TECHNISCHE
UNIVERSITÄT
DARMSTADT

Master Thesis

PSI Meets Signal: Integrating a Malicious-Secure Private Contact Discovery Solution in an Open-Source Instant Messaging Service

Matthias Senker

July 23, 2018

CRISP
Center for Research
in Security and Privacy

Technische Universität Darmstadt
Center for Research in Security and Privacy
Engineering Cryptographic Protocols

Supervisors: M.Sc. Christian Weinert
Prof. Dr.-Ing. Thomas Schneider

Erklärung zur Abschlussarbeit gemäß §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Matthias Senker, die vorliegende Master Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Thesis Statement pursuant to §23 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I, Matthias Senker, have written the submitted Master Thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

I am aware, that in case of an attempt at deception based on plagiarism (§38 Abs. 2 APB), the thesis would be graded with 5,0 and counted as one failed examination attempt. The thesis may only be repeated once.

In the submitted thesis the written copies and the electronic version for archiving are identical in content.

Darmstadt, July 23, 2018

Matthias Senker

Abstract

Mobile messengers like WhatsApp have gained a lot of popularity in recent years. While their security is improving, e.g., with WhatsApp incorporating end-to-end encryption, many of them suffer from a lack of privacy. A prominent example for this is contact discovery. For users to find out, which of their contacts also use the messenger, many apps upload the user's entire address book to their server, including information about people that do not use the messenger.

Some messengers, like Signal, use private contact discovery, a use case of private set intersection (PSI). This allows users to find their friends while keeping any contacts, that do not use the messenger, private. Unfortunately, Signal uses a naive and insecure hashing-based PSI protocol that does not yield sufficient privacy protection. The results of a survey we conducted on 'secure' messengers, show that all of them provide little to no privacy during contact discovery.

We look at two precomputation based PSI protocols that have previously been identified as candidates for efficient and private contact discovery. In both protocols, we reduce the communication required to inform clients about changes to the server's database by up to four times. For one of the protocols, we nearly halve the online phase communication required for each of the user's contacts. We also make both protocols secure against malicious clients. To demonstrate the practicality of our work, we integrate our implementation into the Signal messenger for Android.

Contents

1 Introduction

In recent years, mobile messengers like WhatsApp have gained a lot of popularity, pushing back other forms of mobile text communication, like SMS or e-mail. In 2014, a study predicted that by 2018, 75% of all mobile messaging traffic would come from messenger apps [Row14]. Today, WhatsApp and Viber, two of the most popular messengers, each claim to have over one billion users.

For a more comfortable user experience, most messengers use some form of contact discovery. With this feature, users can immediately see which of their contacts also use the messenger and they can start communicating immediately. In most cases, contact discovery is done by uploading all contacts in a device's address book to the messenger's server, including contacts that do not use the service. This leads to a conflict between comfort and privacy.

Users might be worried about how the server handles this contact information and if it is shared with third parties. For example, in August 2016, WhatsApp changed its usage and privacy policy to include the right to share user data (including contact data) with its parent company Facebook, even if the user did not have a Facebook account. Users had to agree to the new policy if they wanted to continue using the service.

As a consequence, the data protection officer of Hamburg (Germany) forbade Facebook to collect and store the personal data of German WhatsApp users without consent that meets the requirements of German data protection regulations. Facebook's final complaint against this order was dismissed by the Hamburg higher administrative court in March 2018. [Ham18]

Also, in 2017, a German district court came to the conclusion that by using the WhatsApp messenger, users continuously transmit data about their contacts to the WhatsApp company. If this is done without a contact person's permission, that person can issue a chargeable warning to the WhatsApp user. [Dis17]

The solution for these issues is private contact discovery, which allows the user to learn, which of their contacts are also registered with the service, without revealing any information about their other contacts to the service provider.

1.1 Private Set Intersection

Private contact discovery is a use case of private set intersection (PSI). In PSI, two parties compute the intersection of their input sets without revealing to each other any inputs that are not part of that intersection. There are several practical uses for PSI protocols.

On example is measuring ad conversion rates. An advertiser, such as Google or Facebook, and a merchant may wish to find out how many of the users who saw an ad, also purchased the corresponding product, to evaluate the efficiency of the advertisement. With PSI protocols, this can be done without either party having to reveal their entire user base to the other. [PSSZ15]

Private contact discovery falls into a special category of PSI applications, where one side, typically a server, has a very large input set, while the input sets of the clients are rather small. Another application from this category is malware detection, which is, for example, used as motivation for the PSI protocol in [TLP+17]. In this application, an antivirus company holds a large malware database. Users can check if their applications are contained in this database. By using private set intersection, the users do not have to reveal any of their non-malware applications and the company can keep their malware database a business secret. Similar to private contact discovery, the size of the malware database is very large, while each users only has a small set of applications.

Two of the currently most efficient PSI protocols are presented in [PSZ18] and [KKRT16]. Despite their high efficiency, they are not well suited for the case of unequal set sizes, because their communication is always linear in the size of both input sets.

Kiss et al. have shown in [KLS+17], how unequal set sizes can be handled efficiently, by transforming four existing PSI protocols into a precomputation form. In their work, the expensive operations, that are linear in the size of the server set, are only performed once. Future protocol executions then only require communication and computation linear in the client's set size. The presented protocols are secure against semi-honest adversaries, which try to break privacy but follow the protocol specification. They are not secure against malicious attackers, who may deviate from the protocol specification at an point, e.g., by sending manipulated messages.

For private contact discovery, malicious security is especially important on the client side. Everyone can download the messenger app and potentially manipulate it. Malicious clients also have a higher incentive. If they can break the server's privacy and obtain its user database, they could try to sell it on the black market. The server, on the other hand, has little motivation to act malicious, as it would risk loosing users.

1.2 Contribution

The goal of this work is to show that PSI protocols for unequal set sizes can be practical. We motivate the need for such protocols through a survey conducted on secure messengers. The results show that currently used private contact discovery protocols offer only minimal privacy.

In this work, we focus on two of the protocols presented in [KLS+17], GC-PSI and NR-PSI. We suggest several new methods to significantly reduce their communication costs.

By using cuckoo filters [FAKM14] instead of bloom filters, we reduce the amount of data, that the server needs to transmit for every change to its database, by a factor of over 2 for GC-PSI and over 4 for NR-PSI. We also introduce cuckoo filter compression, which can reduce the size of sparse filters multiple times.

For NR-PSI we present a new combination of two oblivious transfer (OT) variants, namely OT precomputation and C-OT. This cuts the online phase communication for each of the client's contacts nearly in half.

We implement the improved protocols on PC and Android. To increase performance, we also add support for multithreading. Our implementations are secure against malicious clients.

In addition to evaluating our implementation, we further demonstrate its practicality by integrating it into Signal, a popular messenger with a strong focus on security and privacy. We chose Signal, because it is open source and its cryptography is well documented[1]. Also, it is often recommended as one of the most secure and yet easy to use messengers available [2,3,4].

This work is structured as follows: Chapter 2 introduces some necessary background information. Chapter 3 presents the results of our motivational survey. In Chapter 4, we present the PSI protocols together with all our improvements. This chapter also discusses the security of the protocols. In Chapter 5, we explain how we implemented the protocols and go into detail about some of the critical components. We also describe how we integrated our implementation into Signal. Our evaluation results are presented in Chapter 6. This work is concluded in Chapter 7.

[1] https://signal.org/docs/

[2] https://www.wired.com/story/ditch-all-those-other-messaging-apps-heres-why-you-should-use-signal/

[3] https://www.bestvpn.com/review/signal-private-messenger/

[4] https://mashable.com/2017/12/19/signal-app-privacy-download-use-it/

1.3 Related Work

In the following, we give an overview of PSI protocols that focus specifically on unequal set sizes. For an overview of further PSI protocols, we refer to the survey in [PSZ18] and the references given there.

As was demonstrated in [TLP+17], trusted execution environments (TEEs), such as Intel SGX[5] and ARM TrustZone[6], can be used for efficient private set intersection with unequal set size. The privacy critical part is executed inside a protected enclave where it is isolated from other processes and even the operating system. The enclave's memory is encrypted, making it unaccessible even for adversaries with hardware access. The TEE allows clients to verify that they are communicating with the correct software and that it is actually running inside a protected enclave. With this solutions, clients can simply upload their input set to the server without having to fear that the server might use it for anything but private set intersection. This work relies on the security of the trusted hardware. In 2017, an software-base side-channel attack was presented, that allows a malicious enclave to read another enclave's memory [SWG+17]. A demonstration for an attack on SGX using the recent Spectre bug [KGG+18] can be found at [Lar18].

A semi-honest secure PSI protocol for unequal set sizes was shown in [CLR17]. By using homomorphic encryption, it achieves communication costs linear in the client's set size but logarithmic in the size of the server set. As the presented evaluation results show, the performance of their protocol lies somewhere in the middle between the slow first run and the fast later runs of the protocols we focus on in this work.

Resende and Aranha were probably the first to use cuckoo filters [FAKM14] in their PSI protocol [RA18], which are also used in this work. Their protocol also uses a form of precomputation, with later runs being far more efficient than the first one. However, their protocol is only secure against semi-honest adversaries and they also do not apply compression to their cuckoo filters, like we do in this work (see Section 4.5).

A very efficient PSI protocol, that also targets private contact discovery, is shown in [DRRT18]. It uses private information retrieval. It is not secure against malicious adversaries. Also, for its strongest security, it requires two non colluding servers that share the same database.

[5] `https://software.intel.com/en-us/sgx`
[6] `https://www.arm.com/products/security-on-arm/trustzone`

2 Background

This chapter provides the necessary background information for the PSI protocols described in Chapter 4.

2.1 Oblivious Transfer

Oblivious Transfer (OT) is a cryptographic primitive introduced by Rabin in [Rab81]. In its most common variant, 1-out-of-2 OT [EGL85], a sender $\mathcal{S}$ provides two messages and a receiver $\mathcal{R}$ selects and receives one of them. $\mathcal{R}$ does not learn anything about the other message and $\mathcal{S}$ does not learn which message was selected. More formally, $\mathcal{S}$ provides messages (s_0, s_1) and $\mathcal{R}$ provides a choice bit b. In the end $\mathcal{R}$ will learn s_b but not s_{1-b} and $\mathcal{S}$ will not learn b. For the remainder of this work, OT refers to 1-out-of-2 OT. Also performing n OTs with messages of size m bits is denoted as OT_m^n.

It was shown in [IR89] that OT always requires some form of public key cryptography. The currently most efficient OT protocol [CO15] requires $3n + 2$ modular exponentiations for n 1-out-of-2 OTs and is secure against malicious adversaries.

2.1.1 OT Extension

Although OT cannot be realized without public key cryptography, [Bea96] proved that a small number (e.g., 128) of 'base OTs' can be extended to a very large number of OTs using only efficient symmetric cryptographic operations. The first OT extension protocol that is efficient in practice was presented in [IKNP03]. Several improvements and alternative flavors were described in [ALSZ13]. The more recent work of [ALSZ17] contains and further extends the results of [ALSZ13]. It also provides a thorough overview on the state-of-the-art of OT extension. An OT extension protocol for the more general 1-out-of-N OT variant was presented in [KK13]. For very short messages (e.g., one bit) it can provide more efficient 1-out-of-2 OTs than [IKNP03].

In the following the improved version of the OT extension protocol from [IKNP03] is described, beginning with some notations:

- n is the number of OTs
- m is the size of each message in bits

- κ is a security parameter (e.g., $\kappa = 128$)
- PRG is a seedable pseudo random number generator
- H is a hash function that outputs m-bit strings

The input of $\mathcal{S}$ consists of n pairs $(s_{0,j}, s_{1,j})$ of m-bit strings, $1 \leq j \leq n$.
The input of $\mathcal{R}$ is a vector of n choice bits $b = (b_1, ..., b_n)$.

Initially $\mathcal{S}$ chooses a random κ-bit vector Δ and $\mathcal{R}$ chooses κ pairs of κ-bit vectors $(k_0^i, k_1^i), 1 \leq i \leq \kappa$. Now the parties compute OT_k^k (e.g., using the protocol from [CO15]) with $\mathcal{R}$ as the sender and $\mathcal{S}$ as the receiver. $\mathcal{R}$ inputs (k_0^i, k_1^i) and $\mathcal{S}$ inputs Δ_i for $1 \leq i \leq \kappa$. As a result, $\mathcal{S}$ receives $k_{\Delta_i}^i$.

Next, both parties expand these κ-bit vectors into n-bit vectors by using them as keys to PRG. For $\mathcal{R}$ this means calculating:

$$t_0^i = \text{PRG}(k_0^i) \quad \text{and} \quad t_1^i = \text{PRG}(k_1^i), \qquad 1 \leq i \leq \kappa$$

$\mathcal{S}$ calculates $t_{\Delta_i}^i = \text{PRG}(k_{\Delta_i}^i)$, but is not able to calculate $t_{1-\Delta_i}^i$.

$\mathcal{R}$ then computes

$$u^i = t_0^i \oplus t_1^i \oplus b, \qquad 1 \leq i \leq \kappa$$

and sends these values to $\mathcal{S}$.

$\mathcal{S}$ calculates

$$q^i = \begin{cases} t_0^i, & if\, \Delta_i = 0 \\ u^i \oplus t_1^i, & if\, \Delta_i = 1 \end{cases}, \qquad 1 \leq i \leq \kappa$$

Notice that in both cases $q^i = t_0^i \oplus \Delta_i \cdot b$. Now consider a $n \times \kappa$ bit-matrix Q that is constructed by using these q^i as columns, i.e., $Q = [q^1|...|q^\kappa]$. Let q_j, $1 \leq j \leq n$ be the rows of Q. In the same way, let t_j be the rows of the matrix $T = [t_0^1|...|t_0^\kappa]$. Note that

$$q_j = t_j + b_j \cdot \Delta, \qquad 1 \leq j \leq n$$

At this point, depending on b_j, t_j is either equal to q_j or to $q_j \oplus \Delta$. Next, S calculates

$$v_{0,j} = \text{H}(j||q_j) \quad \text{and} \quad v_{1,j} = \text{H}(j||(q_j \oplus \Delta)), \qquad 1 \leq j \leq n$$

and R calculates

$$v_{b_j,j} = \text{H}(j||t_j), \qquad 1 \leq j \leq n$$

For the j-th OT, $\mathcal{S}$ now has two basically random values $(v_{0,j}, v_{1,j})$ out of which only one is known to $\mathcal{R}$ but $\mathcal{S}$ does not know which one. This corresponds to an OT with random messages.

In the final step, $\mathcal{S}$ uses these random values to mask the real messages by calculating

$$a_{0,j} = v_{0,j} \oplus s_{0,j} \quad \text{and} \quad a_{1,j} = v_{1,j} \oplus s_{1,j}, \qquad 1 \leq j \leq n$$

The masked values are then sent to $\mathcal{R}$, which unmasks the ones corresponding to its choice bits to receive its output:

$$s_{b_j,j} = a_{b_j,j} \oplus v_{b_j,j}, \qquad 1 \leq j \leq n$$

In addition to the computation and communication costs for the base OTs, this protocol requires only $3n$ evaluations of H, as well as the generation of $3\kappa n$ random bits with PRG. The number of transmitted bits is $\kappa n + 2mn$.

