D14.3
Protocol Implementations

Project number:	609611
Project acronym:	**PRACTICE**
Project title:	Privacy-Preserving Computation in the Cloud
Project Start Date:	1st November, 2013
Duration:	36 months
Programme:	FP7/2007-2013
Deliverable Type:	Report
Reference Number:	ICT-609611 / D14.3 / 1.0
Activity and WP:	Activity 1 / WP14
Due Date:	October 2016 - M36
Actual Submission Date:	3rd November, 2016
Responsible Organisation:	ALX
Editor:	Peter Sebastian Nordholt
Dissemination Level:	Public
Revision:	1.0
Abstract:	This report documents implementations of secure computation protocols derived from theoretical work of work package 13. Together, this report and the implementations it documents constitute the deliverable D14.3.
Keywords:	Protocol, Protocol Suite, Secure Computation

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no. 609611.

Editor

Peter Sebastian Nordholt (ALX)

Contributors (ordered according to beneficiary numbers)

Daniel Demmler (TUDA)
Ágnes Kiss (TUDA)
Thomas Schneider (TUDA)
Michael Zohner (TUDA)
Jonas Lindstrøm (ALX)
Sander Siim (CYBER)
Manuel Barbosa (INESC PORTO)
Vitor Pereira (INESC PORTO)

Executive Summary

This report documents implementations of secure computation protocols derived from theoretical work of work package 13. Together, this report and the implementations it documents constitute the deliverable D14.3.

We consider four different protocol implementations, which are embedded in application oriented secure computation frameworks, and are more or less ready to be used in secure computation applications. First we describe a protocol for generating so-called Beaver Triples which is an important building block in many secure computation protocols, including the TinyTables protocol and the ABY framework which are both described in this report. The implementation is embedded in the Sharemind framework. Then we describe an implementation of a novel two-party secure computation protocol called the TinyTables protocol, which is based on the FRESCO framework. The third protocol we describe is the mixed protocol of the ABY secure computation framework, which allows one to switch between different secure computations protocols in order to optimize performance. The last protocol we describe is a formally verified implementation of Yao's garbled circuits protocol, including the effort to create a proof of the security and correctness of the protocol. The implementation is embedded in the FRESCO framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This deliverable is the accompanying report for chosen implementations of secure computation protocols derived from theoretical work of workpackage 13. We note that there is some overlap between this deliverable and deliverable 13.4 that also reports on such implementations. However, in this deliverable we focus on implementations that are fitted inside larger secure computation frameworks. Thus, while deliverable 13.4 focuses on proofs-of-concepts, this deliverable focuses on protocol implementations which are embedded in application oriented secure computation frameworks. Thus the protocols described here are in some sense *ready* to be utilized in building applications based on secure computation.

We describe four different protocol implementations in this report. The first three implementations represent different levels the PRACTICE project has been dealing with secure computation. First, at the low level we present an implementation of an improved sub-protocol used as a basic building block in many secure computation protocols. Second, at the middle level we present an implementation of a secure computation protocol following a new approach to secure computation in the two party setting. Third, at the high level we present an implementation of a system that takes several secure computation protocols and combines them in to a new mixed protocol benefiting from the distinct strengths of each of the underlying protocols. The fourth implementation described in this report represent a relatively new direction for secure computation. Namely, an implementation of a well known secure computation protocol that has been formally verified to follow the theoretical specification of the protocol.

In more detail we start in chapter 2 by describing an optimized protocol for generating so called Beaver triples. These are widely used in SMC protocols based on secret sharing, and is hence an important building block in many SMC protocols, including the ABY framework discussed in chapter 4 and the TinyTable protocol discussed in chapter 3. The concrete implementation described in this chapter is embedded in the Sharemind framework, and has been used in their prototypes described in deliverable D23.3.

In chapter 3 we describe an implementation of the TinyTable protocol, which is a novel two-party SMC protocol for computing boolean circuits developed by Damgård *et al.* at Aarhus University. The protocol is based on secret sharing, and we present an implementation based on the FRESCO framework described in deliverable D14.2.

Chapter 4 discusses the mixed protocol of the ABY secure computation framework. In particular how it allows one to switch between different protocols, based on boolean or arithmetic sharing as well as Yao's garbled circuits during the evaluation of a circuit in order to optimize the performance.

In chapter 5 we describe a formally verified implementation of Yao's garbled circuits protocol. In particular, the effort to obtain a mechanised proof of the security and correctness of the

protocol is presented. The resulting verified Yao implementation has been integrated in the FRESCO framework as described in deliverable 14.4.

Chapter 2

Efficient Beaver Triple Generation with Oblivious Transfer Extensions

In this chapter, we describe an optimized protocol for generating Beaver triples that are commonly used in two-party secure computation protocols based on secret sharing. Secret sharing based protocols provide a useful alternative to the Yao's garbled circuits approach to two-party computation [31] as they are more communication-efficient, with the drawback of non-constant round complexity. However, there are many practical examples of applications where the algorithms can be efficiently parallelized, greatly reducing the performance impact caused by the larger round complexity of the protocols and resulting in very fast implementations[1].

Two-party computation based on additive and bitwise secret sharing is supported in secure computation frameworks such as ABY (see Chapter 4) and Sharemind [9, 36][2]. In both of these frameworks, performing arithmetic on additively shared values relies on precomputed Beaver triples. The performance bottleneck for these protocols is in the offline precomputation phase, as the online phase is roughly an order of magnitude faster [21, 36]. Therefore, optimizing the Beaver triple generation protocols is most important for improving the efficiency of secret sharing based protocols in these frameworks.

2.1 Protocol Description

Our protocol is similar to the state-of-the-art passively secure Beaver triple generation protocol used in ABY [21]. The protocol in [21] relies on the 1-out-of-2 oblivious transfer extension protocol of Asharov et al. (ALSZ13 [1]). In our protocol, we instead employ the 1-out-of-N oblivious transfer extension of Kolesnikov and Kumaresan (KK13 [29]) to reduce the total communication cost [36].

In addition, we can show that our protocol is secure in the Universal Composability (UC) framework [13] without relying on the random oracle model. The security of our protocol is based on the notion of *correlation robustness* for hash functions [27], which is the underlying security assumption for both the ALSZ13 and KK13 oblivious transfer extension protocols [1, 29]. For details on the security proof, we refer the reader to [36].

Throughout this chapter, we describe protocols executed between two parties $\mathcal{P}_1$ and $\mathcal{P}_2$.

[1]For example, the privacy-preserving tax fraud detection [10, 11] and genome similarity computation [19] applications on Sharemind both employ heavily parallelized algorithms, that result in practically viable implementations.

[2]The online two-party protocols used in Sharemind are described also in PRACTICE deliverable D13.1, Section 3.4 [28].

We denote an additively shared value x as $[\![x]\!]$ and refer to the share of party $\mathcal{P}_i$ as $[\![x]\!]_i$. Additive sharing is performed over a ring $\mathbb{Z}_{2^k}$ for some $k \in \mathbb{N}$, such that $[\![x]\!]_1 + [\![x]\!]_2 \equiv x \mod 2^k$ and the shares $[\![x]\!]_1$, $[\![x]\!]_2$ are uniformly random and independent. We implicitly use modular arithmetic whenever dealing with elements of $\mathbb{Z}_{2^k}$.

We present the overall triple generation protocol as Alg. 1.

Algorithm 1 Computation of ℓ-bit multiplication triples

Input: No input
Output: Beaver triple $[\![a]\!] \cdot [\![b]\!] = [\![c]\!]$

1: $\mathcal{P}_i$ generate uniformly random values $[\![a']\!]_i \leftarrow \mathbb{Z}_{2^\ell}$, $[\![b']\!]_i \leftarrow \mathbb{Z}_{2^\ell}$
2: The parties compute $[\![u]\!] = [\![a']\!]_1 \cdot [\![b']\!]_2$ using Alg. 2
3: The parties compute $[\![v]\!] = [\![a']\!]_2 \cdot [\![b']\!]_1$ using Alg. 2
4: $\mathcal{P}_i$ fixes $[\![c']\!]_i = [\![a']\!]_i \cdot [\![b']\!]_i + [\![u]\!]_i + [\![v]\!]_i$
5: $[\![a]\!] \leftarrow \mathsf{reshare}([\![a']\!])$, $[\![b]\!] \leftarrow \mathsf{reshare}([\![b']\!])$ and $[\![c]\!] \leftarrow \mathsf{reshare}([\![c']\!])$
6: **return** $[\![a]\!], [\![b]\!], [\![c]\!]$

Here, $\mathsf{reshare}()$ denotes a secure resharing protocol, which produces fresh random shares for a secret-shared value, without changing the actual value. The resharing step is required to show UC security of our protocol [36]. The crux of the protocol is multiplying the shares of $[\![a']\!]$ and $[\![b']\!]$. For this, we use a modified version of Gilboa's protocol for multiplying secret inputs from different parties [23] that uses 1-out-of-N oblivious transfer (denoted by $\binom{N}{1}$-OT). For $x \in \mathbb{Z}_{2^k}$, we denote by $x[i]$ the ith bit of x, where $1 \leq i \leq k$. The protocol is presented as Alg. 2.

Algorithm 2 Protocol for multiplying ℓ-bit integers held by different parties using $\binom{N}{1}$-OT

Setup: Values $L_1, \ldots, L_k$, such that $L_i \geq 1$ and $\sum_{i=1}^{k} L_i = \ell$
Input: $\mathcal{P}_1$ inputs $x \in \mathbb{Z}_{2^\ell}$, $\mathcal{P}_2$ inputs $y \in \mathbb{Z}_{2^\ell}$
Output: Additively shared result $[\![z]\!]$, where $z = xy$

1: $t = 0$
2: **for** $i \in \{1, \ldots, k\}$ **do** ▷ Perform $\binom{2^{L_i}}{1}$-OT for multiplying next L_i bits
3: $\mathcal{P}_1$ computes $x' = x \cdot 2^t$ and generates random $r_i \leftarrow \mathbb{Z}_{2^\ell}$
4: $\mathcal{P}_1$ fixes OT messages $m_i^j = r_i + x' \cdot (j-1)$ for $j \in 1, \ldots, 2^{L_i}$
5: $\mathcal{P}_2$ fixes choice index $a_i = \left(\sum_{j=1}^{L_i} 2^{j-1} \cdot y[j+t]\right) + 1$
6: Parties run $\binom{2^{L_i}}{1}$-OT with messages m_i^j from $\mathcal{P}_1$ and choice index a_i from $\mathcal{P}_2$. $\mathcal{P}_2$ receives $s_i = m_i^{a_i}$.
7: $t = t + L_i$
8: **end for**
9: $\mathcal{P}_1$ fixes $[\![z]\!]_1 = -\sum_{i=1}^{k} r_i$
10: $\mathcal{P}_2$ fixes $[\![z]\!]_2 = \sum_{i=1}^{k} s_i$
11: **return** $[\![z]\!]$

Note that for $L_1 = \ldots = L_\ell = 1$, we get the original protocol from [23] using $\binom{2}{1}$-OT. In the original protocol, x and y are multiplied bit-by-bit. Our protocol is a generalized version that multiplies chunks of consecutive L_i bits of the inputs using a single $\binom{N}{1}$-OT, as opposed to performing $\binom{2}{1}$-OT L_i times. Our insight is that this way, the total communication of the protocol can be reduced when using appropriate values of L_i.

Similarly to the original protocol, instead of using a standard $\binom{N}{1}$-OT functionality, we can use a more efficient *correlated oblivious transfer* in Alg. 2. In correlated $\binom{N}{1}$-OT, the sender does not fix the first message m_1 as input to the protocol, but instead, a random m_1 is generated as part of the protocol. The sender inputs correlation functions $f_2, \ldots, f_N$, which are used in the protocol to fix the other $N-1$ messages to $m_i = f_i(m_1)$ for $i \in \{2, \ldots, N\}$. It is straightforward to apply correlated OT to Alg. 2, since the first message is randomly generated by $\mathcal{P}_1$.

Another optimization that is mentioned also in [21], is that as the value of x' in the protocol is shifted left bitwise, only the uppermost bits need to be obliviously transferred, since the lower bits are zeroed out. This means that the message size for each successive OT is effectively smaller, the i-th OT then has message size $\ell - \sum_{j=1}^{i-1} L_j$. Naturally, all the transfers can be performed in a single round in parallel.

