use cases we realized and those that have been implemented in related work. Especially the use of recent SC hardware might offer entirely new functionality and massively increased performance.

We pre-generated the circuits used in our implementation. This could be changed in future work and circuits might be dynamically generated on the phone during runtime in the

online phase. This allows for custom sized circuits that directly adapt to the user's input sizes.

Additional research can be put into making apps usable for an average user and how to highlight security or privacy sensitive data. At the same time developers should be made aware of privacy concerns and be given the option to include secure computation functionality in their code in a convenient way.

List of Figures

List of Tables

List of Protocols

Abbreviations

AES	Advanced Encryption Standard
APDU	Application Protocol Data Unit
API	Application Programming Interface
CA	Certificate Authority
ECC	Elliptic Curve Cryptography
EEPROM	Electrically Erasable Programmable Read-Only Memory
GC	Garbled Circuit
GUI	Graphical User Interface
HW	Hardware
ISO	International Organization for Standardization
IV	Initialization Vector
JC	Java Card
MAC	Message Authentication Code
MSC	Mobile Security Card SE 1.0
MT	Multiplication Triple
NFC	Near Field Communication
OAEP	Optimal Asymmetric Encryption Padding
OT	Oblivious Transfer
PRNG	Pseudo-Random Number Generator
PRF	Pseudo-Random Function
RAM	Random Access Memory
RFC	Request For Comments
ROM	Read-Only Memory
SC	Smart Card

SFE Secure Function Evaluation

SHA Secure Hash Algorithm

SSL Secure Sockets Layer

SW Software

TLS Transport Layer Security

WPA Wi-Fi Protected Access

WPS Wi-Fi Protected Setup

Bibliography

[AP13] N. AlFardan, K. Paterson. “Lucky Thirteen: Breaking the TLS and DTLS Record Protocols”. In: (02/2013). URL: http://www.isg.rhul.ac.uk/tls/ (cit. on pp. 27, 31).

[BCGS09] P. Bichsel, J. Camenisch, T. Gross, V. Shoup. “Anonymous credentials on a standard java card”. In: *Computer and Communications Security (CCS'09)*. ACM, 2009, pp. 600–610 (cit. on p. 18).

[Bea91] D. Beaver. “Efficient Multiparty Protocols Using Circuit Randomization”. In: *Advances in Cryptology – CRYPTO'91*. Vol. 576. LNCS. Springer, 1991, pp. 420–432 (cit. on p. 10).

[Bea95] D. Beaver. “Precomputing Oblivious Transfer”. In: *Advances in Cryptology – CRYPTO'95*. Vol. 963. LNCS. Springer, 1995, pp. 97–109 (cit. on p. 9).

[BM10] C. Boyd, A. Mathuria. *Protocols for Authentication and Key Establishment*. 1st. Springer Publishing Company, Incorporated, 2010 (cit. on p. 27).

[CHK+12] S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, D. Rubenstein. “Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces”. In: *Cryptographers' Track at the RSA Conference (CT-RSA'12)*. Vol. 7178. LNCS. Springer, 2012, pp. 416–432 (cit. on pp. 11, 18).

[CMSA12] G. Costantino, F. Martinelli, P. Santi, D. Amoruso. “An implementation of secure two-party computation for smartphones with application to privacy-preserving interest-cast”. In: *Conference on Privacy, Security and Trust (PST'12)*. IEEE, 2012, pp. 9–16 (cit. on p. 18).

[DR08] T. Dierks, E. Rescorla. *RFC 5246: The Transport Layer Security (TLS) Protocol Version 1.2*. 08/2008. URL: http://tools.ietf.org/html/rfc5246 (cit. on pp. 5 sq., 27 sq.).

[DR11] T. Duong, J. Rizzo. “Here come the ⊕ Ninjas”. In: *Unpublished manuscript* (2011) (cit. on p. 27).

[DSV10] M. Dubovitskaya, A. Scafuro, I. Visconti. “On Efficient Non-Interactive Oblivious Transfer with Tamper-Proof Hardware”. In: *IACR Cryptology ePrint Archive* 2010 (2010), p. 509 (cit. on p. 18).

[ECR12] ECRYPT II. *Yearly Report on Algorithms and Keysizes (2011-2012)*. 2012. URL: http://www.ecrypt.eu.org/documents/D.SPA.20.pdf (cit. on pp. 28, 38).

[GMW87] O. GOLDREICH, S. MICALI, A. WIGDERSON. "How to play ANY mental game". In: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. STOC '87. New York, New York, USA: ACM, 1987, pp. 218–229. URL: http://doi.acm.org/10.1145/28395.28420 (cit. on pp. 2, 9 sq.).

[GT08] V. GUNUPUDI, S. R. TATE. "Generalized Non-Interactive Oblivious Transfer Using Count-Limited Objects with Applications to Secure Mobile Agents". In: *Financial Cryptography and Data Security (FC'08)*. Springer, 2008, pp. 98–112 (cit. on p. 18).

[HCE11] Y. HUANG, P. CHAPMAN, D. EVANS. "Privacy-preserving applications on smartphones". In: *Proceedings of the 6th USENIX conference on Hot topics in security*. HotSec'11. San Francisco, CA: USENIX Association, 2011. URL: http://dl.acm.org/citation.cfm?id=2028040.2028044 (cit. on p. 18).