As described in [ALSZ13], the protocol can be run again to compute more OTs without computing new base OTs as long as PRG produces different random values for the new run. This can be achieved by including a counter into PRG's seeds that is incremented with each run. In a similar way, the entire protocol, after base OT computation, can be broken into smaller chunks which allows for an easy and efficient parallelization.

2.1.2 OT Extension Flavors

The following flavors for OT extension are described in [ALSZ17]:

Sender Random OT (SR-OT) In this flavor, the sender does not input any messages. Instead, for each OT, it obtains two random messages as output. The Receiver still obliviously receives one of these two messages. To achieve this, the protocol ends after the hash function evaluation. $\mathcal{S}$ then outputs $(v_{0,j}, v_{1,j})$ and $\mathcal{R}$ outputs $v_{b_j,j}$ for $1 \leq j \leq n$. Since no masked messages are transmitted, the communication cost is reduced to κn bits.

Receiver Random OT (RR-OT) If $\mathcal{R}$ wishes to select the messages randomly, a small optimization is possible. Instead of providing the n-bit choice vector b as input, it is calculated by the protocol as $b = t_0^1 \oplus t_1^1$. Note that only $\mathcal{R}$ can calculate this b because $\mathcal{S}$ does not know both t_0^1 and t_1^1. While this choice for b is obviously random, it also guarantees that u^1 is a zero vector. Thus, u^1 no longer needs to be transmitted, reducing the communication costs to $(\kappa - 1)n + 2mn$ bits.

Random OT (R-OT) SR-OT and RR-OT can be combined. In this case, neither party inputs any values. For each OT $\mathcal{S}$ outputs two random messages and $\mathcal{R}$ outputs a random choice bit together with the corresponding message. The communication cost for this flavor is $(\kappa - 1)n$ bits.

Correlated OT (C-OT) This flavor is useful when $\mathcal{S}$ wants to transmit random but correlated messages. For each OT (i.e., for $1 \leq j \leq n$), $\mathcal{S}$ picks the first message randomly as $s_{0,j} = v_{0,j}$ just like in SR-OT. However, the second message is calculated as a function of the first (i.e., $s_{1,j} = f_j(s_{0,j})$). f_j can be different for each OT. $s_{1,j}$ is then masked with $v_{1,j}$ and transmitted to $\mathcal{R}$. Depending on its choice bit, $\mathcal{R}$ either knows $v_{0,j}$ (and thus $s_{0,j}$) directly, or it knows $v_{1,j}$ and can obtain $s_{1,j}$ by unmasking the received message. Since only one masked message is transmitted for each OT, the communication cost for C-OT is $\kappa n + mn$ bits.

2.1.3 OT Precomputation

OTs can be precomputed, as was shown in [Bea95]. Any OT protocol can be split into an offline and an online phase, where all computationally expensive operations are performed in the offline phase by executing the protocol with random inputs. In the online phase, these random values are used to mask the actual inputs, requiring only inexpensive XOR operations.

For a detailed description, let (v_0, v_1) be the random inputs of $\mathcal{S}$ for one individual OT and let r be the random choice bit of $\mathcal{R}$. These values could be obtained via the R-OT flavor, described in Section 2.1.2. In the online phase, $\mathcal{R}$ uses r to mask its actual choice bit b. It calculates $b' = b \oplus r$ and sends b' to $\mathcal{S}$.

Now if $b' = 0$, then the random choice from the offline phase matches $\mathcal{R}$'s actual choice and $\mathcal{S}$ masks its actual messages (s_0, s_1) by calculating

$$s'_0 = s_0 \oplus v_0 \quad \text{and} \quad s'_1 = s_1 \oplus v_1$$

If $b' = 1$, then the random choice differs from the actual choice. In this case, $\mathcal{S}$ must swap the actual messages:

$$s'_0 = s_1 \oplus v_0 \quad \text{and} \quad s'_1 = s_0 \oplus v_1$$

$\mathcal{S}$ sends (s'_0, s'_1) to $\mathcal{R}$, which obtains its output by calculating

$$s_b = s'_r \oplus v_r$$

The only communication overhead introduced by precomputation is in the masked choice bits, i.e., 1 bit per OT.

OT precomputation is as secure as the OT protocol used in the offline phase. As long as a corrupt $\mathcal{R}$ only learns one of (v_0, v_1) per OT in the offline phase, it can only unmask one message in the online phase. And all that $\mathcal{S}$ learns in the online phase is if $r = b$ which does not reveal b.

2.1.4 Malicious-Secure OT Extension

The protocol described in Section 2.1.1 is only secure against semi-honest adversaries. A malicious receiver can obtain the sender's secret Δ, allowing them to unmask all of $\mathcal{S}$'s messages. Such an attack was already described in [IKNP03] and is based on $\mathcal{R}$ violating the protocol by not using the same choice vector b when calculating each of $(u^1, ..., u^\kappa)$. In the same work, Ishai et al. also provided a malicious-secure version of their protocol, however it is very expensive and requires multiple runs of the semi-honest protocol. The currently most efficient malicious-secure OT extension protocol was presented in [KOS15] and is described below. The protocol in [ALSZ15] is less efficient but requires weaker assumptions for its security proof. In [OOS17], malicious-secure 1-out-of-N OT extension was realized by adding checksums to the protocol from [KK13].

A modified version of [IKNP03] was introduced in [KOS15]. It adds only a small overhead in the form of a few checksums, to ensure $\mathcal{R}$ is not violating the protocol. To prevent a malicious $\mathcal{S}$ from using these checksums to learn anything about $\mathcal{R}$'s choice bits, a small and constant number of additional OTs is also added. More specifically, $\kappa + s$ OTs are added, where s is a security parameter. The authors of [KOS15] chose $s = 64$ in their own implementation. $\mathcal{R}$ uses random choice bits for the additional OTs. The checksums are calculated using finite field arithmetic (in $\mathbb{F}_{2^\kappa}$) in the following way:

Let $n' = n + \kappa + s$ be the new total number of OTs that need to be computed. At first, both $\mathcal{S}$ and $\mathcal{R}$ agree on n' random κ-bit vectors, $(x_1, ..., x_{n'})$, which will act as random weights. The checksums must be calculated and checked before $\mathcal{S}$ sends the masked messages $(a_{0,j}, a_{1,j})$ to $\mathcal{R}$ in the final phase of the protocol.

$\mathcal{R}$ calculates its checksums as

$$x = \sum_{j=1}^{n'} b_j \cdot x_j \quad \text{and} \quad t = \sum_{j=1}^{n'} t_j * x_j$$

where $*$ denotes multiplication in $\mathbb{F}_{2^\kappa}$. x and t are then transmitted to $\mathcal{S}$.

After obtaining Q, $\mathcal{S}$ calculates

$$q = \sum_{j=1}^{n'} q_j * x_j$$

Now $\mathcal{S}$ checks that $t = q \oplus x * \Delta$. If the check fails, $\mathcal{S}$ aborts. Otherwise the protocol continues normally.

In [KOS15] it was proven that if this check passes, $\mathcal{R}$ only knows one of the random masks $(v_{0,j}, v_{1,j})$ for each OT. It follows that $\mathcal{R}$ can only unmask one of the actual messages in the final step. Since the check is done before the masking, it can also be applied to the SR-OT and CR-OT flavors (see Section 2.1.2) which differ from the default protocol only in that final masking step. RR-OT only differs in how b is chosen, which also does not interfere with calculating the checksums. Thus, malicious security can be provided for all OT flavors

listed in Section 2.1.2. Finally, malicious-secure OT precomputation (cf. Section 2.1.3) can be achieved by executing malicious-secure R-OT in the offline phase.

2.2 Yao's Garbled Circuits

Yao's garbled circuits [Yao86] is a cryptographic protocol that allows two semi-honest parties to evaluate a function over their private inputs without revealing them to each other. One party, called the garbler, encrypts a boolean circuit, representing the function, and sends it to the other party, the evaluator, who evaluates the circuit without actually knowing the bits on each wire. In Section 2.2.1, the basic protocol is described, followed by its state-of-the-art improvements in Section 2.2.2.

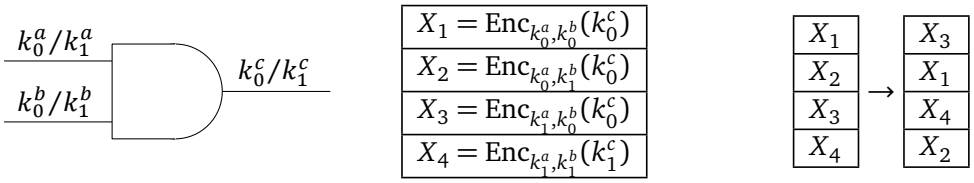
An alternative protocol for secure evaluation of boolean circuits is the Goldreich-Micali-Wigderson protocol (GMW). It was introduced in [GMW87] and is based on secret-sharing the bits on each wire. Improvements were given in [CHK+12], [SZ13] and [ALSZ13]. The GMW protocol is not used in this work because it requires multiple communication rounds during circuit evaluation. This would break the reasoning in Section 4.7.2 about malicious security. It also makes the protocol inefficient in networks with high latency, such as the mobile networks considered in this work. In low-latency networks, GMW can actually outperform Yao's garbled circuits, as shown in [SZ13].

2.2.1 Basic Protocol

Initially both parties construct the same boolean circuit, using only gates that take two input bits and produce one output bit. Next, the garbler assigns two random keys (k_0^w, k_1^w) to each wire w in the circuit. This includes the circuit's input and output wires as well as all intermediate wires. The keys represent 0 and 1 on their wire and are normally 128 bits long. From here on, when discussing individual gates, let a and b be the input wires to a single gate and let c be its output wire. The keys on these wire will be called input and output keys.

Typically the circuit consists only of XOR-gates and AND-gates. Every other two-input gate can be constructed using at most one XOR-gate or one AND-gate and some inversions (e.g., a OR b = NOT ((NOT a) AND (NOT b))). An inversion can be implemented 'for free' by swapping the keys on a wire.

In the next step, the garbler encrypts the circuit, one gate at a time. There are four combinations of input keys per gate, each corresponding to one output key. For each input key pair (k^a, k^b), the garbler encrypts the output key k^c with the corresponding input keys as $\mathrm{Enc}_{k^a,k^b}(k^c)$, where Enc is some symmetric encryption function that uses two keys. After a gate is encrypted, its ciphertexts are put into a random order and sent to the evaluator. Key



(a) assign keys to wires **(b)** encrypt output keys **(c)** reorder ciphertexts

Figure 2.1: Illustration of the steps performed by the garbler in Yao's garbled circuits protocol for a single AND gate.

assignment, encryption and reordering of ciphertexts for a single gate are illustrated in figure 2.1.

Before evaluation can begin, the evaluator must obtain one key for each of the circuit's input wires. The garbler sends the keys corresponding to their private input bits to the evaluator. Because the keys are random, the evaluator does not learn the garbler's input. For the evaluator to obtain the keys for their own input bits, both parties perform an oblivious transfer (see Section 2.1). For each input wire, the garbler sends both keys and the evaluator obliviously receives one of them, depending on its input bit. In this basic protocol, it would also be possible to let the OT protocol generate the keys in the first place, by using the SR-OT flavor described in Section 2.1.2. However, this would need to be done before the garbler encrypts the circuit, because the keys are required for encryption.

Next, the evaluator processes each gate, starting with the gates that directly consume two of the circuit's input keys. Using the gate's input keys, the evaluator tries to decrypt each of the four ciphertexts. Only one of these decryptions will be successful and return the gate's output key. Because the order of the ciphertexts is random, the evaluator has no way of knowing, which bit the output key represents.

After processing all gates, the evaluator knows one key for each output wire of the circuit but not the actual output bits. The garbler on the other hand, knows both keys, but not which one belongs to the actual output. Depending on which party shall learn the output, the garbler may send its key-to-bit mapping to the evaluator or the evaluator may send its output keys to the garbler. Whoever reconstructs the output can then send it to the other party, if they are both supposed to learn it.

2.2.2 Improvements

This section briefly describes the current state-of-the-art improvements for Yao's garbled circuits.

Point-and-Permute In [BMR90] a technique named point-and-permute was described that enables the evaluator to process a gate with only one decryption (instead of up to four). When assigning keys to a wire w, the garbler picks a random selection bit s and sets the least significant bit of the keys to $k_0^w[0] = s$ and $k_1^w[0] = 1 - s$.

Instead of garbling the ciphertexts for a gate, they are now sorted by the selection bits of the input keys, which still appears as a random order to the evaluator. When processing a gate, the selection bits of the input keys are used to identify the ciphertext that needs to be decrypted.

With this technique, the encryption scheme no longer needs to tell if decryption was successful, which may allow for smaller ciphertexts. A simple scheme can be implemented with a hash function H as $\mathrm{Enc}_{k^a,k^b}(k^c) = \mathrm{H}(k^a||k^b) \oplus k^c$.

Another advantage of point-and-permute is in the output reconstruction step, where it is sufficient to transmit the selection bits, instead of entire keys.

Free-XOR A way to fully eliminate the communication costs for XOR-gates was shown in [KS08]. For each wire w, the keys are chosen so that $k_1^w = k_0^w \oplus R$ where R is a global random bit-string only known to the garbler. For XOR-gates, the output keys are calculated from the input keys as $k_0^c = k_0^a \oplus k_0^b$ (and again $k_1^c = k_0^c \oplus R$). Note that if $R[0] = 1$ this approach is compatible with point-and-permute. To obtain the output key for an XOR-gate, the evaluator no longer requires any ciphertexts. Instead they just calculate $k^c = k^a \oplus k^b$.

Since now only AND-gates require the transmission of ciphertexts, circuits should be designed to have the lowest number of AND-gates possible. In [SHS+15] it was demonstrated that logic synthesis techniques can be used to achieve this.

Because the keys for a wire are now correlated, when transmitting the keys for the evaluator's input via oblivious transfer, it is no longer possible to use the SR-OT flavor as described in the previous section. Instead, the C-OT flavor (see Section 2.1.2) can be used and this is in fact the main use-case for which [ALSZ13] introduced C-OT.

Garbled-Row Reduction In [NPS99] is was shown how to reduce the number of ciphertexts per gate from four to three. The first ciphertext is not computed but instead defined as a string of 0s, which means it does not need to be transmitted. The corresponding output key is no longer randomly chosen, but instead computed by decrypting this 0-string.

Half-Gates A way for reducing the ciphertexts per AND-gate to two, that is compatible with free-XOR, was presented in [ZRE15]. The full approach is too complex to be described here, but the idea is to split AND-gates into two half-AND-gates, which are AND-gates for which one party always knows one of the inputs. Each of these half-AND-gates requires the

transmission of one ciphertext and is processed with one decryption. Internally, this scheme makes use of all the improvements listed so far.