We instantiate the correlated $\binom{N}{1}$-OT using the KK13 oblivious transfer extension protocol [29], since it is currently the most efficient known protocol for passive security. We have calculated the optimal L_i values to minimize total communication of the triple generation for security parameter $\kappa_0 = 128$ (Table 2.2). Note that the equivalent KK13 protocol security parameter is larger, depending on the number of messages in the OT [36]. The corresponding security parameters are presented in Table 2.1.

Table 2.1: KK13 $\binom{N}{1}$-OT security parameters for equivalent security of ALSZ13 protocol.

ALSZ13 security parameter κ_0	**Number of messages N**	**Equivalent KK13 security parameter κ**
128	2	128
128	4	192
128	8	224
128	16	240

Table 2.2: Optimal L_i values for Alg. 2 minimizing total communication of Alg. 1.

Triple length ℓ	**L_i values**	**Required KK13 security parameter κ**	**Communication of Alg. 1 (bits)**
8	(4,4)	240	1320
16	(4,4,4,4)	240	3120
32	(2,3,3,3,3,3,3,3,3,3,3)	224	7430
64	(2, ..., 2)	192	18624

Note that we assume the highest security parameter κ required by the L_i values is used for all oblivious transfers. Theoretically, different κ values can be used, which would result in a different set of optimal L_i values and further reduced communication cost, but this makes an efficient implementation more difficult. We also implicitly assume that optimal message sizes are used for each OT iteration as explained above.

In summary, Table 2.3 compares the communication cost of our approach with the previous state-of-the-art protocol using ALSZ13 $\binom{2}{1}$-OT extension [21]. The communication cost reflects the total bi-directional communication in Alg. 1 for security parameter $\kappa_0 = 128$. Note that since the roles of $\mathcal{P}_1$ and $\mathcal{P}_2$ can be trivially switched in Alg. 1, we are interested in minimizing

the protocol's total communication, although communication is asymmetric for both ALSZ13 and KK13 OT protocols.

Table 2.3: Communication in bits for single ℓ-bit triple computation with different methods.

Triple length ℓ	**ALSZ13 1-out-of-2 OT**	**KK13 1-out-of-2^{L_i} OT**	**Communication reduction**
8	2120	1320	**37.7%**
16	4368	3120	**28.6%**
32	9248	7430	**19.7%**
64	20544	18624	**9.3%**

2.2 Implementation

We have implemented the triple generation protocol as a precomputation step to Sharemind's two-party protocol suite in C++. For benchmarking purposes, we implemented both the standard ALSZ13 $\binom{2}{1}$-OT extension based protocol (the protocol used in ABY [21]) as well as the KK13 $\binom{N}{1}$-OT extension protocol in Alg. 1.

The triple precomputation protocol is executed on a separate thread from the main thread of Sharemind's runtime, that runs privacy-preserving programs and the required online protocols. The precomputation thread computes multiplication triples in fixed-size batches and stores them in a memory buffer with fixed size. The batch size and buffer size for triple generation is read from a configuration file when starting the Sharemind servers.

Access to the buffers is synchronized with the thread running the online protocols, that removes triple elements from the buffers as they are needed in computations. The precomputation thread independently computes a new batch of triples whenever the buffers are depleted enough to fit a single new batch of triples.

2.2.1 Pseudo-random generator

To instantiate the pseudo-random generator (PRG) required in the oblivious transfer extension protocols, we use the AES-128 block cipher in CTR mode, which gives us 128-bit security. In the case where the OT extension parameter $\kappa = 128$, we seed only the AES key with 128 bits, and take the IV (initialization vector) as 0. For larger κ values (in the KK13 protocol), we use the extra bits to also seed the IV. This gives a unique PRG for each seed (supporting up to 256 bit seeds), while still retaining at least 128-bit security for the PRG, according to the NIST recommendations [2].

Using AES allows us to leverage the Intel AES-NI instruction set for much better performance than a software implementation of AES[3]. Especially, we can process 8 blocks of output in parallel in CTR mode. We do not need to use AES-NI intrinsics explicitly in our implementation code, as we use the OpenSSL implementation of AES, which automatically uses AES-NI instructions by default if they are supported by the hardware. Also, parallel encryption/decryption is handled by the OpenSSL implementation in block cipher modes that allow it. Overall,

[3]Intel's hardware accelerated AES implementation `https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni`

the PRG computations are not a bottleneck in our protocols, even though we need to generate pseudorandomness from κ different PRG-s at the same time.

2.2.2 Hash function

We instantiate the correlation-robust hash function with either an AES block cipher construction or SHA256 hash function. For the ALSZ13 OT extension protocol (with $\kappa = 128$), we can use the fixed-key AES construction used in [21]:

$$H(x,t) = AES_K(x \oplus t) \oplus x \oplus t$$

for public K, input x and monotonically increasing nonce t. However, for the KK13 protocol (with $\kappa > 128$), the same construction cannot be used, since the size of the hash input is κ bits in the protocol, but the block size of AES is 128 bits for all key sizes. For KK13 protocol, we thus use SHA256 directly as an instantiation. The AES construction is much faster due to AES-NI instructions and as such, gives the ALSZ13 protocol an advantage in performance of local computations.

Calculating the hash function was the clear bottleneck in our LAN (local area network) setting benchmarks with fast network links, taking up to 80% of the total running time. Due to this, we use multiple parallel threads for calculating the hashes and we performed benchmarks with different numbers of hashing threads. A SHA-256 implementation, which could leverage hardware SIMD (*single-instruction multiple-data*) instructions for calculating many hashes in parallel on a single thread would be very beneficial for increased performance. Currently, we use the OpenSSL implementation of SHA-256, as local benchmarks showed it is more efficient then the implementation of CryptoPP library.

We also briefly considered and tested other instantiations, in particular SHA-3 and an improved version of one of the SHA-3 finalists, BLAKE2. For SHA-3, the only C++ implementation we found was from the CryptoPP library. Initial benchmarks showed that it was ~2 times slower than SHA-256, and so we currently abandoned it as an instantiation candidate.

For BLAKE2, we tested the official implementation[4]. Specifically, we used the SIMD-optimized Blake2b variant with 32-byte outputs. Although the BLAKE2 official web-site advertises very high performance, we only observed relatively little performance gains in our protocols compared to OpenSSL's SHA-256. We suspect that the function and perhaps the implementation also is fine-tuned for computing a single hash from very large input data, but not for our use case of computing a huge amount of hashes on relatively small inputs.

2.2.3 Bit-level operations

Our bit matrix transposition uses a sequential algorithm, which employs Intel's SIMD AVX2 instructions. AVX2 instructions allow to operate directly with 256-bit registers. In our case, we use these operations for bitwise XOR and bitwise AND and a few other specific operations. With AVX2 instructions, we can perform these bitwise operations on 256 bit inputs in roughly the same amount of processor cycles as performing a single 64-bit bitwise operation.

Our bit matrix transposition implementation is based on the code from[5], modified to use AVX2 instructions, as the original code uses only SSE2 instructions with access to 128-bit registers. Bit matrix transposition is required in all OT extension protocols we have considered and is a rather costly computational task, as already noted in [1].

[4] `https://github.com/BLAKE2/BLAKE2`

[5] `https://mischasan.wordpress.com/2011/10/03/the-full-sse2-bit-matrix-transpose-routine/`

Local tests show that Eklundh's algorithm [22] used in ABY (and originally proposed in [1]) performs slightly better than AVX2-based sequential algorithm. Even more gains might be possible with a implementation of Eklundh's algorithm that leverages the SIMD-instructions. We currently have not implemented Eklundh's algorithm ourselves due to time constraints but this would help optimize our implementation.

2.2.4 Batching

Our current implementation generates triples in large batches. After some initial testing, we chose to generate 100 000 triples in a single batch, so that the time it takes to complete one batch is reasonably small. To generate a million triples, we calculate ten of these batches sequentially and so on. Note that for generating 100 000 triples, the corresponding batch size for OT is much higher. E.g. for the baseline ALSZ13 protocol and 64-bit triples, we have to perform 12 800 000 OT-s in total using the OT extension.

We observed that performing triple generation in smaller batches sequentially leads to longer total running time. However, doing the computation in one large batch means that there is a significant overhead introduced by one party having to wait for a message from the other party while it finishes its computations. This overhead is introduced by data dependencies of the network message on local computation results. We attempted to reduce the overhead of local computation by using parallel threads for hashing, which is the most intensive computation done in the protocols. However, using the PRG-s and transposing bit matrices also takes noticeable time on very large matrices.

In hindsight, a more efficient batching strategy would have been to run smaller batches independently and in parallel on the network. This means local computations and network communication would be naturally more interleaved, although some overhead is introduced for multiplexing the messages from different protocol instances. Our current implementation does not support this, but this is a useful optimization strategy for the future. We also believe it would give the KK13-based protocol more of an advantage over ALSZ13, even when using a slower hash function.

2.3 Performance

The benchmarks were performed on a cluster of two machines, with a dedicated fast 10 Gbit/s network link, 128 GB of RAM and two Intel Xeon E5-2640 v3 2,6 GHz/8GT/20M processors, meaning a total of 16 cores and 32 parallel threads with Intel HyperThreading.

We performed benchmarks in both a LAN and WAN (*wide-area network*) setting. The LAN setting means that network performance of the communication channel is very high, especially, latency is very low. The WAN setting simulates low performance network conditions, or when the computing parties are located very far from each other geographically. We simulate the WAN setting in our local cluster by using the Linux command line tool `tc` (traffic control).

- LAN — $<$ 0.1ms latency and up to 10Gbit/s bandwidth
- WAN — 170 ms latency (round-trip time) and throttled bandwidth at 70 Mbit/s (peak rate at 100 Mbit/s)

In Table 2.4, we report the time for computing a total of 100 000 triples in a single batch. The average time of ten iterations of these batches is shown for each experiment in seconds. We performed tests in both LAN and WAN settings and using either 4 or 16 parallel hashing

threads. We did not observe any significant performance gains when using more than 16 threads for batch size of 100 000 triples.

We benchmarked three different methods. First, we measured triple generation using the standard 1-out-of-2 OT extension of ALSZ13, which is the method used in [21]. We tested the ALSZ13 version with both the fast fixed-key AES hash construction and SHA256, to measure the effect of the hash function on overall performance. Finally, we benchmarked our approach of using KK13 1-out-of-N OT extension protocol using SHA256 as the hash function.

Table 2.4: Total running times in seconds for the triple generation to compute 100 000 triples with different OT extension methods. The fastest time for each setting is highlighted in bold.

Threads	**Network**	**Triple bitlength**	**ALSZ13 SHA256**	**ALSZ13 AES**	**KK13 $\binom{2^{L_i}}{1}$ SHA256**
4	LAN	8	0.91	**0.53**	1.13
4	LAN	16	1.62	**0.96**	2.15
4	LAN	32	3.09	**1.71**	3.05
4	LAN	64	5.49	**3.25**	5.31
4	WAN	8	4.39	3.90	**3.84**
4	WAN	16	8.58	**7.61**	8.39
4	WAN	32	18.10	**15.45**	16.55
4	WAN	64	37.93	**33.89**	38.26
16	LAN	8	0.60	**0.48**	0.57
16	LAN	16	1.10	**0.86**	1.01
16	LAN	32	1.93	**1.60**	1.64
16	LAN	64	3.51	**3.09**	3.24
16	WAN	8	3.99	3.92	**3.01**
16	WAN	16	7.75	7.53	**6.32**
16	WAN	32	15.95	15.52	**14.11**
16	WAN	64	34.86	**34.07**	34.94

The results show, that KK13 based protocol performs better in WAN setting, due to the decreased communication cost. However, in our implementation, the local computations proved to be a larger bottleneck than the network overhead, which suggests it could still be heavily optimized. Especially, our ALSZ13 implementation with fixed-key AES does not achieve the same performance as the implementation in ABY [21]. We believe the main issue in our implementation is a suboptimal batching strategy. Since we compute large batches sequentially, there is a significant overhead caused by local computations and network communication not being interleaved. For a more streamlined approach, we should compute many smaller batches in parallel.

Another improvement would be to use a faster hash function, as SHA256 does not live up to state-of-the-art performance standards, even when multiple parallel threads for hashing are used. This is demonstrated by the benchmarks of LAN setting with 16 threads, where the standard ALSZ13 with fixed-key AES still outperforms the KK13 protocol. However, when comparing the KK13 protocol against ALSZ13 with SHA256, we see the KK13 protocol is faster, showing the potential of our approach.