[HEK12] Y. HUANG, D. EVANS, J. KATZ. "Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?" In: *Network and Distributed Security Symposium (NDSS'12)*. The Internet Society, 2012 (cit. on pp. 13, 17, 46).

[HEKM11] Y. HUANG, D. EVANS, J. KATZ, L. MALKA. "Faster secure two-party computation using garbled circuits". In: *Proceedings of the 20th USENIX conference on Security*. SEC'11. San Francisco, CA: USENIX Association, 2011. URL: http://dl.acm.org/citation.cfm?id=2028067.2028102 (cit. on pp. 10, 17).

[IKNP03] Y. ISHAI, J. KILIAN, K. NISSIM, E. PETRANK. "Extending Oblivious Transfers Efficiently". In: *Advances in Cryptology – CRYPTO'03*. Vol. 2729. LNCS. Springer, 2003, pp. 145–161 (cit. on p. 9).

[Int12] INTERNATIONAL DATA CORPORATION. *Android Marks Fourth Anniversary Since Launch with 75.0% Market Share in Third Quarter, According to IDC*. http://www.idc.com/getdoc.jsp?containerId=prUS23771812. [Online; accessed 2013-June-28]. 2012 (cit. on p. 8).

[JKSS10] K. JÄRVINEN, V. KOLESNIKOV, A.-R. SADEGHI, T. SCHNEIDER. "Embedded SFE: Offloading Server and Network Using Hardware Tokens". In: *Financial Cryptography and Data Security (FC10)*. Vol. 6052. LNCS. Springer, 2010, pp. 207–221 (cit. on pp. 2, 18).

[KBC97] H. KRAWCZYK, M. BELLARE, R. CANETTI. *RFC 2104: HMAC: Keyed-Hashing for Message Authentication*. 1997 (cit. on pp. 5, 30, 38).

[Kol10] V. KOLESNIKOV. "Truly Efficient String Oblivious Transfer Using Resettable Tamper-Proof Tokens". In: *Theory of Cryptography Conference (TCC'10)*. Springer, 2010, pp. 327–342 (cit. on p. 18).

[Kra01] H. KRAWCZYK. "The order of encryption and authentication for protecting communications (Or: How secure is SSL?)" In: *Advances in Cryptology—CRYPTO 2001*. Springer. 2001, pp. 310–331 (cit. on p. 30).

[KS08] V. KOLESNIKOV, T. SCHNEIDER. "Improved Garbled Circuit: Free XOR Gates and Applications". In: *Automata, Languages and Programming*. Vol. 5126. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 486–498 (cit. on p. 10).

[Maw12] N. MAWSTON. *Samsung Galaxy S3 Overtakes Apple iPhone 4S to Become World's Best-Selling Smartphone Model in Q3 2012*. http://blogs.strategyanalytics.com/HCST/post/2012/11/08/Samsung-Galaxy-S3-Overtakes-Apple-iPhone-4S-to-Become-Worlds-Best-Selling-Smartphone-Model-in-Q3-2012.aspx. [Online; accessed 2013-June-28]. 2012 (cit. on p. 34).

[MNPS04] D. MALKHI, N. NISAN, B. PINKAS, Y. SELLA. "Fairplay - a secure two-party computation system". In: *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX, 2004, pp. 287–302 (cit. on pp. 10, 17).

[MSW08] P. MORRISSEY, N. P. SMART, B. WARINSCHI. "A Modular Security Analysis of the TLS Handshake Protocol". In: *ASIACRYPT*. 2008, pp. 55–73 (cit. on p. 5).

[NP05] M. NAOR, B. PINKAS. "Computationally Secure Oblivious Transfer". In: *Journal of Cryptology* 18.1 (2005), pp. 1–35 (cit. on p. 9).

[Rab81] M. O. RABIN. "How to exchange secrets with oblivious transfer". In: (1981). Aiken Computation Lab, Harvard University (cit. on p. 9).

[SEE13] SEEK-FOR-ANDROID. *MscSmartcardService*. https://code.google.com/p/seek-for-android/wiki/MscSmartcardService. [Online; accessed 2013-July-03]. 2013 (cit. on p. 42).

[Sun06] SUN MICROSYSTEMS. *Java Card Specification 2.2.2 Final Release*. http://www.oracle.com/technetwork/java/javacard/specs-138637.html. [Online; accessed 2013-July-01]. 2006 (cit. on pp. 35, 37 sq.).

[SZ13] T. SCHNEIDER, M. ZOHNER. "GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits". In: *17th International Conference on Financial Cryptography and Data Security (FC'13)*. LNCS. Springer, 04/2013, pp. 275–292. URL: http://thomaschneider.de/papers/SZ13.pdf (cit. on pp. 11, 18 sq.).

[Wak68] A. WAKSMAN. "A Permutation Network". In: *J. ACM* 15.1 (1968), pp. 159–163 (cit. on p. 16).

[WS96] D. WAGNER, B. SCHNEIER. "Analysis of the SSL 3.0 protocol". In: *Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2*. WOEC'96. Oakland, California: USENIX Association, 1996, pp. 4–4. URL: http://dl.acm.org/citation.cfm?id=1267167.1267171 (cit. on p. 5).

[Yao86] A. C. YAO. "How to Generate and Exchange Secrets". In: *Foundations of Computer Science (FOCS'86)*. IEEE, 1986, pp. 162–167 (cit. on p. 9).