Fixed-Key Blockcipher In [BHKR13] the authors showed a highly efficient construction for the encryption function Enc. Let T be a unique identifier for each gate and let π be a fixed-key block cipher (with the key known to both parties). Enc is defined in [BHKR13] as

$$\mathrm{Enc}_{k^a,k^b}(k^c, T) = \pi(K) \oplus K \oplus k^c, \qquad \text{where } K = 2k^a \oplus 4k^b \oplus T.$$

The evaluator can decrypt a ciphertext by XOR-ing it with $\pi(K) \oplus K$. When π is instantiated as AES encryption (without the key-expansion step), very fast implementations are possible. On Intel platforms, AES-NI [Gue10] can be used for highly efficient hardware-based AES encryption.

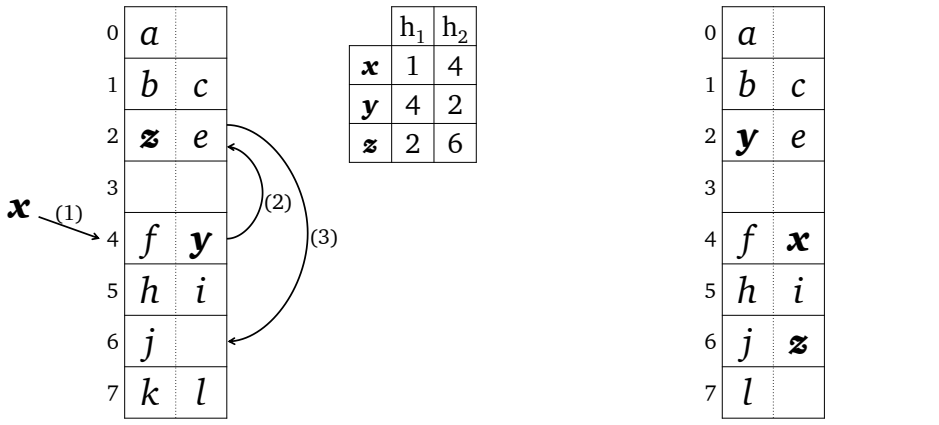
Pipelining The authors of [HEKM11] provided an efficient framework for garbled circuits that uses pipelining to greatly improve performance and reduce memory consumption. In this framework the evaluator first learns all the input keys. Then the garbler encrypts and transmits the gates in an order that allows the evaluator to process each gate as soon as they receive it. At no point does any party need to hold the entire encrypted circuit in memory. In fact the memory requirements are very low and nearly constant. This improves performance by using caches more efficiently. It also allows the framework to handle circuits of almost arbitrary size.

2.3 Cuckoo Filters

Cuckoo filters [FAKM14] are data structures used to represent (large) data sets. They are very space efficient and offer fast membership tests with a low false positive rate. Due to their small size they are especially useful when transferring large sets over a network.

Cuckoo filters are based on cuckoo hash tables ([PR04], [DW07]) which differ from traditional hash tables by using two (sometimes more) hash functions. These tables consist of n fixed size buckets, each with space for b key/value pairs. When a new pair is inserted, two candidate buckets are calculated from its key, using the two hash functions (from here on h_1 and h_2). If one of these buckets has free space, the pair is inserted there. If both buckets are full, the pair randomly replaces another pair in one of these buckets. The replaced pair is then reinserted into its alternate candidate bucket. This process of replacing and reinserting an entry is called relocation. If the alternate candidate bucket for the relocated pair is also full, further relocations are performed until either a non-full bucket is found or a maximum number of iterations were performed, at which point the table is considered to be full. Due to these relocations, Cuckoo hash tables can reach high space utilization (e.g., 95%) with very high probability.

Figure 2.2 shows an example insertion process with two relocations. For simplicity, only the keys are shown. Since in each step, the relocated key is chosen randomly, other outcomes would also be possible.



(a) step 1: x is inserted, replacing y
step 2: y is relocated, replacing z
step 3: z is relocated

(b) the state after the insertion of x and all subsequent relocations

Figure 2.2: Inserting a key into a cuckoo hash table with bucket size $b = 2$ (similar to figure 1 in [FAKM14]).

No matter how many relocations are performed, each inserted key will always be in one of its candidate buckets. This makes lookup and deletion fast and straight forward: Given a key, the two candidate buckets are determined and then searched for that key. For a lookup, the corresponding value is returned and for a deletion the pair is removed from its bucket.

A cuckoo filter differs from a cuckoo hash table in that it does not contain key/value pairs. Instead, for each inserted item x, only a fingerprint f_x is stored, which is a small bit-string obtained by hashing x. To enable relocation for these fingerprints, it must be possible to determine the alternate candidate bucket for a fingerprint without knowing the actual item. Because of this, cuckoo filters define h_1 and h_2 as

$$\mathrm{h}_1 = \mathrm{h}(x)$$
$$\mathrm{h}_2 = \mathrm{h}_1(x) \oplus \mathrm{h}(f_x)$$

where h is a hash function different from the one used to obtain f_x. (In practice one might hash x only once and then split the resulting bit-string to obtain $\mathrm{h}(x)$ and f_x) Note that with this definition, n must be a power of two or else, the XOR-operation might produce an out-of-range bucket index.

Now for a fingerprint f_x located in a bucket i, the other bucket j can always be calculated as

$$j = i \oplus \mathrm{h}(f_x)$$

Due to hash collisions, two items can have equal fingerprints. Because of this, lookups can produce false positives. The false positive rate ϵ is mainly dependent on fingerprint size f and also slightly on bucket size b, as larger buckets result in more possible collisions within each bucket. Given b and ϵ, the minimum fingerprint size is calculated as

$$f \geq \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil \quad \text{bits}$$

The space efficiency of a cuckoo filter depends not only on f but also on the load factor α, which is the ratio of inserted items to available item slots $(n \cdot b)$. The average space cost per item is calculated as $C = f/\alpha$. The maximum achievable value for α grows with b. This means that larger buckets increase the maximum load factor but also increase the fingerprint size. It was shown in [FAKM14] that for $\epsilon \leq 2 * 10^{-3}$ optimal space efficiency is achieved by setting the bucket size to $b = 4$. In this case, α can reach 95% with very high probability and the optimum space cost per item is

$$C \; = \; \lceil \log_2(1/\epsilon) + 3 \rceil / 0.95 \; \approx \; 1.05 \lceil \log_2(1/\epsilon) + 3 \rceil$$

However, this value is not always achievable in practice. Since n is limited to powers of two, the smallest possible filter for a given set of items might provide a load factor much smaller than 95%.

Bloom Filters Introduced in [Blo70], bloom filters are an older alternative to cuckoo filters. They are used in many current networking systems and protocols such as the PSI protocols in [KLS+17]. Just like cuckoo filters, they offer high space efficiency and fast membership tests with a small false positive rate. Bloom filters require $1.44 \log_2(1/\epsilon)$ bits per item which makes them larger than optimally filled cuckoo filters for most false positive rates (e.g., $\epsilon \leq 10^{-3}$). However, the size of a bloom filter is not restricted as it is for cuckoo filters, so for a specific number of items, a size-optimal bloom filter might actually be smaller than the smallest possible cuckoo filter.

The main difference between the two is that bloom filters do not support dynamic item deletion. To delete an item from a bloom filter (without introducing false negatives in future lookups) the entire filter has to be rebuilt. Variants with support for deletion exist (e.g., counting bloom filters [FCAB00]), but they require significantly more space than standard bloom filters.

3 Motivational Survey

This chapter presents the results of a survey performed on secure Android instant messengers. The goal of this survey is to determine how contact discovery is performed in practice. The surveyed messengers are secure in the sense that they offer end-to-end encryption. Evaluating their actual security is not part of this work. Section 3.1 describes how the messengers were surveyed. Section 3.2 describes the contact discovery methods found. Detailed results for each messenger are given in Section 3.3.

3.1 Performing the Survey

This section explains the different methods that were used to analyze the messengers.

3.1.1 Evaluating Privacy Policies

Many messengers describe their contact discovery scheme in their privacy policy. All surveyed messengers come with such a policy, which is also linked on their Google Play Store[1] page. Ideally, a privacy policy states which data an app transmits to its server and how the server processes and stores that data. This includes contact data from the smartphone's address book. As of 25th May 2018, the General Data Protection Regulation [EC16] of the European Union makes the provision of a privacy policy mandatory for all businesses that deal with data of EU citizens.

While all messengers provide a privacy policy, they are not always precise enough to determine the contact discovery method used. Furthermore, some policies only address privacy concerns for the developer's website without mentioning the actual app at all. Such policies are of no use to this survey.

3.1.2 Inspecting App Communication

To determine the contact discovery method for messengers without a proper privacy policy, the communication between app and server was inspected via special software tools acting as a man-in-the-middle. For most messengers, the communication was analyzed using the

[1] `https://play.google.com/store`

android app Packet Capture[2]. It captures messages via a local VPN and requires no root access on the device. For G DATA Secure Chat, Packet Capture would not recognize and decrypt the TLS traffic, because that messenger communicates with its server over port 8080, which is not typical for TLS. For this messenger, mitmproxy[3] was used to analyze the traffic.

Disabling Certificate Pinning

To analyze TLS encrypted communication (as it was used by all analyzed messengers), man-in-the-middle tools come with their own certificate authority (CA). This CA (more precisely, its certificate) has to be installed on the device and must be trusted by the app. However, most of the analyzed messengers employ certificate pinning to prevent man-in-the-middle attacks. With this technique, an app expects a very specific certificate from its server and ignores any installed CAs. Moreover, in recent versions of android, apps no longer trust user-added CAs unless explicitly allowed by the developer.

These issues were circumvented by using the Xposed[4] framework together with the JustTrustMe[5] plugin. Xposed allows its plugins to modify the behavior of an app during its execution, without modifying the app's APK file. JustTrustMe is aware of the certificate check routines in several commonly used security libraries. It can disable these checks in any app that utilizes one of these libraries. This solution requires root access on the device.

3.2 Found Contact Discovery Methods

In total, 14 messengers were surveyed for this work. Only seven of them support actual private contact discovery. The other half consists of five messengers, that just upload contact data directly to their servers and two that don't provide any contact discovery at all.

3.2.1 Uploading Hashed Contact Data

All of the seven messengers with private contact discovery use a hash based scheme. The app generates a hash value for each contact in the address book. Usually only the phone number or e-mail address is hashed. These hashes are then sent to the server, where they are compared to hashes for all registered users. The server then informs the app about any found matches. For the non-matching contacts, the server only learns their hash values, which (in

[2] https://play.google.com/store/apps/details?id=app.greyshirts.sslcapture
[3] https://mitmproxy.org/
[4] http://repo.xposed.info/
[5] https://github.com/Fuzion24/JustTrustMe

theory) cannot be turned back into the actual contact data, thus preserving the privacy of these contacts. However, even if the server cannot determine the actual contact data, it can still tell if two users share a contact, because they would both upload the same hash value. This can be prevented, by using random salt values, generated by the app and sent to the server together with the hashes. In this case the server has to hash the contact information of all its users, for every contact discovery execution. Only one messenger was found in this survey that uses salt values.

While hash based contact discovery is fast, easy to implement and requires very low bandwidth, it does not offer a high level of privacy. The set of possible inputs to the hash function is usually relatively small, especially in the case of phone numbers. A server can hash all possible phone numbers, allowing it to invert all received hashes. This is a well known problem. The developers of Signal discussed it in their blog [Ope14b]. They also mentioned alternatives with better privacy, but argued that those solutions are not practical due to high bandwidth requirements. Reducing those requirements is one of the goals of this work.

3.2.2 Contact Discovery with Intel SGX

Recently, the Signal developers have presented a new contact discovery implementation using Intel SGX, similar to the work in [TLP+17]. It is currently only a proof of concept and not actually used by the Signal messenger. But this may change in the near future, which is why it is mentioned here. In this implementation, the app still transmits hashes to the server, but the server runs the contact discovery inside an SGX enclave, allowing clients to verify, that the correct code is executed and that the hashes are only used for contact discovery. Different from [TLP+17], the Signal developers assume that even though a malicious server cannot access the actual data inside the enclaves' protected memory, it can still observe which memory sections are read or written at any time. A lot of effort was put into making any memory access completely oblivious, thus preventing attacks even from physical adversaries eavesdropping on the servers memory bus. This solution, while tricky to implement, is very efficient, since it requires no more bandwidth than the traditional hash-based contact discovery. Its privacy, however, depends on the security of Intel SGX. Examples for attacks on SGX are given in Section 1.3.

3.3 Details on Surveyed Messengers

This section provides a brief description for each surveyed messenger. This includes their contact discovery scheme and how it was determined. This information is also summarized in table 3.1.

The following three properties apply to all messengers, unless explicitly states otherwise:

- App and server are proprietary, i.e., not open source.
- The messenger does not support anonymous users - instead each account is bound to a verified phone number or e-mail address.
- Contact discovery is mandatory, i.e., the service cannot be used without it.

Confide The goal of Confide[6] is to prevent users from disclosing received messages to others. For this purpose, messages are deleted immediately after reading them and techniques are used that try to prevent taking screenshots of messages. Contact discovery is optional and according to their privacy policy, they "store the Address Book Data in hashed/anonymized form" [Con17].

Cyphr Cyhpr[7] is developed by the swiss company Golden Frog and its servers are all located in switzerland. It does not provide any contact discovery at all. When creating an account, users must provide a valid e-mail address. To establish communication, the address of the communication partner must be known and entered into the app. It is possible to grant Cyphr access to the address book but this only allows users to select e-mail addresses more conveniently instead of typing them manually. Analyzing Cyphr's traffic showed no indication of transmission of address book data at any point. Unfortunately, Golden Frog's privacy policy does not mention the Cyphr app at all.

Dust Dust[8] deletes all messages 24 hours after they are sent or (optionally) immediately after they are read. Dust's privacy policy describes the contact discovery mechanism only vaguely, telling the user that they "may also have access to your contact list(s)" [Rad16]. Traffic analysis showed that contact names (base64 encoded) and phone numbers are directly uploaded to Dust's servers. However, contact discovery is optional. Users can also be found by searching for their user name.

Eleet The Eleet Private Messenger[9] comes with an integrated bitcoin wallet, allowing users to exchanges bitcoins inside of a chat. New Eleet accounts are created anonymously, i.e., without providing a valid e-mail address or phone number. Contacts can be added via their user id or by scanning a QR-Code generated by that contact. Users can link their user ids with a phone number. They can then be found via the optional contact discovery, on which Eleet's privacy policy states: "A copy of the phone numbers and names in your address book (...) will be stored on our servers" [ELE].

[6] https://getconfide.com/
[7] https://www.goldenfrog.com/de/cyphr
[8] https://usedust.com/
[9] https://eleet.im/

G DATA Secure Chat The open source messenger G DATA Secure Chat[10] does not come with its own privacy policy. However, the repository [G D17] shows that this messenger is a fork from the Signal messenger. Analyzing the traffic confirms that it also uses Signal's hash based contact discovery scheme.