One option is to make a construction based on fixed-key AES that can handle more than 128 bits of input. However, since fixed-key AES constructions are secure only in the (optimistic) ideal cipher model, another option is to use AES as a pseudorandom function, similarly to the approach of [24]. The authors show that it is also possible to streamline the costly AES key scheduling operation with AES-NI instructions, which means this approach can be competitive performance-wise with fixed-key AES. This would allow AES256 to be used for 256-bit inputs.

Chapter 3

Tiny Tables

3.1 Protocol Description

The TinyTables protocol is a two-party protocol for securely computing a boolean circuit. The protocol was developed by Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen and Samuel Ranellucci [18]. We have implemented the semi-honest version of this protocol, although a version with malicious security is also described in the paper.

Computing a boolean circuit C consisting of boolean gates $G_1, G_2, \ldots, G_N$ and wires w_1, $w_2, \ldots, w_M$ using the TinyTables protocol is done in two phases: a *preprocessing* phase and an *online* phase. The idea is that the actual value $b_u \in \mathbb{Z}_2$ on a wire w_u in the circuit is encrypted, such that both players instead know an encrypted value $e_u = b_u \oplus r_u$ where $r_u \in \mathbb{Z}_2$ is an additively shared masking parameter where player i knows r_u^i for $i = 1, 2$ and $r_u = r_u^1 \oplus r_u^2$.

Now, when evaluating a gate G with input wires $w_{u_1}, \ldots, w_{u_n}$ and output wire w_o which is a linear function, e.g.

$$G(b_{u_1}, \ldots, b_{u_n}) = a_1 b_{u_1} \oplus \cdots \oplus a_n b_{u_n} \oplus c$$

for some $a_1, \ldots, a_n, c \in \mathbb{Z}_2$, in the preprocessing phase, player i defines his mask of the output wire r_o^i as

$$r_o^i = a_1 r_{u_1}^i \oplus \cdots \oplus a_n r_{u_n}^i,$$

and in the online phase the players let

$$e_o = a_1 e_{u_1} \oplus \cdots \oplus a_n e_{u_n} \oplus c.$$

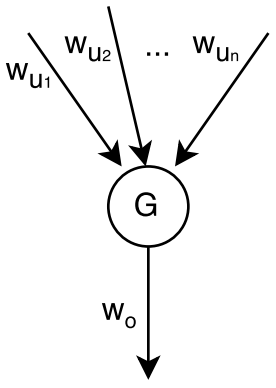

Figure 3.1: A gate G with n input wires $w_{u_1}, \ldots, w_{u_n}$ and one output wire w_o.

For a non-linear gate G with input wires $w_{u_1}, \ldots, w_{u_n}$ and ouptut wire w_o, both players need to calculate an n-dimensional table each during preprocessing, a so-called *TinyTable*, which for player i is denoted $T^i(G)$ and is indexed by elements in $\mathbb{Z}_2^n$. This table is created such that for encrypted input values $(e_{u_1}, \ldots, e_{u_n}) \in \mathbb{Z}_2^n$, the two table entries $T^1(G)_{e_{u_1},\ldots,e_{u_n}}$ and $T^2(G)_{e_{u_1},\ldots,e_{u_n}}$ are an additive sharing of the encrypted output value

$$e_o = b_o \oplus r_o = G(b_{u_1}, \ldots, b_{u_n}) \oplus r_o.$$

Below we will give some more details on how the two phases are done for specific gates. We will omit the proof of the protocol being semi-honest, but it can be found in the paper.

3.1.1 Implemented gates

We have implemented five different gates in the circuit: XOR, NOT, AND, CLOSE, OPEN. Note that here only AND gates are non-linear so we only need to calculate TinyTables for these. Also, even though we allow for an arbitrary number of inputs, all the gates considered have either one or two inputs.

In both the preprocessing and the online phases, evaluation of gates are done in an order such that when a gate G is evaluated, the gates whose output wires are one of G's input wires has already been evaluated.

We recall that after the preprocessing and online phases of a gate G with input wires $w_{u_1}, \ldots, w_{u_n}$ and output wire w_o, both players should know the encrypted value

$$e_o = G(b_{u_1}, \ldots, b_{u_n}) \oplus r_o$$

where r_o is additively shared between the two players. We leave it to the reader to verify this for each gate below.

XOR An XOR gate G with two input wires w_u and w_v and output wire w_o is linear and is defined as $G(b_u, b_v) = b_u \oplus b_v$.

Preprocessing Player i let $r_o^i = r_u^i \oplus r_v^i$ for $i = 1, 2$.

Online Both players let $e_o = e_u \oplus e_v$.

NOT A NOT gate G with a single input wire w_u and an output wire w_o is linear and is defined as $G(b_u) = b_u \oplus 1$.

Preprocessing Player i let $r_o^i = r_u^i$ for $i = 1, 2$.

Online Both players let $e_o = e_u \oplus 1$.

AND An AND gate G with two input wires, w_u and w_v and an output wire w_o is non-linear and defined as $G(b_u, b_v) = b_u b_v$.

Preprocessing Player 1 picks r_o^1 and two masking parameters x_0 and x_1 at random. Now, player 1 acts as the sender in two *Oblivious Transfer* (OT) protocol instances, using $(x_o, x_o \oplus r_u^1)$ and $(x_1, x_1 \oplus r_v^1)$ respectively as inputs, where player 2 acts as the receiver and uses r_v^2 and r_u^2 respectively as selection bits. The outputs of the two OTs for player 2 are

$$y_0 = x_0 \oplus r_u^1 r_v^2$$
$$y_1 = x_1 \oplus r_v^1 r_u^2,$$

so now x_0 and y_0 is an additive secret sharing of $r_u^1 r_v^2$, and x_1 and y_1 is an additive secret sharing of $r_u^2 r_v^1$, so player 1 and 2 now have a secret sharing of $r_u r_v$.

Player 1 now defines his TinyTable $T^1(G)$ as

$$\begin{aligned} T^1(G)_{0,0} &= r_o^1 \oplus x_0 \oplus x_1 \oplus r_u^1 r_v^1, \\ T^1(G)_{0,1} &= T^1(G)_{0,0} \oplus r_u^1, \\ T^1(G)_{1,0} &= T^1(G)_{0,0} \oplus r_v^1, \\ T^1(G)_{1,1} &= T^1(G)_{0,0} \oplus r_u^1 r_v^1. \end{aligned}$$

Player 2 picks his share r_o^2 of the mask of the output wire at random and calculates his TinyTable $T^2(G)$ as follows:

$$\begin{aligned} T^2(G)_{0,0} &= y_0 \oplus y_1 \oplus r_u^2 r_v^2 \oplus r_o^2, \\ T^2(G)_{0,1} &= T^2(G)_{0,0} \oplus r_u^2, \\ T^2(G)_{1,0} &= T^2(G)_{0,0} \oplus r_v^2, \\ T^2(G)_{1,1} &= T^2(G)_{0,0} \oplus r_u^2 \oplus r_v^2 \oplus 1. \oplus 1. \end{aligned}$$

It is now straight-forward to verify that

$$T^2(G)_{c,d} = T^1(G)_{c,d} \oplus r_o \oplus G(r_u \oplus c, r_v \oplus d)$$

for all $(c, d) \in \mathbb{Z}_2^2$.

Online On input values e_u, e_v, player i looks up the corresponding value in his TinyTable, $T^i(G)_{e_u,e_v}$, and sends this value to the other player. Both players now let $e_o = T^1(G)_{e_u,e_v} \oplus T^2(G)_{e_u,e_v}$.

CLOSE A CLOSE gate has one input wire w_u which is unmasked, ie. $r_u = 0$, and one output wire w_o. One of the players i is the *inputter* (the player who will provide the input in the online phase).

Preprocessing The inputter, say player i, picks r_o^i at random, and the other player, say player j, picks $r_o^j = 0$.

Online The inputter, say player i, let $e_o = b_o \oplus r_o^i$ where b_o is his input value, and sends e_o to the other player.

OPEN An OPEN gate has one input wire w_u and one output wire w_o.

Preprocessing Player i let $r_o^i = r_u^i$ for $i = 1, 2$.

Online Both players send their share of r_o to the other player, and let $b_o = e_o \oplus r_o$.

3.2 Implementation

3.2.1 Architecture

We have implemented the TinyTable protocol in the FRESCO framework, whose architecture is described in detail in [20], and is available for download at `https://github.com/aicis/fresco/`. The preprocessing and online phases have been implemented as two seperate protocol suites in FRESCO, see figure 3.2 and also figure 3.1 in [20]. In order to evaluate a cirtcuit, it has

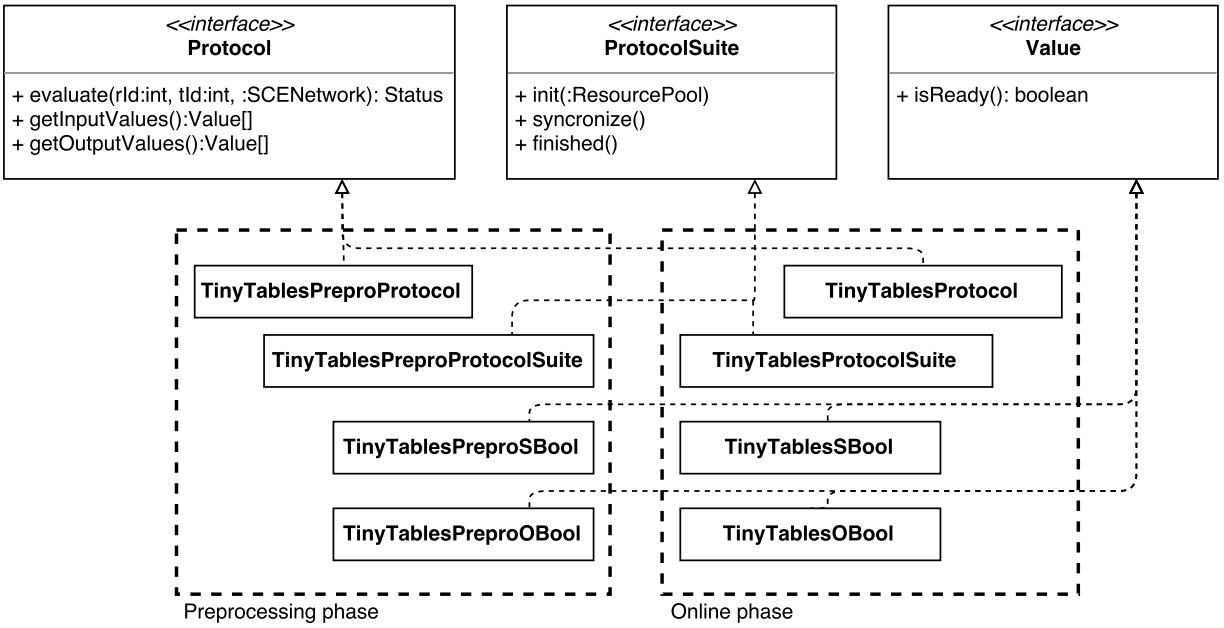

Figure 3.2: UML-diagram of how the the TinyTable protocol is implemented as an instance of a protocol suite in the FRESCO framewoek.

to be evaluated first using the `TinyTablesPreproProtocolSuite` which is an implementation of the preprocessing phase, and then `TinyTablesProtocolSuite` which is an implementation of the online phase. After the preprocessing, each player stores his generated TinyTables and the masks for his inputs to a file so they can be used in the online phase.

We have implemented CLOSE, OPEN, NOT, AND and XOR protocols for both phases. In the preprocessing phase the corresponding classes are named `TinyTablesPreproXProtocol` an in the online phase they are called `TinyTablesXProtocol`. These are created by two factory classes, `TinyTablesPreproFactory` and `TinyTablesFactory` respectively.

Encrypted wires in the preprocessing phase are represented by instances of the `TinyTables PreproSBool` class. Note that the wires in the preprocessing phase holds no values since values are not assigned to the wires until the online phase where wires are represented by `TinyTablesSBool`, but we let the `TinyTablesPreproSBool` hold the players share of the masking parameter, which in the preceeding section was denoted by r_w^i for players i's share of the mask of the wire w.

In order to be able to load the correct TinyTable for a gate in the online phase, an ID is assigned to each gate by the `TinyTablesPreproFactory` and `TinyTablesFactory` based on the order of the gates with the first being assigned an ID equal to 0, the next is assigned an ID equal 1, etc. To maintain consistency between the preprocessing and online phases, the circuit has to be constructed in the exact same order in the two phases.

The gates we have considered in this implementation either have one or two input wires. However, the protocol allows more complicated gates with an arbitrary number of inputs. This will for example allow us to implement an S-box, which is used in AES, as a gate with a corresponding TinyTable, such that evaluating an S-box will be reduced to a single look-up in the TinyTable. Note that the size of a TinyTable is 2^n bits where n is the number of input wires, so an S-Box, which has eight input wires, will use 32 bytes.

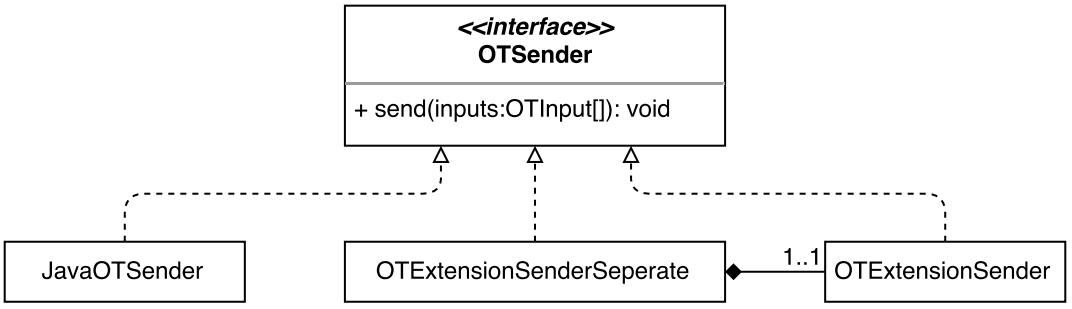

Figure 3.3: UML-diagram of the different implementations of `OTSender`.

3.2.2 Oblivious transfers implementations

We use OTs during the generation of the TinyTable for an AND-gate. We do this in the `finishedEval` method of the `TinyTablesPreproProtocol` class, which is called when the evaluation of the circuit is finished.

We use the SCAPI[1] library for performing the OTs. The SCAPI library features several implementations of OT, and we use two different semi-honest implementations: one implemented in Java, and a faster protocol, OT-Extension, which is implemented in C++ with a *Java Native Interface* (JNI) to make it usable for Java applications. In the implementation, we let both of these be realizations of the interfaces `OTSender` and `OTReceiver`. We also include a version of the `OTSender` and `-receiver` where the `OTExtensionSender` and `-receiver` are called from a seperate Java application. This is only used for test purposes where both have to be run in the same Java Virtual Machine. We have shown the realtionship between the different implementations of `OTSender` in figure 3.3, and the relations between the different implementations of `OTReceiver` are equivalent.

The SCAPI Java-library is already included in FRESCO, so the first version of OT is available already, but the OT-extension implemented with JNI requires the SCAPI library to be installed[2]. OT-extension is faster but is not always available, so in order to use the fastest solution the players negotiate before the preprocessing begins whether they both have the SCAPI library installed. If this is the case, the faster version is used and if not, they fall back to the slower version. The implementation of this is in the method `TinyTablesPreproProtocolSuite.negotiateOTExtension`.

To optimize performance, we do all OTs (two per AND-gate) in one batch in the `finishedEval`-method of the `TinyTablesPreproProtocolSuite` class.

Using precomputed multiplication triples

It is possible to compute a secret sharing of $r_u r_v$ using a precomputed additively shared multiplication triple, e.g. three secret shared values a, b, c such that $ab = c$, which can be computed using OTs *before* the preprocessing phase (see chapter 2 for an in depth discussion on how these triples can be generated efficiently). Now if the players open the secret shared values $d = r_u \oplus a$ and $e = r_v \oplus b$ so they are known by both players, they can locally compute their additive share of

$$de \oplus ea \oplus db \oplus c = r_u r_v.$$

[1] `https://scapi.readthedocs.io/en/latest/`

[2] See `http://scapi.readthedocs.io/en/latest/install.html` on how to install the SCAPI library.

Table 3.1: Timings for two players performing an instance of 128-bit AES using the TinyTables protocol implemented in the FRESCO framework measured as the average after 10 executions.

	Preprocessing				**Online**			
Evaluation strategy	Sequential		Parallel		Sequential		Parallel	
Player	1	2	1	2	1	2	1	2
Timing (ms)	522	537	538	557	1352	1458	1088	1087

However, since they have to open the values d and e, this will require two bits of additional network communication each way compared to the solution currently implemented where only the OTs are needed during preprocessing.

3.3 Performance

We have benchmarked our implementation of TinyTables with respect to timing and network traffic. Specifically, our measurements are done on an encryption of a 128-bit plaintext provided by player 1, using the 128-bit AES cipher and a key provided by player 2. After the evaluation of the protocol, both players know the cipher text but not the other player's input.

The platforms used for testing are two virtual machines running on the same host. Both virtual machines run Ubuntu Linux 16.04, and the host is an early 2013 Macbook Pro with 2,8 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 RAM, running Mac OSX 10.11.6.

Timings of how fast the preprocessing and online phases are completed are shown in table 3.1. The FRESCO framework allows several evaluation strategies of the gates in a protocol, and here we consider two strategies: sequential evaluation, where all gates are evaluated one after the other, and parallel evaluation where gates, if possible, are evaluated in parallel, allowing some network communication to be batched. Since we have been running the tests locally, running in parallel does not give much advantage. However, on a network with high latency, we believe it will give a significant advantage.

Timings are measured from the `init` method which is called on the used instance of `ProtocolSuite`, until the `finishedEval` method is called on the same instance.

The amount of network traffic is shown in table 3.2. We have specified both the actual amount of data transmitted, but also the theoretical amount of data needed to be transmitted by the protocol. Note that only the CLOSE, OPEN and especially AND gates require communication between the two parties, and in the specficitaion of the AES protocol we have used there are 256 CLOSE gates, 128 OPEN gates and 6,800 AND gates.

The difference between the theoretical and actual amount of transmitted data in the online phase is very large. This is because in the FRESCO framework, a class whose instances are to be transmitted over the network must implement the `Serializable`[3] interface. This ensures that instances of the class can be encoded as a `byte`-array and reconstructed again by the receiver. In the TinyTables protocol most of the objects transmitted are of the primitive type `boolean`, which must be wrapped as a `Boolean` object, which contrary to `boolean`s do implement the `Serializable` interface, in order to be transmitted. However, the serialization of a `Boolean` object as a `byte`-array results in an of length 71, giving a factor 568 overhead to the one bit of theoretical information transmitted. In future releases of the FRESCO framework, we plan to allow the transmission of simpler types, e.g. `byte`-arrays, which will make it possible to reduce

[3] `https://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html`

Table 3.2: The network traffic between the two parties in the preprocessing and online phases of a protocol which using 128-bit AES encrypts a clear text provided by player 1 using a key provided by player 2.

	Preprocessing		**Online**	
	Actual			
Direction	$1 \rightarrow 2$	$1 \leftarrow 2$	$1 \rightarrow 2$	$1 \leftarrow 2$
Amount (bytes)	13,600	108,800	500,976	500,976
	Theoretical			
Direction	$1 \rightarrow 2$	$1 \leftarrow 2$	$1 \rightarrow 2$	$1 \leftarrow 2$
Amount (bytes)	13,600	108,800	882	882

the overhead in this implementation by encoding `boolean`s as bytes, giving a factor 8 overhead, or by batching transmitted information as `Bitset`s[4].

All the network traffic in the preprocessing phase is due to the OTs which is done using native code from the SCAPI library, which does not give any overhead, so here there is no difference between the actual and theoretical amount of data transmitted.

[4] `https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html`

Chapter 4

Mixed-protocol implementation – ABY

In secure two-party or secure multy-party computation, the function is often to be expressed and evaluated as a Boolean or arithmetic circuit. As described in deliverable D13.1 [28], ABY allows for mixing the secure computation protocols that are used for the secure evaluation of the circuit.

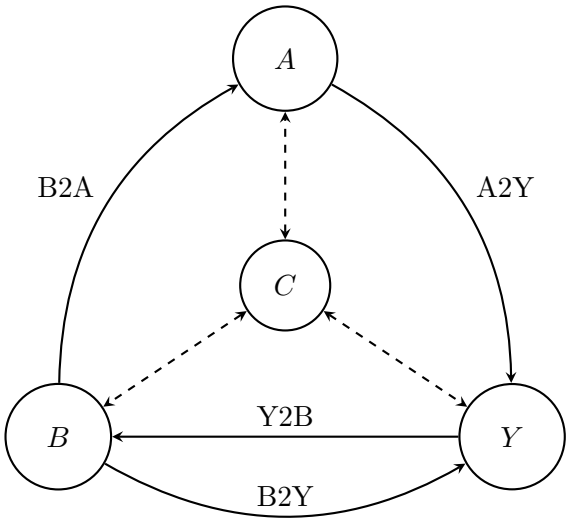

Figure 4.1: Overview of the ABY framework that allows efficient conversions between **C**leartexts and three types of sharings: **A**rithmetic, **B**oolean, and **Y**ao.

4.1 Protocol Description

Deliverable D13.1 [28] provides the detailed description on how mixing the secure computation protocols, i.e., switching between two protocols is achieved in the ABY secure computation framework. It efficiently combines arithmetic sharing, Boolean sharing with the GMW protocol and Yao's garbled circuits.

Protocols can be split in two phases: a setup phase that can take place at any time and an online phase that takes place as soon as the parties' inputs are known. The goal is thus to have a very fast online phase and shift most computation to the setup phase.

A crucial part of the performance of ABY is the precomputation of so called multiplication triples in the setup phase. These triples are then used for the computation of a multiplication in arithmetic sharing and an AND gate in Boolean sharing. Both operations still require

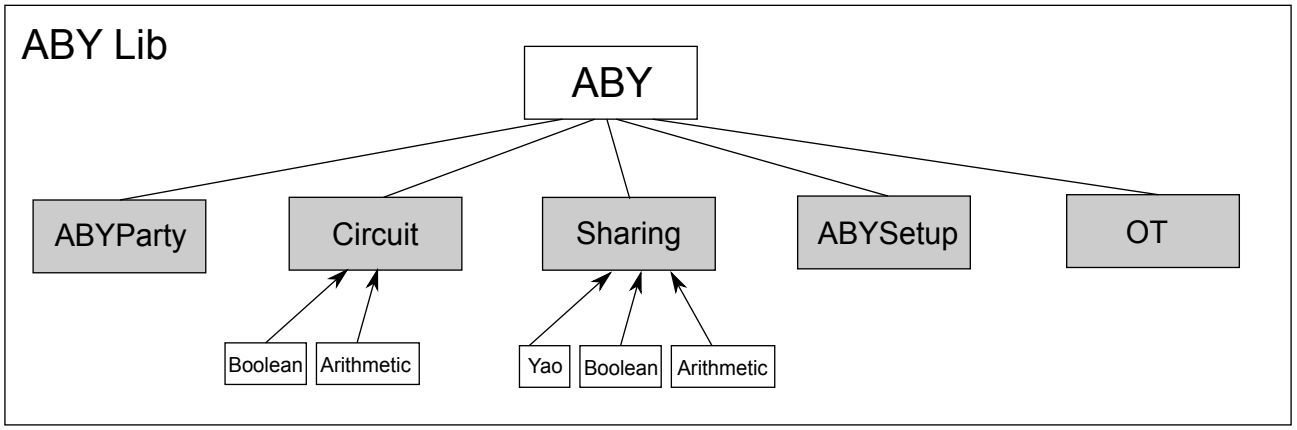

Figure 4.2: Architecture of our open source ABY library at `https://github.com/encryptogroup/ABY`. Upward arrows denote class inheritance.

interaction between the parties, but instead of evaluating cryptographic operations, only very fast arithmetic resp. bit operations are required.

While Boolean multiplication triples are pre-computed using OT, arithmetic multiplication triples can also be precomputed using additively homomorphic encryption. We analyzed two encryption schemes and compared their performance in Section 4.3: The work of Damgård-Jurik-Nielsen (DJN) [16, 17], which is a generalization of Paillier's encryption scheme [34] and the encryption scheme of of Damgård-Geisler-Krøigaard (DGK) [14, 15].

4.2 Implementation

The prototype implementation of the ABY secure computation framework from [21] is available as an open source project at `https://github.com/encryptogroup/ABY`.