Hoccer The German messenger Hoccer[11], with all its servers located in Germany, puts a great emphasis on anonymity. Users do not provide Hoccer with their phone number or any other personal information. Contacts are added manually by sending them an SMS or e-mail or by one user generating a QR-code which is scanned by the other user. A traditional contact discovery scheme is not used and Hoccer's privacy policy clearly states: "We also do not know any phone numbers or E-mail addresses of you or your communications partners." [Hoc] However, users can choose to upload their geographic location to Hoccer's server to find other Hoccer users nearby.

Signal The open source messenger Signal[12] by Open Whisper Systems is the successor to their previous products Redphone and TextSecure. As mentioned in 1.2, Signal is often recommended as one of the most secure mobile messengers available. The privacy policy states that "Information from the contacts on your device may be cryptographically hashed and transmitted to the server in order to determine which of your contacts are registered" [Ope]. According to Signal's API protocol documentation, only the phone numbers are hashed using SHA-1 [Ope14a].

SIMSme SIMSme[13] is the private messenger developed by the Deutsche Post AG. All its servers are located inside Germany. Its privacy policy states: "An encrypted list of the telephone numbers of your contacts are uploaded to our servers in order to compare them and establish contact between users." [Deu14]. The term "encrypted" in this statement could refer to hash values but is not perfectly clear. Analyzing the transmitted data shows, that for each phone number, a short, random-looking string is transmitted. These strings always have the same size, even for very large phone numbers. This strongly indicates that a hash based contact discovery scheme is used. The strings differ when executing the protocol multiple times on the same contacts. Also, the app uploads a seemingly random value, labeled as salt, to the server. So we conclude that SIMSme uses randomly salted hash functions for its contact discovery.

Telegram The Telegram[14] client (but not the server) is open source. The security of Telegram is often discussed, with its end-to-end encryption only available in so called 'secret chats' which

[10] `https://www.gdata.de/securechat`
[11] `https://hoccer.com/`
[12] `https://signal.org/`
[13] `https://www.sims.me/en`
[14] `https://telegram.org/`

offer only a reduced feature set. The privacy policy states that names and phone numbers from the users address book are stored on Telegram's servers [Tel].

Threema Of all surveyed messengers, Threema[15] is the only one that cannot be obtained for free. The Threema developer and all its servers are located in Switzerland. Contacts can be added manually via their user id. Users may choose to bind their accounts to a phone-number or e-mail address, in which case they can be found by Threema's optional contact discovery scheme. The privacy policy states that data from the address book "is transmitted to the servers in one-way encrypted ("hashed") form" [Thr18b]. The additional cryptography whitepaper states that SHA-256 is used to hash phone numbers and e-mail addresses independently. [Thr18a] Users can also verify each other via QR-codes.

Viber Viber[16] is a popular competitor to WhatsApp, with currently over one billion users, according to their own website. It provides end-to-end encryption since version 6.0. No privacy is offered during contact discovery: "A copy of the phone numbers and names of all your contacts (...) will be collected and stored on our servers" [Vib17].

WhatsApp WhatsApp[17] is probably the most popular mobile messenger, with over one billion users, as stated on their website. Although originally not a secure messenger, current versions of Whatsapp provide end-to-end encryption by default, based on the Signal Protocol designed by Open Whisper Systems [Wha17]. According to WhatsApp's privacy policy, users "provide (WhatsApp) the phone numbers in (their) mobile address book on a regular basis" [Wha18].

Wickr Me The Wickr Me[18] private messenger is one of several Wickr products that all use the same open source messaging protocol. Users of Wickr Me can choose to bind their phone number to their account. If they do, they can be found via the optional contact discovery scheme, for which "the Wickr Me App will send a disguised representation of (the user's) contacts phone number to (Wickr's) servers" [Wic17]. The rest of their privacy policy makes it clear, that 'disguised' means 'hashed' in this statement.

Wire Wire[19] is an open source messenger with servers located in Germany and Ireland. A special feature is the support of multiple accounts (e.g., business and private) which can be switched easily from within the app. Contact discovery is optional and hash based. In addition to a privacy policy, they provide a detailed security white paper, which states: "each address

[15] https://threema.ch/en
[16] https://www.viber.com/
[17] https://www.whatsapp.com/
[18] https://www.wickr.com/personal
[19] https://wire.com/en/

book entry is hashed (using SHA-256) before being transmitted to the server" [Wir17].

Messenger	Contact Discovery Method	What was analyzed?
Confide	hashed contacts	privacy policy
Cyphr	none	network communication
Dust	address book upload	privacy policy and network communication
Eleet	address book upload	privacy policy
G DATA Secure Chat	hashed contacts	network communication
Hoccer	(can upload location to find nearby users)	privacy policy
Signal	hashed contacts	privacy policy and protocol API
SIMSme	hashed contacts (with salt)	privacy policy and network communication
Telegram	address book upload	privacy policy
Threema	hashed contacts	privacy policy
Viber	address book upload	privacy policy
WhatsApp	address book upload	privacy policy
Wickr Me	hashed contacts	privacy policy
Wire	hashed contacts	privacy policy

Table 3.1: Survey results summary

4 Optimizing PSI Protocols for Unequal Set Sizes

This chapter presents the two PSI protocols, NR-PSI [HL08] and GC-PSI [KLS+17], that were implemented and evaluated for this work. These protocols were chosen based on the results from [KLS+17]. In that work, Kiss et al. showed how existing PSI protocols for unequal set sizes could be transformed into a precomputation form, which is well suited for several mobile applications, including private contact discovery.

While all four protocols presented in [KLS+17] are secure against semi-honest adversaries, only NR-PSI and GC-PSI can also be secured against malicious clients in a straightforward way. This is why they were chosen for this work. In addition to improved security, this work also shows how the performance of the protocols can be increased. For both protocols the procedure of informing clients about updates in the server database has been improved. Additionally, a new combination of C-OT and OT-precomputation is presented that reduces communication costs for NR-PSI.

This chapter is organized as follows: Section 4.1 introduces some notations. The overall protocol structure, which is the same for both NR-PSI and GC-PSI is described in Section 4.2. Sections 4.3 and 4.4 then explain the more specific details about NR-PSI and GC-PSI respectively, including the reduction in communication for NR-PSI. The procedure for more efficient server updates, which is also shared by both protocols, is given in Section 4.5. Section 4.6 discusses a critical bottleneck of the protocols and how it can be mitigated in certain scenarios. Finally, Section 4.7 analyzes the security of the protocols.

4.1 Notation

This chapter, as well as the remainder of this work, uses the notation from Table 4.1.

4.2 Common Structure

The basic idea for both protocols is as follows: The server, S, encrypts all items in its database and sends them to the client, C. The secret key used for encryption is only known to S, so C cannot learn anything anything about those items. S and C then perform secure two-party computation, to encrypt C's inputs with S's secret key, without revealing either to the

S:	The server party. Typically with strong computation power and large input set, also referred to as S's database.
C:	The client party. Has a much smaller input set and also less computation power.
N_S:	Number of initial server inputs.
N_S^U:	Number of server inputs that have changed since the last time S and C have executed the PSI protocol.
N_C:	Number of initial client inputs.
N_C^U:	Number of inputs that were added to C' input set since the last protocol execution.
N_C^{max}:	Maximum total number of client inputs. This includes the initial N_C inputs, as well as any added in later runs of the PSI protocol.
$\{x_1, ..., x_{N_S}\}$:	server inputs
$\{y_1, ..., y_{N_S}\}$:	client inputs
$x_i[j]$:	j-th bit of i-th input
l:	length of each input in bits, assuming all inputs have the same length.
1 KB:	1024 bytes
1 MB:	1024 KB

Table 4.1: Notation for PSI protocols

other party. Finally, C obtains the intersection by checking which of its encrypted inputs are contained in the encrypted database it received from S.

Note that it is not necessary for either party to perform any decryptions. Thus, it is fine to hash all inputs to a common size, l, before encrypting them. In fact, any randomization function can be used for encryption, as long as it uses a secret key, only known to S, and is secure, so that C cannot invert it.

As Kiss et al. have shown, when S and C execute a PSI protocol multiple times, it is not necessary to repeat all steps for every run. Instead, the protocols can be put into a pre-computation form, where the operations with high communication and computation costs are only done during the first run, so that future runs become far more efficient. For this purpose, the protocols are split into the following phases which are also summarized in table 4.1:

Base Phase The base phase performs precomputations to make the following encryption of C's inputs as efficient as possible. For each element that shall be encrypted later, some data is generated and stored by both parties. This data will be referred to as *resources* from now on. The computation and communication complexity of this phase is linear in the number of generated resources. Ideally, the base phase is only executed once, generating resources to encrypt up to N_C^{max} inputs. N_C^{max} must be chosen large enough to encompass the initial N_C inputs, as well as any new inputs in future protocol runs.

The base phase is independent of both parties' actual inputs. Thus, it can be executed long before the rest of the protocol. However, in practice it will most likely be executed at the beginning of the first protocol run.

If the generated resources run out, the base phase must be repeated, which undermines the precomputation approach. Thus N_C^{max} should be chosen large enough so that most clients will never need to repeat the base phase.

Setup Phase In this phase, S transmits its encrypted database to C. While in [KLS+17], the database was represented as a bloom filter, this work uses a cuckoo filter instead. Section 4.5 explains this in more detail. Also, in [KLS+17], S would actually encrypt all its inputs during the setup phase using a unique key for each client. Depending on the protocol, this made the setup phase highly computationally expensive. However, Kiss et al. also point out that there is basically no disadvantage when just using the same key for all clients and only generating the encrypted database once. This is the approach chosen for this work, basically eliminating all computation costs from the setup phase.

Still, the size of the filter and thus the communication costs are linear in N_S. In practice, these costs can be very high. Section 4.6 discusses how they can be reduced in certain situations. However, the setup phase only needs to be executed during the first protocol run. C stores the received filter, so that for later runs the far more efficient update phase can be used to inform C about any changes to S's database.

Online Phase During the online phase, the resources from the base phase are used to encrypt C's inputs. In the first protocol run, all of C's initial inputs are encrypted and thus the complexity is linear in N_C. The ciphertexts are cached by C so that during future protocol runs, only the N_C^U inputs, that were added since the last run, have to be encrypted.

Update Phase During the update phase, S informs C about the N_S^U items in its database that have changed since the last protocol run. Due to the use of cuckoo filters, this process is highly efficient and can even handle the deletion of items with ease. More details on the update phase are given in Section 4.5.

4.2.1 Phases in different protocol runs

To further clarify the different phases and their interactions, look at the differences between the first protocol run and any following runs:

Phase	Purpose	Executed during	Complexity
Base Phase	generate resources	first run	$O(N_C^{max})$
Setup Phase	transmit encrypted server database	first run	$O(N_S)$
Online Phase	use resources to encrypt client inputs	every run	$O(N_C)$ or $O(N_C^U)$
Update Phase	inform C about updates to S's database	every run except the first	$O(N_S^U)$

Figure 4.1: Phases of the PSI protocols

First run For the first run, the rather expensive base phase and setup phase are executed to generate resources and transmit S's encrypted database to C. This is followed by an online phase to encrypt C's initial inputs. After that, C can determine the intersection.

Future runs In future runs, an online phase is performed only for any new inputs C might have. Additionally, an update phase is done so that C can update the cached database with any additions or deletions. Both phases are highly efficient, although the actual costs depend on how much the input sets of both parties have changed since the last run.

4.2.2 Differences between the protocols

The only differences between NR-PSI and GC-PSI are in how the items are actually encrypted. This includes the server side encryption when filling the cuckoo filter as well as the oblivious encryption that both parties prepare during the base phase and finish in the online phase. These details are presented in the following sections.

4.3 NR-PSI

NR-PSI was proposed in [HL08], based on the Naor-Reingold PRF from [NR04].

It should be noted, that in [KLS+17], a slightly different definition for this PRF is used. While these differences do not affect security or performance, they do prevent the reduction in communication, presented in Section 4.3.3. For this reason, the definition from [HL08] is used in this work.

For the sake of the security arguments in Section 4.7, any oblivious transfers mentioned in this section use the malicious-secure OT extension protocol from [KOS15] (see Section 2.1.4).

4.3.1 The Original Protocol

Initially, S and all clients must agree on parameters p, q and g, where p is an n-bit prime number, q is an m-bit prime divisor of $p-1$ and g is a generator of a multiplicative subgroup in $\mathbb{Z}_p^*$ of order q.

Additionally, S chooses a secret key, a, which consists of $l+1$ n-bit values $a=(a_0,a_1,...,a_l)$, all of which are randomly chosen from $\mathbb{Z}_q^*$.

To encrypt its own inputs, S calculates:

$$\mathrm{PRF}_a(x_i) = g^{a_0 \cdot A_i} \mod p, \qquad \text{where} \quad A_i = \prod_{j=1}^{l} a_j^{x_i[j]} \mod q, \qquad 1 \le i \le N_S$$

For executing the PRF on C's inputs, we introduce the original version of NR-PSI first, without any precomputation or different phases. For each of C's inputs, S chooses l random n-bit values $r_{i,1},...,r_{i,l}$. S then calculates

$$r_i^{\mathrm{inv}} = (\prod_{j=1}^{l} r_{i,j})^{-1} \mod q, \qquad 1 \le i \le N_C.$$

followed by

$$\tilde{g}_i = g^{a_0 \cdot r_i^{\mathrm{inv}}} \mod p, \qquad 1 \le i \le N_C$$

which are sent to C. Both parties then perform l oblivious transfers per input where S sends these two values:

$$R_{i,j}^0 = r_{i,j} \quad \text{and} \quad R_{i,j}^1 = r_{i,j} \cdot a_j \mod q, \qquad 1 \le i \le N_C, 1 \le j \le l$$

and C chooses to obliviously receive $R_{i,j}^{y_i[j]}$. Next, C multiplies the received values:

$$A_i' = \prod_{j=1}^{l} R_{i,j}^{y_i[j]} = \prod_{j=1}^{l} r_{i,j} \cdot \prod_{j=1}^{l} a_j^{y_i[j]} \mod q, \qquad 1 \le i \le N_C$$

Finally C obtains the PRF results of its inputs as:

$$\mathrm{PRF}_a(y_i) = \tilde{g}_i^{A_i'} = g^{a_0 \cdot \prod_{j=1}^{l} a_j^{y_i[j]}} \mod p, \qquad 1 \le i \le N_C$$

4.3.2 Precomputation Form

The NR-PSI version of Kiss et al. uses OT precomputation (see Section 2.1.3) to split the original protocol into base and online phase. The OTs are precomputed during the base phase, after S has transmitted $\tilde{g}_i$. The OTs are finalized during the online phase, after which C can finish the encryption. The only overhead introduced by this is one correction bit per OT that C must send to S.

4.3.3 Reduced Communication via C-OT

During the OT step in the original version of NR-PSI, the first of the two values, $R^0_{i,j}$ is random, which means C-OT can be applied to reduce communication (cf. Section 2.1.2). This reduces the amount of data S has to send for each input by $l \cdot n$ bits. However, to apply this improvement to the precomputation form from Kiss et al., C-OT and OT precomputation have to be combined, which - to our knowledge - has not been done before.