An outline of the architecture is given in Figure 4.2. Here we give a high level overview of the classes that build up the framework. An instance of the **ABYParty** class is one of the two parties (server or client) performing the secure computation. A **Circuit** object is built up of gates and corresponds to the function that is to be evaluated. **Sharing**s correspond to the secure computation protocols available in ABY: arithmetic, Boolean or Yao sharing. **ABYSetup** is used for performing the offline phase of the secure computation. **OT** is the oblivious transfer extension implementation described in deliverables D13.3 [7] and D13.4 [8].

4.2.1 Architecture

1. Init – Initialization

In more details, in the first, initialization phase the following initializations take place that are independent of the function we want to evaluate:

1.1. Base OTs are generated,

1.2. Sharings are initialized,

1.3. Circuits are built.

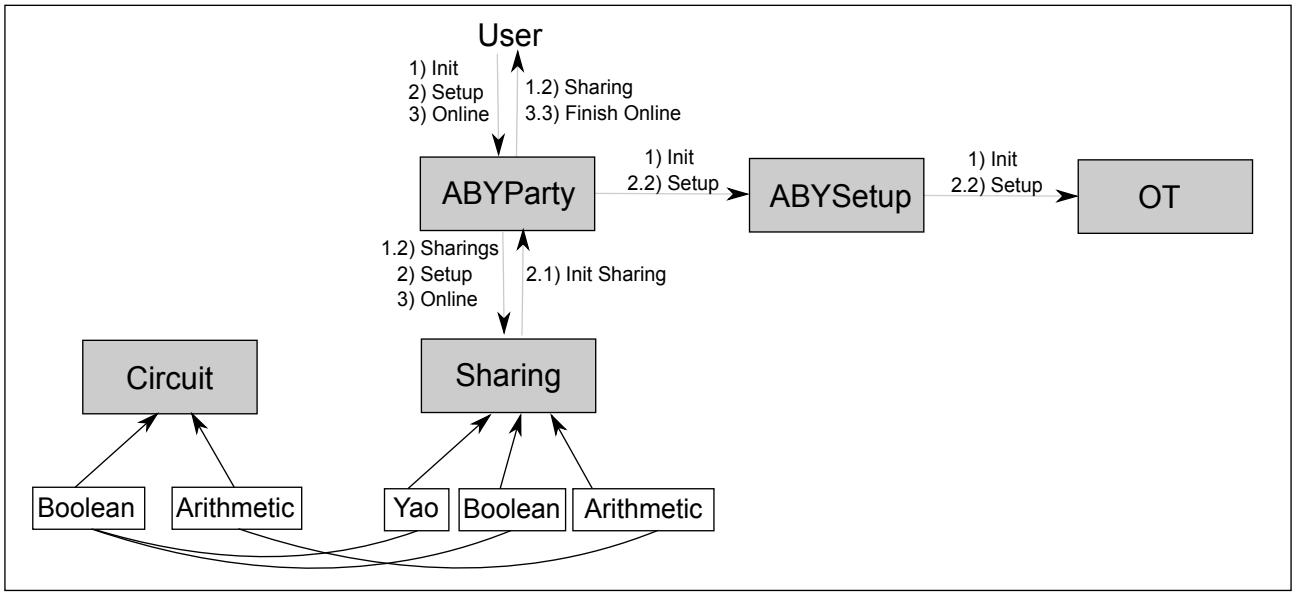

Figure 4.3: Detailed architecture of ABY. Upward arrows denote class inheritance, grey arrows denote communication.

2. Setup – Offline Phase

In the second, setup or offline phase all the precomputations are performed, i.e. the precomputations that may be dependent on the size of the circuit that is to be computed as well as the inputs of the parties.

2.1. Sharing Setup is initialized,

2.2. Setup phase is performed, e.g. OT extension or garbling for Yao sharing,

2.3. Setup phase is finished.

3. Online – Online Phase

In the last, online phase the secure evaluation of the circuit takes place:

3.1. Sharing Online is initialized,

3.2. For $d \in \{1, \ldots, \text{depth}\}$

3.2.1. Sharing layer d is initialized,

3.2.2. Sharing layer d is evaluated,

3.2.3. Sharing layer d is finished,

3.2.4. Parties communicate,

3.3. Online phase is finished.

The interactions between the ABY classes are shown in Figure 4.3.

4.2.2 Functions

In this section, we describe how a developer that works with the ABY framework can compute on secret shared values using several pre-defined *Function gates* and describe how to convert secret shared values between different secure computation schemes using *Conversion gates*.

Function Gates

Function gates are used to perform various computation operations on the provided circuit. Based on the provided circuit type, the gate operations are handled differently. Operations such as AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), MUX and GE ($\geq$) are considered to be Boolean circuit operations. Therefore, such operations are only implemented for circuits of type `C_BOOLEAN`. For compatibility of the operations supported by the various sharing types refer to Table 4.1.

All operations in Table 4.1 are possible with Boolean circuits (`C_BOOLEAN`) while some are not possible when using arithmetic circuits (`C_ARITHMETIC`).

Table 4.1: Operations

Operations	AND	XOR	OR	ADD	MUL	SUB	GE	MUX	CONS	INV
Arithmetic	✗	✗	✗	✓	✓	✓	✗	✗	✓	✓
Boolean	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Yao	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

In the rest of this section, we will provide detailed information on the various available pre-defined gate types.

PutANDGate `PutANDGate` performs a bitwise AND operation on the two input shares and returns the result as share object of the same bitlength as the inputs.

```
share* PutANDGate(share* ina, share* inb);
```

PutXORGate `PutXORGate` performs a bitwise XOR operation on the two input shares and returns the result as share object of the same bitlength as the inputs.

```
share* PutXORGate(share* ina, share* inb);
```

PutORGate `PutORGate` performs a bitwise OR operation on the two input shares and returns the result as share object of the same bitlength as the inputs.

```
share* PutORGate(share* ina, share* inb);
```

PutADDGate `PutADDGate` performs an arithmetic addition operation on the two input shares and returns the result as share object.

In arithmetic circuits the addition is carried out modulo 2^ℓ, where ℓ is the bitlength of the sharing.

```
share* PutADDGate(share* ina, share* inb);
```

PutMULGate `PutMULGate` performs an arithmetic multiplication operation on the two input shares and returns the result as share object

```
share* PutMULGate(share* ina, share* inb);
```

In arithmetic circuits the multiplication is carried out modulo 2^ℓ, where ℓ is the bitlength of the sharing.

PutSUBGate `PutSUBGate` performs an arithmetic subtraction operation on the two input shares and returns the result as share object.

```
share* PutSUBGate(share* ina, share* inb);
```

Parameters

- `ina` contains the minuend (the number to subtract from).
- `inb` contains the subtrahend (the number to subtract).

This gate computes `ina` − `inb`, i.e. it subtracts `inb` from `ina`. In arithmetic circuits the multiplication is carried out modulo 2^ℓ, where ℓ is the bitlength of the sharing. Thus the result is always positive.

PutGEGate `PutGEGate` performs an greater-or-equal operation ($\geq$) on the two input shares and returns a single bit result as share object.

```
share* PutGEGate(share* ina, share* inb);
```

This gate computes `ina` $\geq$ `inb`. It returns `1` if this is true and `0` otherwise.

PutMUXGate `PutMUXGate` implements a multiplexer and returns one of two given data inputs based on a selection bit.

```
share* PutMUXGate(share* ina, share* inb, share* sel);
```

Parameters

- `ina` share input containing the first value.
- `inb` share input containing the second value.
- `sel` selection bit input. `1` returns `ina`. `0` returns `inb`.

If the selection bit `sel` is 1, the content of `ina` is returned. If `sel` is 0, `inb` is returned.

PutCONSGate The `PutCONSGate` function can be used to input a constant plaintext value, which is known to both parties, into the circuit. The function returns a share object, which represents the secret-shared or encrypted constant.

```
share* PutCONSGate(uint64_t value, uint32_t bitlen);
```

Parameters

- `value` the value of the constant that is supposed to be secret-shared.
- `bitlen` the bit-length of the constant.

Conversion

The ABY framework allows to perform secure computation using Arithmetic, Boolean or Yao secure computation schemes and to arbitrarily convert secret-shared values between them. In order to perform the conversion of shares, *Conversion* gates can be used. Unlike the function gates, introduced in the previous section (Section 4.2.2), conversion gates do not change the secret-shared value. Instead, conversion gates transform the shares, held by each of the parties, from the representation of one secure computation scheme into another secure computation scheme.

In the following, we will use the short notation `A2B` to denote that a method that converts a share from Arithmetic to Boolean sharing. Given the existing schemes, this gives us six possible conversion methods: `A2B`, `A2Y`, `B2A`, `B2Y`, `Y2A`, `Y2B`. Note, however, that currently only four of these methods are implemented: `A2Y`, `B2A`, `B2Y`, and `Y2B`, as depicted in Figure 4.1. The remaining two methods, namely `A2B` and `Y2A` can be implemented by computing `Y2B(A2Y)` and `B2A(Y2B)`, respectively. Note that the conversion gate function needs to be done on a `Circuit` of the target sharing, i.e., the `A2Y` function would need to be invoked on a `Circuit` for Yao sharing.

A2Y The `A2Y` function converts an Arithmetic share into a Yao share. Note that the `A2Y` function needs to be called on a `Circuit` in Yao sharing.

```
share* A2Y(share* ina);
```

The returned share is a Yao share has the same plaintext value as the input Arithmetic share.

Example The following example secret shares two 32-bit numbers A and B in the Arithmetic sharing and multiplies them. The product is converted to Yao sharing and again multiplied by two.

```
1 share *shra, *shrb, *shrres;
2 Circuit* ac = sharings[S_ARITH]->GetCircuitBuildRoutine();
3 Circuit* yc = sharings[S_YAO]->GetCircuitBuildRoutine();
4 shra = ac->PutINGate(A, 32, SERVER);
5 shrb = ac->PutINGate(B, 32, CLIENT);
6 shrres = ac->PutMULGate(shra, shrb);
7 shrres = yc->PutA2YGate(shrres);
8 shrres = yc->PutADDGate(shrres, shrres);
```

B2A The `B2A` function converts a Boolean share into an Arithmetic share. Note that the `B2A` function needs to be called on a `Circuit` in Arithmetic sharing.

```
share* B2A(share* ina);
```

The returned share is a Arithmetic share and has the same plaintext value as the input Boolean share.

B2Y The `B2Y` function converts a Boolean share into a Yao share. Note that the `B2Y` function needs to be called on a `Circuit` in Yao sharing.

```
share* B2A(share* ina);
```

The returned share is a Yao share and has the same plaintext value as the input Boolean share.

Y2B The `Y2B` function converts a Yao share into a Boolean share. Note that the `Y2B` function needs to be called on a `Circuit` in Boolean sharing.

```
share* B2A(share* ina);
```

The returned share is a Boolean share and has the same plaintext value as the input Yao share.

4.3 Performance

We provide performance results of the ABY framework that we measured in two settings: a fast local network and a slower cloud network through the internet. The machines used in the local setting were two Desktop PCs, each using an Intel Haswell i7-4770K CPU with 3.5 GHz and 16 GB RAM, that are connected via Gigabit-LAN.

In the cloud setting, we run the benchmarks on two Amazon EC2 `c3.large` instances with a 64-bit Intel Xeon dualcore CPU with 2.8 GHz and 3.75 GB RAM. One virtual machine is located at the US east coast and the other one in Japan. The average throughput in this scenario was 70 MBit/s, while the latency was 170 ms.

In Table 4.2 we list numbers for the communication cost and average runtimes for generating multiplication triples (cf. Section 4.1) for typical data type sizes. The resulting numbers are amortized cost for generating a single multiplication triple out of 100 000, excluding fixed one-time computations.

The following tables show resulting performances and communication requirements of three use cases in which mixed-mode secure computation is beneficial:

- Modular exponentiation (Table 4.3), which can be used in public-key cryptography.
- Private Set Intersection (Table 4.4), which securely computes the intersection of two private sets.
- Biometric matching (Table 4.5), which computes an Euclidean distance to determine the similarity between biometric samples.

We provide runtimes in seconds, the amount of data that is exchanged between the parties and the number of communication rounds.

Table 4.2: Overall amortized complexities for generating one multiplication triple using Homomorphic Encryption or Oblivious Transfer Extension with two threads. Smallest values marked in bold.

	Communication [Bytes]				**Time** [μs]							
					Local				Cloud			
Bit-length	8	16	32	64	8	16	32	64	8	16	32	64
Paillier-based												
legacy	528	531	541	555	245	246	278	328	842	867	990	1 139
medium	1 039	1 043	1 051	1 067	1 430	1 475	1 572	1 748	4 485	4 654	5 198	5 669
long	1 551	1 555	1 563	1 579	4 309	4 374	4 565	4 957	12 990	13 080	13 805	14 614
DGK-based												
legacy	384	384	**384**	**384**	94	104	151	322	449	464	572	1 134
medium	768	768	**768**	**768**	259	313	465	1 020	971	1 128	1 651	3 107
long	1 152	1 152	**1 152**	**1 152**	534	629	929	2 005	1 894	2 118	3 049	6 319
Oblivious Transfer Extension-based												
legacy	**169**	**354**	772	1 800	**3**	**4**	**8**	**20**	**39**	**62**	**86**	**170**
medium	**233**	**482**	1 028	2 312	**3**	**6**	**10**	**24**	**44**	**77**	**107**	**219**
long	**265**	**546**	1 156	2 568	**3**	**6**	**11**	**27**	**46**	**82**	**110**	**224**

Table 4.3: Modular Exponentiation: Setup, Online, and Total run-times (in s), communication, and number of messages for the modular exponentiation on `len`= 32-bit inputs and long-term security. Smallest entries marked in bold.

	Local			**Cloud**			**Comm.**	**#Msg**
	S	O	T	S	O	T	**[MB]**	
Y-only	0,9	0,4	1,3	5,8	**0,9**	**6,7**	27.1	**2**
A+B+Y	0,6	**0,3**	**0,9**	5,6	29,5	35,1	**18.7**	353

Table 4.4: PSI: Setup, Online, and Total run-times (in s), communication, and number of messages for the Private Set Intersection application on $n = 4\,096$ elements of length $\sigma = 32$-bits and long-term security. Smallest entries marked in bold.

	Local			**Cloud**			**Comm.**	**#Msg**
	S	O	T	S	O	T	**[MB]**	
Y-only	3,5	0,7	4,3	32,2	**1,8**	34,0	247	**2**
B-only	2,0	**0,6**	**2,6**	11,5	22,6	34,1	**163**	123
B+Y	2,6	0,7	3,3	23,4	7,1	**30,0**	182	27

Table 4.5: Biometric Identification: Setup, Online, and Total run-times (in s), communication, and number of messages for biometric identification on 512 elements with a length of $\sigma = 32$-bits and with dimension $d = 4$ and long-term security. Smallest entries marked in bold.

	Local			**Cloud**			**Comm.**	**#Msg**
	S	O	T	S	O	T	**[MB]**	
Y-only	2,24	0,31	2,55	23,78	0,84	24,62	147.7	**2**
B-only	2,15	0,28	2,43	10,34	29,07	39,41	99.9	129
A+Y	0,14	**0,05**	**0,19**	2,98	**0,44**	**3,42**	5.0	8
A+B	0,08	0,13	0,21	2,34	24,07	26,41	**4.6**	101

Chapter 5

Formally Verified Implementation of Yao's SFE Protocol

The PRACTICE formally verified secure computation framework was outlined in deliverable D22.2 [32], and specified in detail in D12.3 [12]. One fundamental component of this framework is a formally verified secure computation protocol suite that is capable of evaluating arbitary computations expressed as Boolean circuits. The work dedicated to the implementation and validation of this protocol, namely the effort required to obtain a mechanised proof of security and correctness, as well as a formally verified implementation, were conducted within WP14 and will be reported in this document. The integration of the resulting implementation, which is automatically generated as OCaml code, into the FRESCO framework is presented in Deliverable D14.4 as part of the validation of the general architecture for secure computation engines developed in WP14. The work reported here has also been closely integrated with activities carried out in WP22, namely in the development of formally verified computation specification generation tools that are compatible with the protocol implementation that we present here. This complementary effort is presented in D22.4 [35]. With respect to the status of the work presented at the end of year 2, the material in this Chapter describes the effort carried out to conclude the verification work (namely the verification of the low-level garbling procedure, and which included a full migration to the new version of EasyCrypt), and finally a whole new extraction of the verified code.

5.1 Protocol Description

Yao's protocol was described in D12.3 [12], where the formal verification requirements for this high-assurance secure computation protocol suite were specified. We also provide a succinct description here, so as to facilitate the understanding of our description of the verification and implementation work. Also described in detail in D12.3 are the potential usages of Yao's protocol in a wide range of applications; we omit a discussion of these applications here and refer the interested reader to D12.3 for more details.

Yao's protocol allows one to perform two-party secure function evaluation, i.e., to securely compute functions expressed as circuits composed of Boolean gates, that output a single value to be revealed to both parties, and that take secret inputs from both parties. Informally, Yao's idea of garbling circuits consists of:

- Expressing a circuit as a set of truth tables with information about the wiring between gates.

- *Garble* the Boolean values in the truth tables by replacing the Boolean values with random cryptographic keys, or labels.
- Translate the wiring relation using a system of locks, meaning that for each possible combination of the input wires, the corresponding labels are used as encryption keys that lock the label for the correct Boolean value at the output of that gate.

The evaluation of a garbled circuit is straightforward: given a set of labels representing values for the input wires encoding the inputs of both parties, only one entry in the corresponding truth table will be decryptable, revealing the label of the output wire. The output of the circuit will comprise the labels at the output wires of the output gates.

There are two security assets regarding Yao's protocol: i. unless one is given the association between labels and Boolean values for input and output wires, no information about the input or output of the circuit are revealed; ii. given the label/Boolean value associations for x_1 and the output wires of the circuit, but only an encoding of x_2 in the form of an input label assignment, the garbled circuit reveals nothing other than $f(x_1, x_2)$ about x_2.

To build an SFE protocol between two honest-but-curious parties, one can use Yao's garbled circuits as follows:

1. Bob (holding x_2) garbles the circuit and provides this to Alice (holding x_1) along with the label assignment for the input wires corresponding to x_2 and all the information required to decode the Boolean values of the output wires.
2. Using an oblivious transfer (OT) protocol, Alice obtains the labels that encode x_1 from Bob, without revealing anything about x_1 and learning no more than the labels she requires.
3. Finally, Alice evaluates the circuit, recovering the output, and providing the output value back to Bob.

An oblivious transfer protocol is a particular case of a two-party SFE protocol that will allow one party to obtain the labels corresponding to the encoding of its input without revealing anything about it and without the party learning anything more than the labels it requires.

Formally, party P_1 inputs to the protocol an array of bits of size n, i.e., $I_1 = (x_1, \ldots, x_n) \in 0, 1^n$. Party P_2 inputs to the protocol an n-tuple of pairs of tokens, i.e.,

$$I_2 = ((X_1^0, X_1^1), \ldots, (X_n^0, X_n^1)) \in (\mathsf{Token} \times \mathsf{Token})^n$$

where Token is the type of keys (or labels) associated with the Boolean values of wires in a garbled circuit (bitstrings of fixed value). The evaluation algorithms ev establish that Party P_2 receives no local output at the end of the protocol, whereas Party P_1 obtains $Y = (X_{i_1}^1, ..., X_{i_n}^n)$. We require that ev is so defined for all $n > 0$, thereby imposing that the OT protocol is correct for arbitrary input lengths.

5.2 Implementation

Our verified implementation of Yao's protocol was obtained in three steps. We first specified the protocol in EasyCrypt [3] (cf. D22.1 [25] and D12.3 [12]), an interactive proof assistant for cryptography. We then used this tool to prove the protocol correct and secure, according to the goals specified in D12.3. Finally, we used EasyCrypt and the Why3 [37] framework to extract an

OCaml implementation of the verified protocol, which preserves the correctness and security properties of the EasyCrypt specification by construction. We next describe each of these steps in detail.

5.2.1 Formalizing and verifying Yao's Protocol in EasyCrypt

Yao's SFE protocol is a specific concretisation of an abstract notion of a two-party protocol, that results of the composition of an oblivious transfer (OT) protocol (also a particular case a two-party protocol) and a garbling scheme.

5.2.1.1 Two-party protocols

We first start by generically defining two-party protocols, that generalize both Secure Function Evaluation and Oblivious Transfer, and their security. A generic two-party secure function evaluation protocol, such as that proposed by Yao, is viewed formally as tuple (Π, ev), where $\Pi = (\Pi_1, \Pi_2)$ and $\mathsf{ev} = (\mathsf{ev}_1, \mathsf{ev}_2)$ are pairs of probabilistic polynomial-time (ppt) algorithms.

Intuitively, algorithms ev_1 and ev_2 deterministically compute functions $f_1(I_1, I_2)$ and $f_2(I_1, I_2)$ that represent the outputs recovered by each party at the end of the protocol. Algorithms Π represent the implementation of the cryptographic protocol that will be executed between the two parties P_1 and P_2. Party P_1 will run Π_1 and party P_2 will run Π_2. Each party i iteratively runs Π_i on its current state and an incoming message to produce an outgoing message, a local output, and a decision to halt or continue. The initial state of party i is its private input I_i. The incoming message to the party initiating the protocol can be taken to be the empty string.

In EasyCrypt, the generic definition of a two-party protocol is obtained by combining a series of abstract declarations:

- The types of the inputs and outputs of each party.
- The randomness they require to execute the protocol.
- The admissible leakage of each input. Here, *leakage* refers to the amount of information that the protocol is allowed to reveal about some input (for example, its length).
- The function to be computed.
- The exchanged messages throughout the execution of the protocol, that we denote by conv.
- Predicates that enforce well-formedness restrictions on inputs and randomness.
- The protocol itself.

Note that these elements are just abstract types or function headers, that will later be fulfilled with concrete types and operations in order to obtain concrete implementations of two-party protocols. Our abstract formalisation of a two-party protocol can be found in Figure 5.1.

The security of a two-party SFE protocol is defined as follows. For $i = 1, 2$, consider an adversary $\mathbf{Adv}_i = (\mathbf{Adv}_i^1, \mathbf{Adv}_i^2)$ that represents a malicious party i, and operates as follows: i. $\mathbf{Adv}_i^1$ takes no input and outputs a pair (I_1, I_2), and possibly some state information st; ii. on input of view view and state st, $\mathbf{Adv}_i^2$ outputs a bit b.

```
theory Protocol.
 type input1, output1.
 type input2, output2.
 type leak1, leak2.

 op f: input1 → input2 → output1 * output2.

 type rand1, rand2, conv.
 op prot: input1→ rand1→ input2→ rand2→ conv * output1 * output2.

 op validInputs: input1 → input2 → bool.
 pred validRands: (input1, input2, rand1, rand2).

 op Ψ1: input1 → leak1.
 op Ψ2 : input2 → leak2.
end Protocol.
```

Figure 5.1: Abstract Two-Party Protocol.

A two-party SFE protocol is secure in the presence of semi-honest adversaries if for all ppt adversaries $\mathbf{Adv} = (\mathbf{Adv}_1, \mathbf{Adv}_2)$, there exists a ppt simulator $\mathsf{S} = (\mathsf{S}_1, \mathsf{S}_2)$ such that, the probability of $\mathbf{Adv}_i$ to distinguish between the *real-world* and the *ideal-world* experiment is negligible, i.e. the adversary $\mathbf{Adv}_i$ is not able to distinguish whether its view has been provided by the protocol or by the simulator.

In EasyCrypt, we express this security notion using the module defined in Figure 5.2. Notice that in our definition the two algorithms $(\mathbf{Adv}_i^1, \mathbf{Adv}_i^2)$ are provided by the same module $\mathbf{Adv}_i$ and so can share state information. We define the advantage of an adversary $\mathbf{Adv}_i$ against a two-party protocol prot with leakage $\Psi = (\Psi_1, \Psi_2)$, running with simulator S_i, and randomness generators R_1 and R_2 as

$$\mathsf{Adv}^{\mathsf{prot}^i_\Psi}_{\mathsf{prot},\mathsf{S}_i,\mathsf{R}_1,\mathsf{R}_2}(\mathbf{Adv}_i) = \left|2 \cdot \Pr\left[\mathsf{Sec}_i(\mathsf{R}_1, \mathsf{R}_2, \mathsf{S}_i, \mathbf{Adv}_i) : \mathsf{res}\right] - 1\right| .$$

The intuition of this simulation-based definition is that the existence of a successful simulator establishes that the view of party P_1 (resp. party P_2) cannot possibly release more information about I_2 (resp. I_1) in addition to the information received by the simulator, which includes the evaluation result destined to that party and the admissible leakage. Security with respect to leakage function Ψ clearly implies security with respect to leakage function Ψ' that releases more information about the secret inputs. Indeed, given the leakage produced by Ψ', it is easy to remove information from it to obtain the leakage produced by Ψ before running the simulator.