In both C-OT and OT precomputation the sender, $\mathcal{S}$, obtains two random values, (v_0, v_1) (cf. Section 2.1). In C-OT the sender selects v_0 to be the random message, while v_1 is used as a mask for the transmission of the correlated message. The receiver, $\mathcal{R}$, knows only of the two random values, v_b, depending on its choice bit b. If $b = 0$, $\mathcal{R}$ immediately knows the random message. If $b = 1$, $\mathcal{R}$ can unmask the correlated message.

However, in OT precomputation, $\mathcal{R}$ does not know its actual choice during the precomputation phase. Instead, it obtains v_r, where r is random. For example, if $r = 0$, $\mathcal{R}$ will not be able to receive the correlated message later.

The solution to this problem is for $\mathcal{S}$ to decide which value to choose as random message and which as mask only after receiving $\mathcal{R}$'s correction bit $b' = r \oplus b$. The random message is chosen as $v_{b'}$, while $v_{1-b'}$ is used as mask. This way, it only depends on b, which message $\mathcal{R}$ receives. If $b = 0$, $\mathcal{R}$ always receives the random message. If $b = 1$, it receives the correlated message.

In the following, all four possible combinations of random and actual choice are listed, showing that $\mathcal{R}$ always receives (only) the message of its choice.

- $r = 0, b = 0, b' = 0$: Because $r = 0$, $\mathcal{R}$ knows v_0. Due to $b' = 0$, $\mathcal{S}$ chooses v_0 as random message and v_1 as mask. Thus, $\mathcal{R}$ knows the random message, but cannot unmask the correlated message.
- $r = 0, b = 1, b' = 1$: $\mathcal{R}$ knows v_0. $\mathcal{S}$ chooses v_1 as the random message and v_0 as the mask. So while $\mathcal{R}$ does not know the random message, it is able to unmask the correlated message.
- $r = 1, b = 0, b' = 1$: $\mathcal{R}$ knows v_1, which is also chosen by $\mathcal{S}$ as the random message. Without knowing v_0, $\mathcal{R}$ cannot learn the correlated message.
- $r = 1, b = 1, b' = 0$: $\mathcal{R}$ knows v_1. $\mathcal{S}$ chooses v_0 as random message and v_1 as mask. Thus, $\mathcal{R}$ can unmask the correlated message but does not know the random message.

Performance Gain When applied to NR-PSI in precomputation form, this technique nearly cuts the communication costs during the online phase in half, as S only needs to send half as many messages.

However, because S can now only select the random numbers $r_{i,j}$ after the OTs have been resolved in the online phase, it also has to wait until then, before it can calculate and transmit

$\tilde{g}_i$. This means some computation as well as communication is moved from the base phase to the online phase.

In practice, the added communication is much smaller than what is saved. For each input, n bits are added, while $l \cdot m$ bits are saved. Consider a realistic example with $l = 128$, $n = 2048$ and $m = 224$. In this case, the amount of data, S has to transmit during the online phase, is reduced by about 46%.

The additional computation is only on the server side, which usually has strong computation power. Also, the client has to perform similarly expensive operations anyways. Because of this, the overall computation time should not increase.

A Word on Security This technique does not affect the security of the underlying OT protocol. $\mathcal{R}$ always receives only one message and cannot obtain the other. Also, since r and therefore also b' are random, $\mathcal{S}$ cannot learn anything from the correction bits, just like in classical OT precomputation.

4.4 GC-PSI

GC-PSI was introduced in [KLS+17]. In this protocol, the inputs are encrypted via AES. When S encrypts its own inputs, it does so in a straightforward way by choosing a secret key k and computing $AES_k(x_i), 1 \leq i \leq N_S$.

To encrypt C's inputs, AES is evaluated via garbled circuits (see Section 2.2). The use of garbled circuits, that evaluate AES, in secure multi-party computation was originally proposed in [PSSW09].

In the base phase, S first expands its key k into k', according to the Key-Expansion phase of AES. As this step does not depend on C's input, there is no need to do it inside the garbled circuits. Then S constructs N_C^{max} circuits, using k' as its input value. As was shown in [HS13], garbled circuits for AES can be constructed with 5,120 AND-gates by using the S-box construction from [BP10]. The circuits are then sent to C, together with the wire keys for S's input, k'. In addition, both parties precompute $l \cdot N_C^{max}$ OTs via the malicious-secure OT extension protocol from [KOS15] (see Section 2.1.4).

During the online phase, the precomputed OTs are used so that C obtains its input keys for the circuit. For each input y_i, C resolves l OTs, by using $y_i[j], 1 \leq j \leq l$ as choice bits. C then evaluates the circuit without any further interaction with S.

Note that even though the keys for each input wire are correlated (cf. Section 2.2.2, Free-XOR), C-OT cannot be used here. An optimization as was presented for NR-PSI in Section 4.3.3 is not possible for GC-PSI. When combining C-OT with OT precomputation, S learns the random messages only after the precomputed OTs are resolved in the online phase. However, the random messages are keys for the circuits' input wires and are required for encrypting the

circuits. Thus, the circuits would have to be encrypted and transmitted during the online phase which would completely go against the precomputation approach.

4.5 Efficient Server Updates

When S's database changes between protocol runs, it is usually more efficient to only send these changes to C instead of retransmitting the entire database. This is done during the update phase, as explained in Section 4.2.

This section explores the efficiency of updates for cuckoo filters (Section 4.5.1) and addresses the issue of maximum filter capacity (Section 4.5.2)

4.5.1 Updates for Cuckoo Filters

In [KLS+17], bloom filters were used to represent the encrypted database, but those are not ideal when it comes to updates. To inform C about any added inputs, the most efficient way for bloom filters is usually to just transmit the encrypted input and have C insert it into the filter. Also, deleting inputs is only possible by using bloom filter variations with much larger space requirements (cf. Section 2.3).

With cuckoo filters, all information, that is required to add a new item, is its fingerprint and the index of one of its candidate buckets. As shown in Section 2.3, that information is enough to calculate the second candidate bucket and perform relocations. The same information is also sufficient to delete an item.

To find out how costly these updates are, consider the following extreme values: For a false positive rate of $\epsilon = 10^{-9}$, fingerprints need to be 33 bits long. For a cuckoo filter with up to one billion entries and a bucket size of $b = 4$, $n = 2^{28}$ buckets are required, wich means the bucket indices are 28 bits long. These results show, that for any realistic cuckoo filter, the amount of data per updated item that S has to send to C should be less than 64 bits. To update, e.g., 100,000 items, less than 800 KB would need to be transmitted.

4.5.2 Compression of Sparse Cuckoo Filters

Another issue with updates, that was not considered in [KLS+17], is that both bloom and cuckoo filters actually have limitations on how many additional items may safely be inserted. For a bloom filter, the false positive rate increases, as more items are inserted. If a bloom filter has the optimal size for a required ϵ, no more items can be inserted without breaking that requirement.

For cuckoo filters, there is a strict limit on how many items they can hold. The only way to support significant updates in the future is to start off with a large enough filter, that has a

lot of empty entries. Depending on the scenario, it might be reasonable to use a filter with a capacity several times larger than N_S. However, this would pose a significant overhead during the setup phase.

This can be compensated by compressing the cuckoo filter during transmission. In fact, due to their table structure, cuckoo filters can be compressed in a straight forward way. For each entry, an additional bit is transmitted, indicating if that entry is empty or holds a fingerprint. The entry itself is only transmitted, if it is not empty. Thus, the filter is reduced to a bit map and a list of fingerprints, as illustrated by figure 4.2. To our knowledge, compressing cuckoo filters in this way, has not been done before.

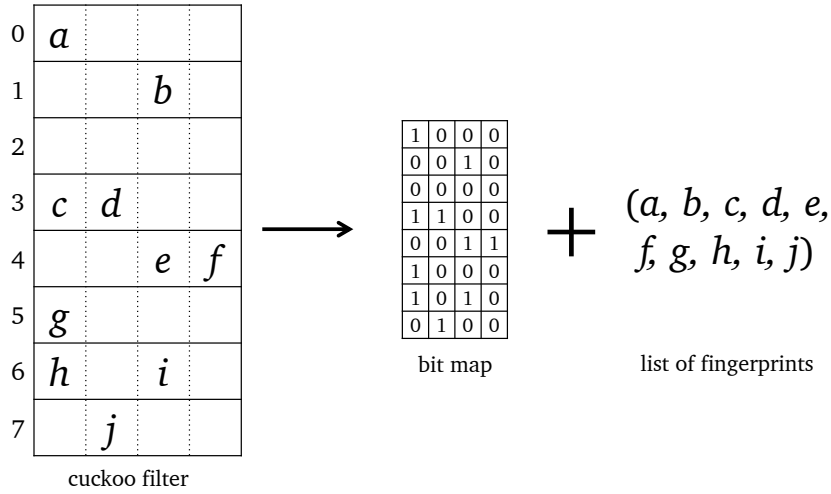


Figure 4.2: Compression of a sparsely filled cuckoo filter.

For evaluating this method, consider a cuckoo filter with $b = 4$ and $\epsilon = 10^{-6}$, resulting in a fingerprint size of 23 bits. To hold one million items, this filter would require $n = 2^{18}$ buckets, or 2^{20} entries. Its size would be 2,944 KB. This filter would have a very high load factor of over 95%, meaning further insertions would be risky.

Now consider a similar filter with four times more space, i.e. 2^{22} entries. Uncompressed, this filter would obviously be four times larger. However, compressed it only requires $2^{22} + 10^6 \cdot 23$ bits $\approx 3,320$ KB. This is only about 13% larger than the original filter.

It should be noted that more efficient compression schemes are possible (although not used in this work). The bit map requires one additional bit per entry, which is b bits per bucket. However, the actual position of each fingerprint within a bucket is not important, so it would be enough to just encode the number of fingerprints (0 to b) in each bucket, requiring only $\log_2(b+1)$ bits per bucket.

4.6 Smaller Cuckoo Filters for Efficient Private Contact Discovery

One of the most critical bottlenecks of the presented PSI protocols is the size of the cuckoo filter that is transmitted during the setup phase. It is linear in N_S, which, in some scenarios, can easily be in the order of several millions. One such scenario is private contact discovery - the main focus of this work.

This section explores how the cuckoo filter size can be reduced for private contact discovery, but the results may also be interesting for other scenarios.

It is difficult to obtain exact user numbers for messenger apps. For Signal, the Google Play Store simply states that it has been downloaded over five million times and Apple's iTunes does not provide any download statistics at all. WhatsApp claims to have over a billion users.

For this section, consider a fictional messenger with 100 million users, so $N_S = 10^8$. Also consider a cuckoo filter with $b = 4$, $n = 2^{27}$ and $\epsilon = 10^{-6}$, which means it has space for up to about half a billion contacts and each fingerprint requires 23 bits. Compressed (see section 4.5.2), this filter would be $4 \cdot 2^27 + 10^8 \cdot 23$ bits ≈ 338.2 MB, which is more than some users might be willing to download solely for private contact discovery.

4.6.1 Adjusting false positive rate

One way of reducing the size of the cuckoo filter is to increase the false positive rate ϵ. However, a too high value might undermine the users privacy. This is because in practice, after both parties finish the PSI protocol, C will most likely reveal the intersection to S. This is not only to eliminate false positives, but also to obtain additional information about these contacts, e.g., what features they support. During this process, any false positive contacts will be revealed to S.

It should also be noted that reducing the filter size like this by any significant amount, will have a large impact on the false positive rate. For example, to reduce the size of a filter with $\epsilon = 10^{-6}$ by about 50%, it is necessary to increase ϵ to 10^{-3}, resulting in 1000 times more false positives.

4.6.2 Splitting the Database into different regions

Many users only have contacts from their own country or continent. For those users, it seems unnecessary to transmit the entire server database. Instead a smaller database containing only users from a specific region can be transmitted. Before performing contact discovery, users would select which regions their contacts are from. Since this might reveal more about the user, than they are comfortable with, there should always be a 'world wide' option. The regions could be countries, continents or anything else, like 'Western Europe'. However, they

should not overlap, so that users do not end up downloading the same contacts multiple times.

To see the effects of such a regional contact discovery, we need to know how app users might be distributed across the world. SensorTower[1] is a webservice, that collects statistics about mobile apps, mainly for more efficient advertisement. Among those statistics are the shares of users the app has in different countries. For the Signal messenger for Android, SensorTower claims that, as of June 2018, the country with the largest share of users (25%) is the USA, followed by Germany with 12% and Russia with 5%. Let us use these shares for our fictional messengers.

If we simply apply these percentages to the original filter's size, we end up with about 84.5 MB / 40.6 MB / 16.9 MB. However, the number of buckets in a cuckoo filter must always be a power of two. Assuming that, like the original filter, each regional filter should have enough space for at least five times the current number of users, this leads to actual (compressed) filter sizes of 84.5 MB / 40.9 MB / 17.7 MB. Due to compression, the restrictions on the filter size barely matter.

To put these results into perspective consider the amount of data users need to download during the installation of a messenger. Signal on Android has a download size of around 30 MB. For users that are sensitive about their privacy, downloading a regional filter of similar size should not be big hurdle.

4.7 Security

This section discusses the security of the presented PSI protocols. It explains why they are secure against semi-honest adversaries and on what assumptions that security is based (Section 4.7.1). It also shows why the protocols are secure even against malicious clients (Section 4.7.2). Finally, Section 4.7.3 demonstrates that the protocols are not secure against malicious servers, if C reveals the intersection to S.

4.7.1 Security against Semi-Honest Adversaries

This section argues about the security of NR-PSI and GC-PSI, against semi-honest adversaries, similar to [KLS+17].

The security of NR-PSI was proven in [HL08] under the decisional Diffie-Hellman assumption. This assumption requires that given random values a, b, c it is impossible to distinguish (g^a, g^b, g^{ab}) from (g^a, g^b, g^c). Additionally, for NR-PSI to be secure against semi-honest adversaries, the OT protocol used must also offer that security. In this work, the OT extension

[1] `https://sensortower.com/`

protocol from [KOS15] is used, which is secure against semi-honest (and even malicious) adversaries.

The security of GC-PSI depends on the generally accepted assumption that AES is secure as well as on the security of garbled circuits, which was proven in [LP09]. Again, it is also required that the OT protocol is secure against semi-honest adversaries.

4.7.2 Malicious Client

A malicious client differs from a semi-honest client in that it may deviate from the protocol description arbitrarily, e.g., by manipulating messages sent to the server. As pointed out in [KLS+17], for both protocols, the only opportunity for such a deviation is during OT computation as these are the only instances where C sends any messages to S. By using the malicious-secure OT extension protocol from [KOS15], both protocols are secured against malicious clients.

4.7.3 Malicious Server

A malicious server can always perform certain trivial attacks, like executing the protocol with a manipulated database. For example, in the malware detection scenario (see Section 1.1), S could prevent a specific user from detecting an infection (maybe at the direction of a government) by simply removing the corresponding entries from the encrypted database before transmitting it. However, such attacks are possible with any PSI protocol.

This section focuses on how a malicious server might learn any of C's inputs that are not part of the intersection. At first this appears impossible because S does not receive any output from the PSI protocols at all. In fact, the only time S receives any information from C is during OT extension, which is malicious-secure.