Correctness and security definitions for two-party protocols are parametrized by the Protocol theory. This allows us to instantiate these notions, as well as some generic lemmas (e.g. to manipulate probabilities conditioning on ideal/real worlds) with any two-party protocol that fits our abstract definition.

Oblivious Transfer. In EasyCrypt, an abstract OT protocol is a refinement of an two-party protocol that imposes the type definitions described above, but still leaves some other types undefined (e.g. the type of randomness) to be fixed by concrete protocols. Our partial instantiation is shown in Figure 5.3. Defining OT security is then simply a matter of instantiating the general notion of security for two-party protocols via cloning. Looking ahead, we use $\mathbf{Adv}^{\mathsf{OT}^i}$ to denote the resulting instance of $\mathbf{Adv}^{\mathsf{prot}^i_{(|\cdot|,|\cdot|,|\cdot|)}}$, and similarly, we write $\mathbf{Adv}_i^{\mathsf{OT}}$ the types for adversaries against the OT instantiation.

```
type leak1, leak2.   op φ1 : input1 → leak1.   op φ2 : input2 → leak2.
type view1 = rand1 * conv.   type view2 = rand2 * conv.

module type Sim = {
  proc sim1(i1: input1, o1: output1, l2: leak2) : view1
  proc sim2(i2: input2, o2: output2, l1: leak1) : view2
}.

module type Simi = {
  proc simi(ii: inputi, oi: outputi, l3−i: leak3−i) : viewi
}.

module type Advprot_i = {
  proc choose(): input1 * input2
  proc distinguish(v: viewi) : bool
}.

module Sec1(R1: Rand1, R2: Rand2, \ec{Sim}: Sim1, Adv1: Advprot_1) = {
  proc main() : bool = {
    var real, adv, view1, o1, r1, r2, i1, i2;
    (i1,i2) = Adv1.choose();
    real <$- {0,1};
    if (!validInputs i1 i2)
      adv <$- {0,1};
    else {
      if (real) {
        r1 = R1.gen(φ1 i1);
        r2 = R2.gen(φ2 i2);
        (conv,_) = prot i1 r1 i2 r2;
        view1 = (r1, conv);
      } else {
        (o1,_) = f i1 i2;
        view1 = \ec{Sim}.sim1(i1, o1, φ2 i2);
      }
      adv = Adv1.distinguish(view1);
    }
    return (adv = real);
  }
}.
```

Figure 5.2: Security of a two-party protocol protocol.

5.2.1.2 Garbling Schemes

Bellare et al.[5] view Yao's garbled circuits as a particular case of a new encryption primitive called garbling scheme. A garbling scheme $\mathcal{G}$ is a five-tuple of ppt algorithms (Gb, En, De, Ev, ev) where:

- Gb is the garbling algorithm which, on input a circuit f implementing a function of type $\{0,1\}^n \to \{0,1\}^m$, outputs a garbled version of this circuit F, along with an encoding key $e \in (\mathsf{Token} \times \mathsf{Token})^n$.
- En is the input encoding algorithm. We take this to be the specific mapping that, on input a bit string $x = x_1, \ldots, x_n \in \{0,1\}^n$ and the encoding key $e = (X_1^0, X_1^1), \ldots, (X_n^0, X_n^1)$, outputs garbled input $X = X_x^1, ..., X_x^n$.
- Ev is the garbled evaluation algorithm which, given a garbled circuit F and a garbled input X, produces a garbled output Y .
- De is the output decoding algorithm. We take this to be a public efficient mapping that, given a garbled output $Y \in (\mathsf{Token})^m$, outputs $y = y_1, \ldots, y_m \in \{0,1\}^m$. In the concrete garbling scheme we formalised and implemented this corresponds to taking the least significant bit of a token.

```
clone Protocol as OT with
  type input1 = bool array,
  type output1 = msg array,
  type leak1 = int,
  type input2 = (msg * msg) array,
  type output2 = unit,
  type leak2 = int,
  op φ1 (i1 : bool array) = length i1,
  op φ2 (i2 : (msg * msg) array) = length i2,
  op f (i1 : bool array) (i2 : (msg * msg) array) = i1_{i2}.
  op validInputs(i1 : bool array) (i2 : (msg * msg) array) =
      0 < length i1 ≤ n_max ∧ length i1 = length i2,
...
```

Figure 5.3: Instantiating Two-Party Protocols into Abstract OT.

- ev is the deterministic (cleartext) evaluation algorithm that describes the functionality of the scheme. Given a a circuit f implementing a function of type $\{0,1\}^n \to \{0,1\}^m$ and an input $x = x_1, \ldots, x_n \in \{0,1\}^n$, it produces an evaluated output $y \in \{0,1\}^m$.

Abstract garbling schemes are captured in EasyCrypt via the type and operator declarations presented in Figure 5.4. We only consider *projective schemes* [5], where Boolean values on each wire are encoded using a fixed-length random *token*. This fixes the type funcG of garbling schemes, and the outputK and decode operators.

```
type func, input, output.
op eval : func → input → output.
op valid: func → input → bool.

type rand, funcG, inputK, outputK.
op funcG : func → rand → funcG.
op inputK : func → rand → inputK.
op outputK: func → rand → outputK.

type inputG, outputG.
op evalG : funcG → inputG → outputG.
op encode: inputK → input → inputG.
op decode: outputK → outputG → output.
```

Figure 5.4: Abstract Garbling Scheme.

The security of a garbling scheme is defined using a simulation-based notion of security. Consider a ppt adversary $\mathbf{Adv} = (\mathbf{Adv}_1, \mathbf{Adv}_2)$ that operates as follows: i. $\mathbf{Adv}_1$ takes no input and outputs a pair (f, x), and possibly some state information st; ii. on input a garbled circuit and input pair (F, X) and state st, $\mathbf{Adv}_2$ outputs a bit b.

A garbling scheme is SIM-CPA$_\Psi$-secure if, for every $\mathbf{Adv}$ outputting valid (f, x) pairs, there exists a simulator S that, on input ev(f, x), where (f, x) has been generated by $\mathbf{Adv}_1$, and some leakage $\Psi(f)$, outputs a garbled circuit and input pair (F, X) such that the adversary $\mathbf{Adv}_2$ has a low probability to distinguish between a pair (F, X) generated using the garbling scheme or generated using the simulator S, i.e. $(F, X) \leftarrow \mathsf{S}(\mathsf{ev}(f, x), \Psi(f))$.

Intuitively, the definition states that if a valid S exists, then real ciphertexts do not (computationally) leak more information than simulated ones, and these cannot possibly contain more than the information given to the simulator, i.e., the value of $f(x)$ output and the value of $\Psi(f)$, where Ψ is a leakage function, that dictates which parts of the circuit description can be leaked by the scheme.

We define the SIM-CPA$_\Psi$ advantage of an adversary $\mathbf{Adv}$ against garbling scheme enc and simulator S as

$$\mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_\Psi}_{\mathsf{Gb},\mathsf{R},\mathsf{S}}(\mathbf{Adv}) = |2 \cdot \Pr\left[\mathsf{SIM}(\mathsf{R},\mathsf{S},\mathbf{Adv}) : \mathsf{res}\right] - 1|$$

We note that all of the above security definitions are parametrized by randomness sampling algorithms R, which allows the specifications of schemes to be deterministic and explicitly take randomness. Theorem statements then quantify over all possible randomness generation algorithms, which means that they hold in particular for the standard definitions where randomness is sampled uniformly at random from the set of values in the appropriate data type.

5.2.1.3 SFE from Garbling Schemes and OT

Our construction of Yao's generic SFE protocol follows closely what is described in [5], with a slight adaptation that enables the input to the circuit to be split between the two parties, which is natural in projective schemes. Intuitively this corresponds to party P_2 hardwiring its part of the input x_2 in the garbled circuit before sending it to P_1. The protocol relies on a garbling scheme $\mathcal{G}$ and an OT protocol Π_{OT} as described above. For inputs $I_1 = x_1$ and $I_2 = (x_2, f)$, where f takes inputs of length n, $|x_1| = n_1$ and $|x_2| = n_2$ such that $n = n_1 + n_2$, the two parties proceed as follows:

1. Party P_2 uses the garbling scheme to compute $(F, e) \leftarrow_{\$} \mathsf{Gb}(f)$. It then splits the encoding key

   $$e = ((X_1^0, X_1^1), ..., (X_n^0, X_n^1))$$

   into two sequences

   $$e_1 = ((X_1^0, X_1^1), ..., (X_{n_1}^0, X_{n_1}^1))$$
   $$e_2 = ((X_{n_1+1}^0, X_{n_1+1}^1), ..., (X_n^0, X_n^1))$$

   and encodes x_2 as

   $$X_2 = (X_{n_1+1}^{x_2^1}, \ldots, X_n^{x_2^{n_2}})$$

   Finally it sends (F, X_2) to P_1.

2. The two parties execute the OT protocol on (x_1, e_1), so that P_1 obviously obtains an encoding of x_1 as

   $$X_1 = (X_1^{x_1^1}, \ldots, X_{n_1}^{x_1^{n_1}})$$

3. P_2 then uses the garbled evaluation algorithm to compute $Y \leftarrow^{\$} \mathsf{Ev}(F, X_1||X_2)$ and decodes $y \leftarrow \mathsf{De}(Y)$.

We express this particular view of Yao's protocol in EasyCrypt by refining the abstract view we presented earlier as follows (some notations are simplified for clarity) in Figure 5.5.

Again, the definition of security for such an SFE protocol is obtained by instantiation of the general security notion we presented in Section 5.2.1.1. We note that these definitions are still abstract, i.e., they need to be parameterized by a concrete OT protocol and a concrete garbling scheme. However, EasyCrypt allows us to define and prove the correctness and security of such

```
clone Garble as Gb.
clone OT as OT.

clone Protocol as SFE with
  type rand1 = OT.rand1,
  type input1 = bool array,
  type output1 = Gb.output,
  type leak1 = int,
  type rand2 = OT.rand2 * Gb.rand,
  type input2 = Gb.func * bool array,
  type output2 = unit,
  type leak2 = Gb.func * int,
  op f i1 i2 = let (c,i2) = i2 in Gb.eval c (i1 || i2),(),
  type conv = (Gb.funcG * token array * Gb.outputK) * OT.conv,
  op validInputs (i1:input1) (i2:input2) =
    0 < length i1 ∧ Gb.validInputs (fst i2) (i1 || snd i2),
  op prot (i1:input1) (r1:rand1) (i2:input2) (r2:rand2) =
    let (c,i2) = i2 in
    let fG = Gb.funG c (snd r2) in
    let oK = Gb.outputK c (snd r2) in
    let iK = Gb.inputK c (snd r2) in
    let iK1 = (take (length i1) iK) in
    let (ot_conv, (t1,_)) = OT.prot i1 r1 iK1 (fst r2) in
    let GI2 = Gb.encode (drop (length i1) iK) i2 in
    (((fG,GI2,oK),ot_conv), (Gb.decode oK (Gb.evalG fG (t1 || GI2)),())).
```

Figure 5.5: Abstract SFE Construction.

a protocol at this abstract level, based on the security and correcntess of these underlying (unspecified) components, as follows.

Take any oblivious transfer protocol OT and any garbling scheme Gb, and let SFE_a be the SFE protocol built using Yao's construction from OT and Gb. Fix also arbitrary randomness generators R^{Gb}, $\mathsf{R}_1^{\mathsf{OT}}$ and $\mathsf{R}_2^{\mathsf{OT}}$, which parametrize the games that define underlying assumptions of security on the garbling scheme, and let $\mathsf{R}_1^{\mathsf{SFE}}$ and $\mathsf{R}_2^{\mathsf{SFE}}$ be the induced randomness generation modules for for the SFE security games, when these are instantiated with our construction. Then the following theorem was proven in EasyCrypt.