However, as mentioned in Section 4.6.1, in many practical implementations, after the PSI protocol is completed, C will reveal the intersection to S. If S is capable of enforcing false positives for some or all of C's inputs, S will learn these inputs as soon as C reveals them to S. As shown in the following, enforcing false positives is very easy in both PSI protocols. In fact, whenever C is trying to encrypt one of its inputs, y_i, a malicious S can make sure that the result is not the encrypted form of y_i but in fact the encrypted form of a different input, z_i, which can be freely chosen by S.

NR-PSI During the online phase, while resolving the precomputed OTs, S sets $R^0_{i,j} = R^1_{i,j} = r_{i,j}$. This renders C's choices irrelevant and always leads to A'_i being equal to $\prod_{j=0}^{l} r_{i,j}$. Note that C

has no way of detecting this. S also does not send $\tilde{g}_i$, but instead sends

$$\hat{g}_i = \tilde{g}_i^{\hat{A}_i} \qquad \text{where} \quad \hat{A}_i = \prod_{j=1}^{l} a_j^{z_i[j]}$$

Now, when C calculates $\hat{g}_i^{A'_i}$, A'_i and r_i^{inv} cancel out and the result is $\mathrm{PRF}_a(z)$.

GC-PSI While encrypting one of C's inputs during the online phase, S would normally input two different keys for each input wire, representing 0 and 1. C makes its choice, depending on the input bit $y_i[j]$. A malicious S however, can select one of the keys according to $z_i[j]$ and input that key for both choices, practically replacing $y_i[j]$ with $z_i[j]$.

These results show, that both protocols are not secure against malicious servers if C reveals the intersection.

5 Implementation

This chapter presents all software components that were implemented for this work to evaluate the PSI protocols from Chapter 4.

All implementations were done within the ABY framework[1]. ABY implements different secure two-party computation protocols as well as means to combine them [DSZ15]. For this work, the relevant components from ABY are its implementation of Yao's garbled circuit protocol, the AES circuit implementation, the OT extension library[2] (which can also be used separately) and its utility library, which, among other things, offers cryptographic operations and networking functionality.

For its garbled circuit implementation, ABY uses most of the optimizations listed in Section 2.2.2, with the exception of pipelining. Also, ABY makes use of AES-NI (if available on the current platform) for AES operations (cf., Section 2.2.2, Fixed-Key Blockcipher).

This chapter starts with explaining the selected security parameters in Section 5.1. Section 5.2 shows our implementation of the malicious-secure OT extension scheme from [KOS15], followed by our implementation of Cuckoo Filters in Section 5.3. Sections 5.4 and 5.5 present our implementations of NR-PSI and GC-PSI, respectively. Section 5.6 discusses our test application and its port to Android. Finally, Section 5.7 explains, how we integrated our implementations into the Signal Android app.

5.1 Security Parameter Choices

For the malicious-secure OT extension protocol from [KOS15], we use $\kappa = 128$ bits as the symmetric security parameter. This is the same choice as in [KLS+17] and according to the NIST, it provides security beyond the year 2030 [Bar16]. This protocol also requires s additional OTs, to prevent the checksums from leaking any information. In this work, we use $s = 64$, which was also used in [KOS15].

For the PSI protocols, we use an input length of $l = 128$ bits and bring all inputs to that size by hashing them with SHA-256 and truncating the result.

[1] `https://github.com/encryptogroup/ABY`
[2] `https://github.com/encryptogroup/OTExtension`

In [KLS+17], the sizes of p and q were chosen as $n = 2048$ bits and $m = 256$ bits. According to [Bar16], this choice for p is secure until 2030. For better comparison, we aim for the same level of security as [KLS+17] and also chose $n = 2048$ bits. However, for q we choose $m = 224$ bits, which is in agreement with the NIST recommendations for this level of security. For long term security, at least 3072 bits for p and 256 bits for q should be used.

5.2 Malicious-Secure OT Extension from [KOS15]

The OT extension library used by ABY only supported protocols with security against semi-honest adversaries. An implementation of the malicious-secure protocol from [KOS15] can be found in the libOTe[3] library. However, that library heavily relies on Intel-specific machine instructions and does not run on the ARM processors used in most smartphones. Because of that, we added our own implementation of [KOS15] to ABY's OT extension library, building up on its implementation of the protocol from [IKNP03]. As described in Section 2.1.4, the critical changes [KOS15] makes over [IKNP03] are the required checksums as well as s additional OTs.

Efficient Checksums The performance critical parts of the checksum calculation are the multiplications in the finite field $\mathbb{F}_{2^\kappa}$. It was suggested in [KOS15], to use carry-less multiplications for these and to omit the following reduction by the finite field polynomial. This doubles the checksum size but allows for much more efficient implementations.

On Intel platforms, the `PCLMUL` instruction can be used to perform carry-less multiplications very efficiently. A documentation can be found in [GK10]. This was already suggested in [KOS15] and is also used in libOTe. We have also included it in our implementation, but it is only available on Intel platforms. For ARM, we had to write a software-based version. The newer 64-bit ARMv8 architecture actually has hardware instructions for carry-less multiplication [arm14]. We did not use these however, because our testing device has a 32-bit ARMv7 CPU.

Additional OTs The existing OT extension protocols in ABY's library calculate the OTs in chunks. This was described at the end of Section 2.1.1 in the context of parallelization, but in ABY it is also used to reduce the maximum memory consumption. While they share the same base OTs, each of these chunks can be regarded as an individual protocol execution. And with [KOS15], they would also each need their own checksum verification and s additional OTs. To avoid these OTs, we had to make sure that no chunk enters the final masking step, before the checksums are verified. This means, the values used to generate the masks must all remain in memory until that point.

[3] `https://github.com/osu-crypto/libOTe`

However, for multithreading we accept the small overhead of s additional OTs per thread. This is a choice in favor of a cleaner implementation, without having to synchronize the threads before the final masking step.

5.3 Cuckoo Filters

Bin Fan et al., the inventors of cuckoo filters, also offer an open source implementation[4]. It focuses on efficiency while limiting flexibility. Instead of arbitrary false positive rates, only certain fingerprint sizes are supported that allow for an efficient implementation. For this reason, it was not used for this work. Still it served as a guideline for our own implementation.

Our implementation is based on a bit vector class that is part of ABY's utility library. It allows us to pack fingerprints of arbitrary size close together and handles any alignment issues (when a fingerprint starts or ends in the middle of a byte). It minimizes memory consumption, while slightly increasing computation costs. However, the cuckoo filters are not a bottleneck in our PSI protocols, so we consider this an acceptable trade-off.

For all hashing operations in our cuckoo filters, we use SHA-256, truncated to the desired length.

5.4 NR-PSI

For the most part, the implementation of NR-PSI is straightforward. For the modular arithmetic operations, the GNU Multiple Precision Arithmetic Library (GMP)[5] is used, which was already a dependency of ABY. Network communication was realized with the help of ABY's utility library.

Multithreading To increase performance of NR-PSI, multithreading support was included in the base and online phase, as these are the only phases that pose any significant computation costs. With the improvements from Section 4.3.3, the base phase only performs OT precomputation. Here we use the existing multithreading support of the OT extension library. For the online phase, each input is processed independently. Thus, the work can easily be split into multiple threads.

[4] `https://github.com/efficient/cuckoofilter`
[5] `https://gmplib.org/`

5.5 GC-PSI

For GC-PSI, we use the AES circuit implementation that already comes with ABY. It is constructed as shown in [HS13], using 5,120 AND gates. It also uses ABY's SIMD feature, where one circuit can be used to process multiple inputs. This slightly improves performance, e.g., by calculating the base OTs, for the OT extension protocol, only once.

We use this feature in our implementation, to make the number of inputs per circuit freely configurable. However, this can lead to wasted resources. If, for example, the circuits, constructed in the base phase, are designed to process 100 inputs, but the client only has 50 inputs during the online phase, it still must 'use up' an entire circuit.

In practice, if N_C is known during the base phase, we suggest constructing one circuit for these N_C inputs, that will be used in the first online phase, and $N_C^{max} - N_C$ single-input circuits for future protocol runs.

Splitting Circuit Execution When executing a garbled circuit in ABY, encryption and evaluation are tied together and cannot be run separately. For this work, it was necessary to split up this process and perform the encryption in the base phase and the evaluation in the online phase.

Internally, ABY already splits the execution into an offline and an online phase. However, this did not quite fit our needs. In addition to making this split available from the outside, we had to apply two more changes. First, we moved the transmission of the garbler's (i.e., *S*'s) input keys from online to offline phase. These keys represent *S*'s expanded AES key, which never changes, so there is no need to delay their transmission to the online phase. Second, we added a way to set the evaluator's (i.e., *C*'s) input between offline an online phase instead of during circuit construction.

Construction / Destruction Overhead In ABY, it is not possible to reuse the same structure for multiple circuits. It is possible to reuse a circuit after it has been evaluated, but that is of no use to us. When creating multiple circuits in the base phase, each circuit must be created from scratch. Since each added gate performs its own memory allocations, circuit construction takes a very long time (up to a second per circuit, even longer on smartphones). To a smaller extent, this is also true for the destruction of used circuits in the online phase.

Moreover, the large amount of memory allocations severely limits the number of circuits that can exist at the same time. For example, more than a handful of circuits are enough to crash our implementation on Android.

In a production-ready implementation, the circuit structure would only be built once, probably at program start. When constructing a new circuit instance, only one buffer for all keys would have to be allocated.

Because this overhead is ABY specific and not fundamental to GC-PSI, we remove the construction and destruction times from our measurements in Chapter 6.

Multithreading For parallelization we use ABY's multithreading feature. However, as we found out during our evaluation, this only applies to the OT extension protocol, not the encryption or evaluation of the circuit itself. Unfortunately, when we noticed this, we did not have enough time to change our code. Thus, our GC-PSI implementation is practically not parallelized.

5.6 Test Application

In this section we describe our test application, which we used to perform the evaluation in Chapter 6. The following sections explain our initial command line application, as well as its Android port.

5.6.1 Command-Line Application

Our command-line application is written as an ABY project and can act as server or client.

The server side can connect to multiple clients at the same time and most configuration is done on the client. This is useful, because it allows us to test different parameters without restarting the server. Clients can even choose the PSI protocol, because the server keeps two encrypted database at the same time, one for each protocol.

The different phases can be triggered individually on the client side at any time and after each phase, both parties display the required runtime and communication.

For both parties, inputs can be added or removed at any point, by providing a file with a list of inputs.

For NR-PSI, our client can also store a session to persistent memory and then reload it later. This includes the cuckoo filter as well as the resources generated during the base phase. Unfortunately, ABY does not support storing and loading of garbled circuits, so we cannot support this feature for GC-PSI.

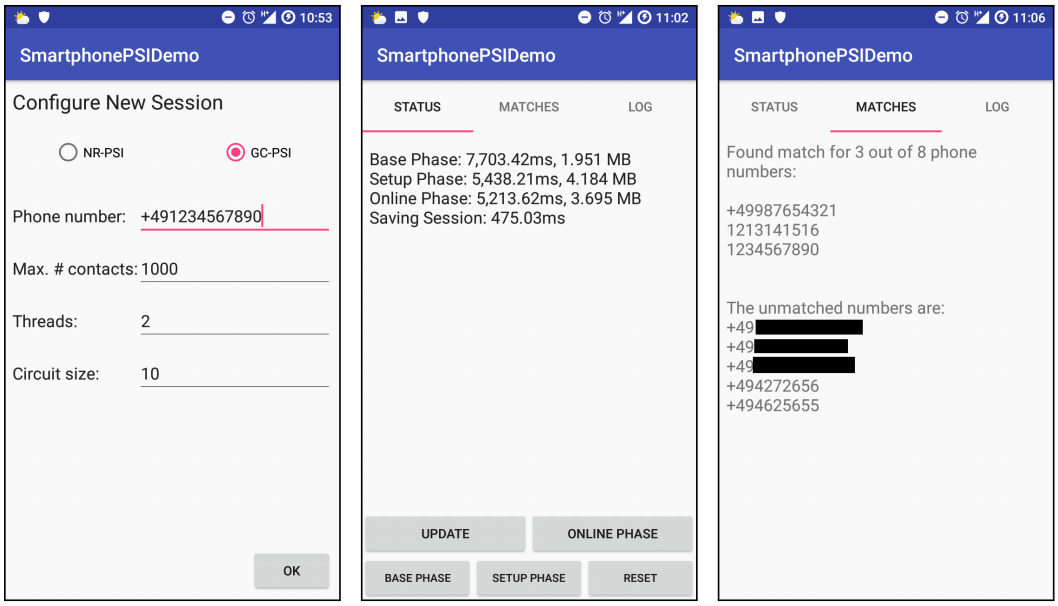
5.6.2 Port to Android

ABY's code is architecture independent, so it runs on ARM without any modifications. The larger hurdle was to integrate ABY into Androids build process for native code. At the time we created our implementation, ABY could only be built via a rather complicated makefile. To make ABY compatible with the Android NDK, we created a suitable CMake script. For the

dependent libraries, openssl and gmp, we used existing precompiled versions for Android[6,7]. We wrote a detailed manual, that allows others to effortlessly integrate ABY into their Android app.

From our test application, we only brought the client side to Android. Instead of reading inputs from files, our app reads the phone numbers from the device's address book and uses them as inputs. It also registers the device's own phone number in the server database when the client establishes a connection, so that it can be discovered by other clients.

In general, the app offers the same features as the command line application, although through a graphical UI, which is shown in Figure 5.1.



(a) Choose parameters. **(b)** Trigger phases and display measurements. **(c)** Show matches.

Figure 5.1: Screenshots of the PSI test application on Android.

5.7 Signal Integration

This chapter describes how we integrated our PSI protocol implementations into the Signal Android app.

Due to time constraints and because setting up our own Signal server proved to be more difficult than expected, we decided against integrating our protocols into Signal's server.

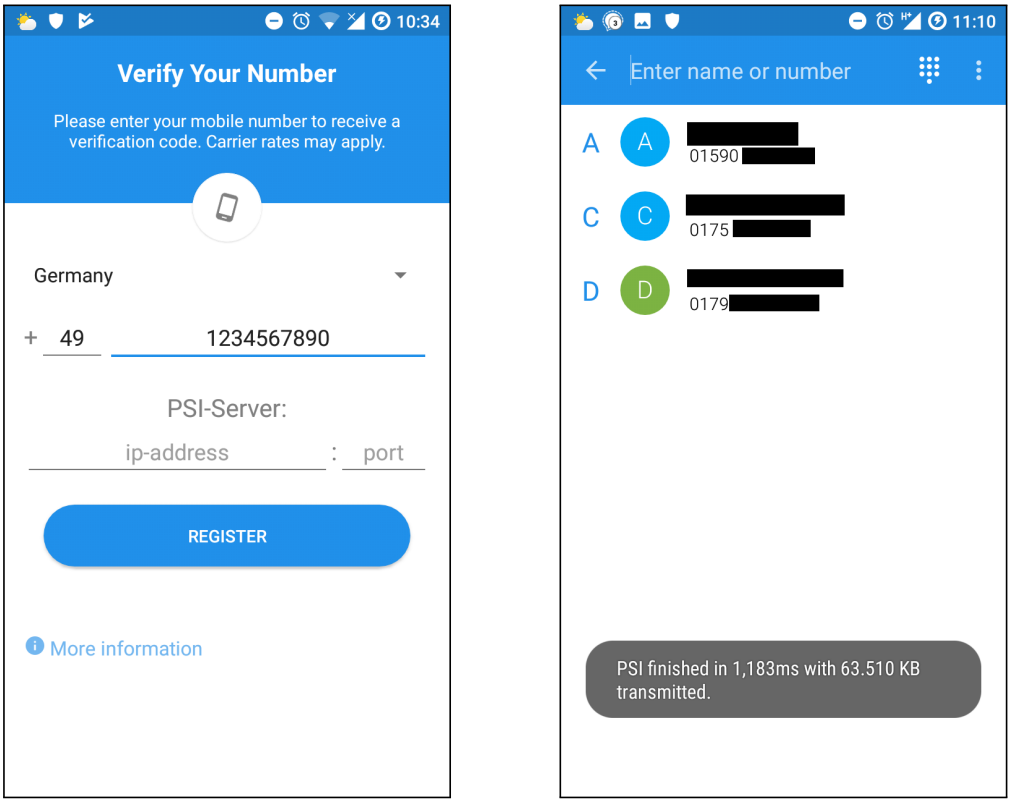
[6] https://github.com/r4sas/OpenSSL-1.1-Android-Prebuilt
[7] https://github.com/Rupan/gmp

Instead, the modified app executes our protocol first. The resulting intersection is then used as input for Signal's own contact discovery protocol.

Note that a very similar approach would have been necessary, even if we integrated our protocol into the Signal server. In Signal's current hashing-based contact discovery protocol, the server does not only send the intersection to the client, but also additional information about each matching contact [Ope14a]. To obtain this information with our PSI protocols, the clients would have to send the matching contacts to the server, which would then not only respond with the additional information, but also inform the client about which contacts were actually found in the database, to eliminate false positives. This additional step is basically what we are doing when we input the intersection into Signal's own contact discovery.

The most important addition to Signal's code base, aside from copying most of the code from our test app, is a helper class, which manages sessions (i.e., storing/loading the cuckoo filter and resources to/from persistent memory) and automatically runs the necessary phases, given only a list of inputs. Because of the limitations of our GC-PSI implementation (see Section 5.5), the helper always uses NR-PSI. For the first base phase, $N_C^{max} = \max(N_C, 1000)$ is chosen and if the resources ever run out, a new base phase is run automatically. The actual integration of our protocols into Signal's contact discovery scheme is done with only a single line of code, where we take the list of phone numbers, Signal has prepared, pass it to our helper class and finally replace it with the obtained intersection.

These changes are completely transparent to the user, which demonstrates the practicality of our implementation. Only two changes to the UI were made and those are only for debugging purposes. In the user registration process on first app start, we added input fields for IP-address and port of our PSI server. This is merely for our convenience. Also, we added a small Toast message that is displayed after every run of our PSI protocol and shows how much time and communication was required. These changes can be seen in Figure 5.2.



(a) Specify PSI server.

(b) Show required runtime and communication.

Figure 5.2: UI changes to Signal.

6 Evaluation

Here we present the evaluation results for our implementations from Chapter 5.

This chapter begins with describing our test scenarios in Section 6.1. The evaluation of our implementation of the malicious-secure OT extension protocol from [KOS15] is given in Section 6.2. Section 6.3 evaluates the construction of the encrypted database. Setup and update phase are evaluated together in Section 6.4, because their performance depends on the same parameters. The same is true for base and online phase, which are evaluated in Section 6.5.

6.1 Test Scenarios

In all our tests we used the same desktop computer to run the server. It uses an Intel i5-4570 CPU with 3.2 GHz. The server is run inside a virtual machine (using VirtualBox[1]) with Linux Mint as the operating system. The virtual machine uses four cores and 5 GB of RAM. AES-NI is available inside the virtual machine.

For our client device, we distinguish the following three scenarios:

LAN The client device is a laptop with an Intel i5-2430M CPU with 2.4 GHz and is connected via 100 Mbit Ethernet to the same LAN as the desktop computer that runs the server. The client runs inside a virtual machine with two cores and 4 GB of RAM. This setting has the lowest communication overhead.

WiFi The client is a Samsung Galaxy S3 with a quad-core 1.4 GHz Cortex-A9 CPU. It is connected to the server via WiFi (IEEE 802.11 n, 72 Mbps). This scenario, with moderate communication overhead, is especially relevant for the rather communication intensive first protocol run, because users may not want to download that much data over their mobile network.

[1] `https://www.virtualbox.org/`

Mobile Network This scenario uses the same Samsung Galaxy S3 as the WiFi scenario. Here however, the smartphone connects to the server via its mobile network, which is a 3G network with HSPA+. This is not only the scenario with the lowest bandwidth (8 Mbps download / 2.8 Mbps upload), but it also has the highest latency at 75 ms.

6.2 Evaluating OT extension

This section compares our implementation of the malicious secure OT extension protocol from [KOS15] to the implementation of [IKNP03], that comes with ABY's OT extension library. As the communication overhead is well defined and, for large numbers of OTs, neglible, we only focus on computation time here. To further highlight the differences in computation, we only consider the LAN scenario, where communication costs are smallest.

As OT flavor, we chose R-OT because this skips the final masking step that would be equal for both protocols. The OT precomputations in our implementations of the PSI protocols also use R-OT.

For the checksum calculations, we test both the hardware based implementation, that uses Intel's `PCLMUL` instruction, as well as our software based implementation.

Each experiment was repeated 10 times. The average results are shown in Table 6.1. All protocols show very similar performance. Our software implementation does not seem to introduce a large overhead, compared to using hardware instructions.

For large OT numbers, the malicious secure protocol is slightly slower, due to the addition of checksums. However, for smaller numbers, it is, surprisingly, faster than the semi-honest protocol. We have repeated the tests to make sure that this is not a random fluctuation. We do not fully understand the reason for this result. We can only speculate that it is caused by our changes to how chunks are handled (see Section 5.2 Additional OTs).

Number of OTs	10,000	100,000	1,000,000	10,000,000
[IKNP03]	107.9	619.3	4,773	42,509
[KOS15]	103.4	596.8	4,957	47,348
[KOS15] with `PCLMUL`	96.3	575.6	4,945	47,137

Table 6.1: Runtime of R-OT extension protocol in milliseconds.

6.3 Generating the Encrypted Database

In our implementation, the server database is encrypted outside the actual PSI protocols. However, some implementations could choose to use a different secret key for each client

and encrypt the database in every setup phase. One reason for this could be that the server provides customized databases for each client and wants to prevent clients from figuring out how similar their databases are.

For each protocol we measured the time it took to hash and encrypt 10 million inputs. We also measured the time necessary to insert the encrypted items into cuckoo filters with different false positive rates. Each filter had $n = 2^{24}$ buckets of size $b = 4$. Because the ciphertexts in NR-PSI are longer than in GC-PSI, the insertion also takes more time. The results are shown in Table 6.2.

	Hash & Encryption	Insertion into cuckoo filter		
		$\epsilon = 10^{-3}$	$\epsilon = 10^{-6}$	$\epsilon = 10^{-9}$
NR-PSI	4,828,446	11,010	11,198	11,301
GC-PSI	3,314	3,750	3,815	3,977

Table 6.2: Times in milliseconds required to build the encrypted database.

Our results are for a single-threaded implementation, but since each item can be encrypted independently from the others, a highly efficient multi-threaded implementation would be possible. For the insertion into the cuckoo filter, at least the generation of fingerprints and bucket indices can be parallelized.

6.4 Evaluation of Setup and Update Phase

The setup and update phase are the only phases in both PSI protocols that depend on the false positive rate ϵ of the cuckoo filter. For that reason, they are evaluated together in this section.

As in Section 6.3, we chose $n = 2^{24}$ and $b = 4$ for the cuckoo filters. In the update phase, we insert 100,000 new items into the filters. To account for any implementation specific overhead, not only did we measure the runtime, but also the number of transmitted bytes, instead of using theoretic values.

Table 6.3 shows the communication costs, while Table 6.4 shows the runtimes for all three scenarios.

	Setup Phase			Update Phase		
False positive rate	10^{-3}	10^{-6}	10^{-9}	10^{-3}	10^{-6}	10^{-9}
Data transmitted	23.497 MB	35.418 MB	47.34 MB	452 KB	574 KB	696 KB

Table 6.3: Communication for setup and update phase.

	Setup Phase			Update Phase		
False positive rate	10^{-3}	10^{-6}	10^{-9}	10^{-3}	10^{-6}	10^{-9}
LAN	6,142	8,644	10,993	145	166	187
WiFi	11,235	14,143	18,605	344	366	435
mobile	34,077	49,788	74,556	2,747	2807	3269

Table 6.4: Runtimes in milliseconds for setup and update phase.

Even though our cuckoo filters have space for five times their current number of items, in their compressed form they are smaller (in relation to N_S) than the bloom filters used in [KLS+17]. Also, their results for update sizes are multiple times larger than ours.

Compression Overhead We have observed that our implementation of cuckoo filter compression and decompression introduces a noticeable computation overhead, especially on the smartphone. Our findings are given it Table 6.5. To minimize this overhead in future implementations, we suggest a streaming approach, where parts of the filter are compressed/decompressed while other parts are still transmitted. We believe that the overhead can be nearly eliminated this way, reducing the setup phase runtimes accordingly.

False positive rate	10^{-3}	10^{-6}	10^{-9}
Compression on Desktop	1,086	1,252	1,416
Decompression on Laptop	598	759	875
Decompression on Smartphone	3,442	4,240	5,081

Table 6.5: Cuckoo filter compression and decompression times in milliseconds.

6.5 Evaluation of Base and Online Phase

Base and online phase are evaluated together in this section, because they depend on the same parameters, namely the number of threads and, for GC-PSI, the number of inputs per circuit.

For NR-PSI we ran both phases with $N_C^{max} = N_C = 1000$ inputs. As mentioned in Section 5.5, our GC-PSI implementation only supports a small number of circuits, so we only used five circuits at a time for our measurements. Also, we removed the ABY specific construction and destruction overhead from our results.

To see the effects of different input numbers per circuit, we ran our tests once for single-input circuits and once for circuits with 10 inputs. We refer to the later variant as GC-PSI-10 in the tables.

All results in this chapter are given relative to a single input. This is because both base and online phase are mostly linear in the number of inputs (except for the calculation of base OTs in the base phase).

Table 6.6 shows the communication costs for each phase and protocol. All results come from measurements and are not theoretic values. Tables 6.7, 6.8 and 6.9 show the runtimes in the LAN, WiFi and mobile network scenario, respectively.

	Base Phase	Online Phase
NR-PSI	2,041	3,874
GC-PSI	497,372	4,121
GC-PSI-10	200,894	4,113

Table 6.6: Communication in base and online phase. All results are given in bytes per item.

	Base Phase		Online Phase	
Threads	1	2	1	2
NR-PSI	0.950	0.901	0.923	0.778
GC-PSI	144	140	63	60
GC-PSI-10	45	44	8	8

Table 6.7: Runtimes in milliseconds for base and online phase in the LAN scenario.

	Base Phase		Online Phase	
Threads	1	2	1	2
NR-PSI	2.277	1.697	4	2.235
GC-PSI	126	129	52.2	49
GC-PSI-10	59	61	12	16

Table 6.8: Runtimes in milliseconds for base and online phase in the WiFi scenario.

The communication costs for GC-PSI appear to be abnormally large. This is because, in ABY, each circuit runs its own OT extension protocol, including base OTs. This is an unnecessary overhead and unfortunately it also makes the runtimes much larger than necessary. ABY's SIMD feature mitigates these effects to some degree in GC-PSI-10. Overall, the results for GC-PSI and GC-PSI-10 are far worse than those for NR-PSI. According to [KLS+17], GC-PSI should, at least in the LAN setting, where both devices support AES-NI, have a much faster online phase. We assume that, despite our efforts, we were not able to fully eliminate ABY's implementation specific overhead from our measurements.

For NR-PSI, all our runtimes are a multiple times smaller than those presented in [KLS+17].

	Base Phase		Online Phase	
Threads	1	2	1	2
NR-PSI	9.300	9.052	5.070	5.738
GC-PSI	1,062	1050	244	228
GC-PSI-10	322	293	51	54

Table 6.9: Runtimes in milliseconds for base and online phase in the mobile network scenario.

Multithreading For NR-PSI, the benefits of multithreading are most noticeable in the WiFi scenario. For mobile networks, the communication takes so much time, that computational improvements have less of an impact. In the LAN scenario, even though communication is much faster, the client has enough computation power to make communication the bottleneck again. We conclude that multithreading is only useful in certain scenarios. But given the ease of its implementation and its substantial benefit under the right conditions, we certainly recommend it for any practical implementation.

For GC-PSI, multithreading shows no benefits. As mentioned in Section 5.5, this is a flaw in our implementation. We expect that in the online phase, proper multithreading can lead to improvements similar to the results for NR-PSI. Because the base phase for GC-PSI is so communication heavy, we do not expect significant benefits through multithreading.

7 Conclusion

We have motivated the need for efficient PSI protocols for unequal set sizes with a survey, showing that private contact discovery protocols used in practice offer little privacy.

We have presented several improvements for two existing PSI protocols and made them secure against malicious clients.

We have implemented these protocols within the ABY framework and have ported our implementation to Android. We also integrated it into the Signal messenger app. Together with the results from our evaluation, this integration demonstrates that the protocols can be used in practice.

A remaining bottleneck is the encrypted database transmitted during the first protocol run. While we suggested several ways to reduce its size, our current protocols would not scale well for messengers like WhatsApp with over a billion users. We leave this point as future work.

List of Figures

List of Tables

Bibliography

[ALSZ13] G. ASHAROV, Y. LINDELL, T. SCHNEIDER, M. ZOHNER. **"More efficient oblivious transfer and extensions for faster secure computation"**. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM. 2013, pp. 535–548 (cit. on pp. 5, 7, 10, 12).

[ALSZ15] G. ASHAROV, Y. LINDELL, T. SCHNEIDER, M. ZOHNER. **"More efficient oblivious transfer extensions with security for malicious adversaries"**. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 673–701 (cit. on p. 9).

[ALSZ17] G. ASHAROV, Y. LINDELL, T. SCHNEIDER, M. ZOHNER. **"More efficient oblivious transfer extensions"**. In: *Journal of Cryptology* 30.3 (2017), pp. 805–858 (cit. on pp. 5, 7).

[arm14] ARM. *ARM Compiler armasm Reference Guide: PMULL, PMULL2 (vector)*. 2014. URL: `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0802b/PMULL_advsimd_vector.html` (visited on 07/19/2018) (cit. on p. 40).

[Bar16] E. BARKER. **"National Institute of Standards and Technology Special Publication 800-57 Part 1, Revision 4. Recommendation for Key Management"**. In: *enero de* (2016) (cit. on pp. 39 sq.).

[Bea95] D. BEAVER. **"Precomputing oblivious transfer"**. In: *Annual International Cryptology Conference*. Springer. 1995, pp. 97–109 (cit. on p. 8).

[Bea96] D. BEAVER. **"Correlated pseudorandomness and the complexity of private computations"**. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM. 1996, pp. 479–488 (cit. on p. 5).

[BHKR13] M. BELLARE, V. T. HOANG, S. KEELVEEDHI, P. ROGAWAY. **"Efficient garbling from a fixed-key blockcipher"**. In: *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE. 2013, pp. 478–492 (cit. on p. 13).

[Blo70] B. H. BLOOM. **"Space/time trade-offs in hash coding with allowable errors"**. In: *Communications of the ACM* 13.7 (1970), pp. 422–426 (cit. on p. 15).

[BMR90] D. BEAVER, S. MICALI, P. ROGAWAY. **"The round complexity of secure protocols"**. In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. ACM. 1990, pp. 503–513 (cit. on p. 12).

[BP10] J. BOYAR, R. PERALTA. **"A new combinational logic minimization technique with applications to cryptology"**. In: *International Symposium on Experimental Algorithms*. Springer. 2010, pp. 178–189 (cit. on p. 31).

[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*. Springer. 2012, pp. 416–432 (cit. on p. 10).

[CLR17] H. CHEN, K. LAINE, P. RINDAL. **"Fast private set intersection from homomorphic encryption"**. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2017, pp. 1243–1255 (cit. on p. 4).

[CO15] T. CHOU, C. ORLANDI. **"The simplest protocol for oblivious transfer"**. In: *International Conference on Cryptology and Information Security in Latin America*. Springer. 2015, pp. 40–58 (cit. on pp. 5 sq.).

[Con17] CONFIDE, INC. *Confide privacy policy*. 2017. URL: `https://getconfide.com/privacy` (visited on 05/03/2018) (cit. on p. 20).

[Deu14] DEUTSCHE POST AG. *SIMSme privacy policy*. 2014. URL: `https://www.sims.me/data-protection` (visited on 05/03/2018) (cit. on p. 21).

[Dis17] DISTRICT COURT BAD HERSFELD. *Court order from 20th March 2017, file number F 111/17 EASO*. 2017. URL: `https://www.jurion.de/urteile/ag-bad_hersfeld/2017-03-20/f-111_17-easo/` (visited on 04/30/2018) (cit. on p. 1).

[DRRT18] D. DEMMLER, P. RINDAL, M. ROSULEK, N. TRIEU. **"PIR-PSI: Scaling Private Contact Discovery"**. In: (2018) (cit. on p. 4).

[DSZ15] D. DEMMLER, T. SCHNEIDER, M. ZOHNER. **"ABY-A Framework for Efficient Mixed-Protocol Secure Two-Party Computation."** In: *NDSS*. 2015 (cit. on p. 39).

[DW07] M. DIETZFELBINGER, C. WEIDLING. **"Balanced allocation and dictionaries with tightly packed constant size bins"**. In: *Theoretical Computer Science* 380.1-2 (2007), pp. 47–68 (cit. on p. 13).

[EC16] EUROPEAN PARLIAMENT, COUNCIL OF THE EUROPEAN UNION. *General Data Protection Regulation*. 2016. URL: `https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=celex%3A32016R0679` (visited on 04/25/2018) (cit. on p. 17).

[EGL85] S. EVEN, O. GOLDREICH, A. LEMPEL. **"A randomized protocol for signing contracts"**. In: *Communications of the ACM* 28.6 (1985), pp. 637–647 (cit. on p. 5).

[ELE] ELEET LP. *Eleet privacy policy*. URL: `https://eleet.im/privacy/` (visited on 05/03/2018) (cit. on p. 20).

[FAKM14] B. FAN, D. G. ANDERSEN, M. KAMINSKY, M. D. MITZENMACHER. **"Cuckoo filter: Practically better than bloom"**. In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM. 2014, pp. 75–88 (cit. on pp. 3 sq., 13 sqq.).

[FCAB00] L. FAN, P. CAO, J. ALMEIDA, A. Z. BRODER. **"Summary cache: a scalable wide-area web cache sharing protocol"**. In: *IEEE/ACM transactions on networking* 8.3 (2000), pp. 281–293 (cit. on p. 15).

[G D17] G DATA SOFTWARE AG. *G DATA Secure Chat repository*. 2017. URL: `https://github.com/GDATASoftwareAG/SecureChat` (visited on 05/03/2018) (cit. on p. 21).

[GK10] S. GUERON, M. E. KOUNAVIS. **"Intel® carry-less multiplication instruction and its usage for computing the GCM mode"**. In: *White Paper* (2010) (cit. on p. 40).

[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*. ACM. 1987, pp. 218–229 (cit. on p. 10).

[Gue10] S. GUERON. **"Intel® Advanced Encryption Standard (AES) New Instructions Set"**. In: *Intel Corporation* (2010) (cit. on p. 13).

[Ham18] HAMBURG HIGHER ADMINISTRATIVE COURT. *Press Release on file number 5 Bs 93/17*. 2018. URL: `https://justiz.hamburg.de/aktuelles/10550476/pressemitteilung/` (visited on 04/24/2018) (cit. on p. 1).

[HEKM11] Y. HUANG, D. EVANS, J. KATZ, L. MALKA. **"Faster Secure Two-Party Computation Using Garbled Circuits."** In: *USENIX Security Symposium*. Vol. 201. 1. 2011, pp. 331–335 (cit. on p. 13).

[HL08] C. HAZAY, Y. LINDELL. **"Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries"**. In: (2008) (cit. on pp. 25, 28, 35).

[Hoc] HOCCER BETRIEBS GMBH. *Hoccer Privacy & Security Statement*. URL: `https://hoccer.com/hoccer-xo-privacy-security-statement/` (visited on 05/03/2018) (cit. on p. 21).

[HS13] W. HENECKA, T. SCHNEIDER. **"Faster secure two-party computation with less memory"**. In: *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM. 2013, pp. 437–446 (cit. on pp. 31, 42).

[IKNP03] Y. ISHAI, J. KILIAN, K. NISSIM, E. PETRANK. **"Extending oblivious transfers efficiently"**. In: *Annual International Cryptology Conference*. Springer. 2003, pp. 145–161 (cit. on pp. 5, 9, 40, 48).

[IR89] R. IMPAGLIAZZO, S. RUDICH. **"Limits on the provable consequences of one-way permutations"**. In: *Proceedings of the twenty-first annual ACM symposium on Theory of computing*. ACM. 1989, pp. 44–61 (cit. on p. 5).

[KGG+18] P. KOCHER, D. GENKIN, D. GRUSS, W. HAAS, M. HAMBURG, M. LIPP, S. MANGARD, T. PRESCHER, M. SCHWARZ, Y. YAROM. **"Spectre attacks: Exploiting speculative execution"**. In: *arXiv preprint arXiv:1801.01203* (2018) (cit. on p. 4).

[KK13] V. KOLESNIKOV, R. KUMARESAN. "Improved OT extension for transferring short secrets". In: *Advances in Cryptology–CRYPTO 2013*. Springer, 2013, pp. 54–70 (cit. on pp. 5, 9).

[KKRT16] V. KOLESNIKOV, R. KUMARESAN, M. ROSULEK, N. TRIEU. **"Efficient batched oblivious PRF with applications to private set intersection"**. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 818–829 (cit. on p. 2).

[KLS+17] Á. KISS, J. LIU, T. SCHNEIDER, N. ASOKAN, B. PINKAS. **"Private set intersection for unequal set sizes with mobile applications"**. In: *Proceedings on Privacy Enhancing Technologies* 2017.4 (2017), pp. 177–197 (cit. on pp. 2 sq., 15, 25, 27 sq., 31 sq., 35 sq., 39 sq., 50 sq.).

[KOS15] M. KELLER, E. ORSINI, P. SCHOLL. **"Actively secure OT extension with optimal overhead"**. In: *Annual Cryptology Conference*. Springer. 2015, pp. 724–741 (cit. on pp. 9, 28, 31, 36, 39 sq., 47 sq.).

[KS08] V. KOLESNIKOV, T. SCHNEIDER. **"Improved garbled circuit: Free XOR gates and applications"**. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 2008, pp. 486–498 (cit. on p. 12).

[Lar18] LARGE-SCALE DATA & SYSTEMS (LSDS) GROUP. *Spectre attack against SGX enclave*. 2018. URL: `https://github.com/lsds/spectre-attack-sgx` (visited on 07/22/2018) (cit. on p. 4).

[LP09] Y. LINDELL, B. PINKAS. **"A proof of security of Yao's protocol for two-party computation"**. In: *Journal of Cryptology* 22.2 (2009), pp. 161–188 (cit. on p. 36).

[NPS99] M. NAOR, B. PINKAS, R. SUMNER. **"Privacy preserving auctions and mechanism design"**. In: *Proceedings of the 1st ACM conference on Electronic commerce*. ACM. 1999, pp. 129–139 (cit. on p. 12).

[NR04] M. NAOR, O. REINGOLD. **"Number-theoretic constructions of efficient pseudo-random functions"**. In: *Journal of the ACM (JACM)* 51.2 (2004), pp. 231–262 (cit. on p. 28).

[OOS17] M. ORRÙ, E. ORSINI, P. SCHOLL. **"Actively secure 1-out-of-n OT extension with application to private set intersection"**. In: *Cryptographers' Track at the RSA Conference*. Springer. 2017, pp. 381–396 (cit. on p. 9).

[Ope] OPEN WHISPER SYSTEMS. *Signal privacy policy*. URL: `https://signal.org/signal/privacy/` (visited on 05/03/2018) (cit. on p. 21).

[Ope14a] OPEN WHISPER SYSTEMS. *Signal API Protocol*. 2014. URL: `https://github.com/signalapp/Signal-Server/wiki/API-Protocol` (visited on 05/13/2018) (cit. on pp. 21, 45).

[Ope14b] OPEN WHISPER SYSTEMS. *The Difficulty Of Private Contact Discovery*. 2014. URL: `https://signal.org/blog/contact-discovery/` (visited on 05/13/2018) (cit. on p. 19).

[PR04] R. PAGH, F. F. RODLER. **"Cuckoo hashing"**. In: *Journal of Algorithms* 51.2 (2004), pp. 122–144 (cit. on p. 13).

[PSSW09] B. PINKAS, T. SCHNEIDER, N. P. SMART, S. C. WILLIAMS. **"Secure two-party computation is practical"**. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2009, pp. 250–267 (cit. on p. 31).

[PSSZ15] B. PINKAS, T. SCHNEIDER, G. SEGEV, M. ZOHNER. **"Phasing: Private Set Intersection Using Permutation-based Hashing."** In: *USENIX Security Symposium*. Vol. 15. 2015, pp. 515–530 (cit. on p. 2).

[PSZ14] B. PINKAS, T. SCHNEIDER, M. ZOHNER. **"Faster Private Set Intersection Based on OT Extension."** In: *USENIX Security Symposium*. Vol. 14. 2014, pp. 797–812.

[PSZ18] B. PINKAS, T. SCHNEIDER, M. ZOHNER. **"Scalable private set intersection based on OT extension"**. In: *ACM Transactions on Privacy and Security (TOPS)* 21.2 (2018), p. 7 (cit. on pp. 2, 4).

[RA18] A. C. D. RESENDE, D. F. ARANHA. **"Faster Unbalanced Private Set Intersection"**. In: *Journal of Internet Services and Applications* 9.1 (2018), pp. 1–18 (cit. on p. 4).

[Rab81] M. O. RABIN. **"How to Exchange Secrets with Oblivious Transfer"**. In: (1981) (cit. on p. 5).

[Rad16] RADICAL APP, LLC. *Dust privacy policy*. 2016. URL: `https://usedust.com/privacy-policy` (visited on 05/03/2018) (cit. on p. 20).

[Row14] S. ROWLANDS. **"Mobile messaging: War of the words"**. In: *White Paper* (2014) (cit. on p. 1).

[SHS+15] E. M. SONGHORI, S. U. HUSSAIN, A.-R. SADEGHI, T. SCHNEIDER, F. KOUSHANFAR. **"Tinygarble: Highly compressed and scalable sequential garbled circuits"**. In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 411–428 (cit. on p. 12).

[SWG+17] M. SCHWARZ, S. WEISER, D. GRUSS, C. MAURICE, S. MANGARD. **"Malware guard extension: Using SGX to conceal cache attacks"**. In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2017, pp. 3–24 (cit. on p. 4).

[SZ13] T. SCHNEIDER, M. ZOHNER. **"GMW vs. Yao? Efficient secure two-party computation with low depth circuits"**. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2013, pp. 275–292 (cit. on p. 10).

[Tel] TELEGRAM MESSENGER LLP. *Telegram privacy policy*. URL: `https://telegram.org/privacy` (visited on 05/03/2018) (cit. on p. 22).

[Thr18a] THREEMA GMBH. *Threema Cryptography Whitepaper*. 2018. URL: https://threema.ch/press-files/2_documentation/cryptography_whitepaper.pdf (visited on 06/25/2018) (cit. on p. 22).

[Thr18b] THREEMA GMBH. *Threema privacy policy*. 2018. URL: https://threema.ch/en/privacy (visited on 05/03/2018) (cit. on p. 22).

[TLP+17] S. TAMRAKAR, J. LIU, A. PAVERD, J.-E. EKBERG, B. PINKAS, N. ASOKAN. **"The circle game: Scalable private membership test using trusted hardware"**. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM. 2017, pp. 31–44 (cit. on pp. 2, 4, 19).

[Vib17] VIBER MEDIA S.À R.L. *Viber privacy policy*. 2017. URL: https://www.viber.com/terms/viber-privacy-policy/ (visited on 05/03/2018) (cit. on p. 22).

[Wha17] WHATSAPP INC. *WhatsApp Security Whitepaper*. 2017. URL: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf (visited on 05/06/2018) (cit. on p. 22).

[Wha18] WHATSAPP INC. *WhatsApp privacy policy*. 2018. URL: https://www.whatsapp.com/legal/?l=en#privacy-policy (visited on 05/03/2018) (cit. on p. 22).

[Wic17] WICKR INC. *Wickr privacy policy*. 2017. URL: https://www.wickr.com/privacy-policy (visited on 05/03/2018) (cit. on p. 22).

[Wir17] WIRE SWISS GMBH. *Wire Security Whitepaper*. 2017. URL: https://wire-docs.wire.com/download/Wire+Privacy+Whitepaper.pdf (visited on 05/03/2018) (cit. on p. 23).

[Yao86] A. C.-C. YAO. **"How to generate and exchange secrets"**. In: *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE. 1986, pp. 162–167 (cit. on p. 10).

[ZRE15] S. ZAHUR, M. ROSULEK, D. EVANS. **"Two halves make a whole"**. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2015, pp. 220–250 (cit. on p. 12).