Theorem 1 (Abstract SFE security) *For all SFE adversaries* $\mathbf{Adv} = (\mathbf{Adv}_1, \mathbf{Adv}_2)$*, OT simulators* S^{OT} *and garbling simulator* S^{Gb}*, we can construct efficient adversaries* $\mathbf{Adv}^{\mathsf{OT}} = (\mathbf{Adv}_1^{\mathsf{OT}}, \mathbf{Adv}_2^{\mathsf{OT}})$ *and* $\mathbf{Adv}^{\mathsf{Gb}}$ *and an efficient simulator* S*, such that the following inequalities hold.*

$$\mathsf{Adv}^{\mathsf{SFE}_\Psi^1}_{\mathsf{SFE},\mathsf{S}}(\mathbf{Adv}_1) \leq \mathsf{Adv}^{\mathsf{OT}_\Psi^1}_{\mathsf{OT},\mathsf{S}^{\mathsf{OT}}}(\mathbf{Adv}_1^{\mathsf{OT}}) + \mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_\Psi}_{\mathsf{Gb},\mathsf{S}^{\mathsf{Gb}}}(\mathbf{Adv}^{\mathsf{Gb}})$$

$$\mathsf{Adv}^{\mathsf{SFE}_\Psi^2}_{\mathsf{SFE},\mathsf{S}}(\mathbf{Adv}_2) \leq \mathsf{Adv}^{\mathsf{OT}_\Psi^2}_{\mathsf{OT},\mathsf{S}^{\mathsf{OT}}}(\mathbf{Adv}_2^{\mathsf{OT}})$$

The proof of security of the protocol follows the structure outlined in [5]. One first shows that any attacker against the SFE protocol can be converted into adversaries attacking the simulation-based security of the garbling scheme ($\mathbf{Adv}^{\mathsf{Gb}}$) and OT protocol ($\mathbf{Adv}_1^{\mathsf{OT}}$) and ($\mathbf{Adv}_2^{\mathsf{OT}}$). The assumption that the underlying garbling scheme is secure ensures the existence of an algorithm S^{Gb} that successfully simulates garbled circuits. Similarly, security of the underlying OT protocol ensures the existence of an algorithm S^{OT} that successfully simulates OT traces. Note that the statement of the lemma is quantified for all such simulators.

Such simulators can be used to construct a simulator S for the generic SFE protocol as follows. To simulate the view of P_1, simulator S uses S^{Gb} to create (F, e), follows the procedure of P_2 in creating X, and then S^{OT} to generate a valid view for the OT protocol. The proof

that such a simulation works is structured as a sequence of two game hops that permit upper-bounding the computational distance between the real world and an ideal world instantiated with S. To simulate the view of P_2, S is able to run all of the garbling operations itself, and then use S^{OT} to simulate the OT view. The proof that this simulation is correct is a direct reduction to OT security.

The SFE protocol proved secure in the previous theorem is still abstract, and is parameterized by a secure garbling scheme and a secure oblivious transfer protocol. Therefore, we now provide a description of the concrete realisations of the two primitives and their security proofs. In the rest of this section, the randomness generators and schemes are left implicit and correspond to our concrete instantiations. We replace them with the name of the concrete theory as indices in advantages.

5.2.1.4 A concrete garbling scheme: SomeGarble

Following the Garble1 construction of Bellare et al. [5], we construct our garbling scheme using a variant of Yao's garbled circuits based on a pseudo-random permutation, via an intermediate Dual-Key Cipher (DKC) construction, composed of two functions:

- E - takes as input a tweak (unique IV), two keys and a plaintext and returns a ciphertext.
- D - takes as input a tweak (unique IV), two keys and a ciphertext and returns a plaintext.

We give functional specifications to the garbling algorithms in Figure 5.6. For clarity, we denote functional folds using stateful for loops. The circuit evaluation algorithm eval is defined as expected, and the outputK type and decode function are omitted below since they are fixed by the convention we adopted in the previous section that we are dealing with a projective scheme.

To make use of Theorem 1, when composing this scheme with a secure OT protocol, we need to prove that it is $\mathsf{SIM\text{-}CPA}_{\Psi_{topo}}$-secure. Following Bellare et al.'s definitions [5], the definition of DKC security suffices to prove security of SomeGarble. Therefore, the security proof is made by reducing the security of SomeGarble to the security of the underlying DKC scheme. In order to prove $\mathsf{SIM\text{-}CPA}_{\Psi_{topo}}$ security, we make of a more convenient security notion that informally states that, under certain conditions on the leakage function (if it is efficiently invertible), $\mathsf{IND\text{-}CPA}_{\Psi_{topo}}$-security implies $\mathsf{SIM\text{-}CPA}_{\Psi_{topo}}$-security. This result is discussed below as Lemma 1. Following [5], we view projective garbling schemes as a form of deterministic encryption. Therefore, we adopt a classical definition of $\mathsf{IND\text{-}CPA}_{\Psi_{\mathsf{topo}}}$-security, represented in Figure 5.7

We define the IND-CPA advantage of an adversary $\mathbf{Adv}$ against the encryption operator enc with leakage Ψ as

$$\mathsf{Adv}^{\mathsf{IND\text{-}CPA}_{\Psi}}_{\mathsf{enc},\mathsf{R}}(\mathbf{Adv}) = |2 \cdot \Pr[\mathsf{Game}_{\mathsf{I}}\mathsf{ND}(\mathsf{R}, \mathbf{Adv}) : \mathsf{res}] - 1|$$

where R is the randomness generation module used in the concrete theory.

In the rest of this subsection, we use the following notion of invertibility defined in [5]. A leakage function Ψ on plaintexts (when we instantiate this notion on garbling schemes these plaintexts are circuits and their inputs) is *efficiently invertible* if there exists an efficient algorithm that, given the leakage corresponding to a given plaintext, can find a plaintext consistent with that leakage.

Lemma 1 (IND-CPA-security implies SIM-CPA-security) *If Ψ is efficiently invertible, then for every efficient* SIM-CPA *adversary* $\mathbf{Adv}$*, one can build an efficient* IND-CPA *adversary* $\mathbf{Adv}'$ *and an efficient simulator* S *such that*

$$\mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_{\Psi}}_{\mathsf{enc},\mathsf{S}}(\mathbf{Adv}) = \mathsf{Adv}^{\mathsf{IND\text{-}CPA}_{\Psi}}_{\mathsf{enc}}(\mathbf{Adv}').$$

```
type topo = int * int * int * int array * int array.
type α circuit = topo * (int * α * α,α) map.

type leak = topo.

type input, output = bool array.
type func = bool circuit.

type funcG = token circuit.
type inputG, outputG = token array.
op evalG f i =
  let ((n,m,q,𝔸,𝔹),𝔾) = f in
  let evalGate = λ g x1 x2,
    let x1,0 = lsb x1 and x2,0 = lsb x2 in
    D (tweak g x1,0 x2,0) x1 x2 𝔾[g,x1,0,x2,0] in
  let wires = extend i q in (* extend the array with q zeroes *)
  let wires = map (λ g, evalGate g 𝔸[g] 𝔹[g]) wires in (* decrypt wires *)
  sub wires (n + q − m) m.

type rand, inputK = ((int * bool),token) map.
op encode iK x = init (length x) (λ k, iK[k,x[k]]).

op inputK (f:func) (r:((int * bool),token) map) =
  let ((n,_,_,_,_),_) = f in filter (λ x y, 0 ≤fst x < n) r.

op funcG (f:func) (r:rand) =
  let ((n,m,q,𝔸,𝔹),𝔾) = f in
  for (g,xa,xb) ∈ [0..q] * bool * bool
    let a = 𝔸[g] and b = 𝔹[g] in
    let ta = r[a,xa] and tb = r[b,xb] in
    𝔾̃[g,ta,tb] = E (tweak g ta tb) ta tb r[g,𝔾[g,xa,xb]]
  ((n,m,q,𝔸,𝔹),𝔾̃).
```

Figure 5.6: SomeGarble: our Concrete Garbling Scheme.

Using the inverter for Ψ, $\mathbf{Adv}'$ computes a second plaintext from the leakage of the one provided by $\mathbf{Adv}$ and uses this as the second part of her query in the IND-CPA game. Similarly, simulator S generates a simulated view by taking the leakage it receives and computing a plaintext consistent with it using the Ψ-inverter. The proof consists in establishing that $\mathbf{Adv}$ is called by $\mathbf{Adv}'$ in a way that coincides with the SIM-CPA experiment when S is used in the ideal world, and is performed by code motion.

Armed with Lemma 1, it is sufficient to prove that SomeGarble is IND-CPA$_{\Psi_{topo}}$-secure and that Ψ_{topo} is efficiently invertible to securely use SomeGarble in Yao's construction. We reduce the IND-CPA$_{\Psi\text{topo}}$-security of SomeGarble to the DKC-security of the underlying DKC primitive (see [5]). This is the most intricate part of the proof, and it involved a significant effort, since it is based on a highly intricate hybrid argument. We leave a description of this proof step to the end, and first wrap up our description of the proof.

From Lemmas 1 and the IND-CPA security of our garbling scheme, we can conclude with a security theorem for our garbling scheme.

Theorem 2 (SomeGarble is SIM-CPA$_{\Psi_{\text{topo}}}$-secure) *For every* SIM-CPA *adversary* $\mathbf{Adv}$*, one can construct an efficient simulator* S *and a DKC adversary* $\mathbf{Adv}'$ *such that*

$$\mathsf{Adv}^{\mathsf{SIM\text{-}CPA}_{\Psi_{\text{topo}}}}_{\mathsf{SomeGarble},\mathsf{S}}(\mathbf{Adv}) \leq (\mathsf{bound}+1)\cdot \mathsf{Adv}^{\mathsf{DKC}}_{\mathsf{SomeGarble}}(\mathbf{Adv}').$$

The bound factor is the size of the circuits provided by the adversary. The proof is performed with an implicit quantification over bound. In practice we set a global bound maxBound on it that must be taken into consideration when choosing concrete parameters at implementation-time.

```
module type Adv^IND = {
  fun choose(): ptxt * ptxt
  fun distinguish(c:ctxt): bool
}.

module IND (R:Rand, A:Adv^IND) = {
  fun main(): bool = {
    var p0, p1, p, c, b, b', ret, r;
    (p0,p1) = A.choose();
    if (valid p0 ∧ valid p1 ∧ Ψ p0 = Ψ p1) {
      b <$- {0,1};
      p = if b then p1 else p0;
      r = R.gen(|p|);
      c = enc p r;
      b' = A.distinguish(c);
      ret = (b = adv);
    }
    else ret <$- {0,1};
    return ret;
  }
}.
```

Figure 5.7: Indistinguishability-based Security for Garbling Schemes.

PROVING SOMEGARBLE SECURE. As already mentioned above, the proof of security of SomeGarble is done by reduction of the security of the DKC primitive used when defining SomeGarble. We now provide a description of the security proof. We omit the proof of correctness, as it is simply made by proving that any bit of the output of ev bitstring corresponds to the decoding of the corresponding label obtained via En. To perform the proof, we follow the indications in [5], with some deviations to make the proof amenable to verification in reasonable time. The result we prove is slightly weaker than that originally established by [5], but it suffices to cover the instantiation of the Dual Key Cipher construction that we adopt in our verified implementation. Details follow.

Consider the following circuit topology. Integers n, m and q define the number of input wires, number of output wires and number of gates, respectively, vectors A and B save the first and second incoming wires to some gate and map G describes the functionality of each gate. Vectors A and B are defined for values of wires in $[0 \ldots n+q[$, whereas map G is defined for values of gates in $[n \ldots n+q[$. The proof is organised by means of three cryptographic *games*: GameReal, GameFake and GameHybrid. GameReal is a rewrite of IND-CPA game from Figure 5.7, GameFake is identical to GameReal except that it replaces every entry that may not be opened by visible tokens by truly random bitstrings and it is independent from the decision bit b (i.e., the adversary has no advantage) and GameHybrid, parametrised by a value l, captures the steps needed to go from GameReal to GameFake. Therefore, GameHybrid parametrised by $l = 0$ is equivalent to GameReal and parametrised by $l = n + q - m$ is equivalent to GameFake. The distance between GameHybrid when parametrised with a value l and a value $l + 1$ will then be bounded by building a distinguishing attacker that breaks the underlying DKC scheme, thereby also bounding the distance between the two extreme cases $l = 0$ and $l = n + q - m$.

All of these games operate in the same way: i. first it goes through all the wires and, for each one of them, generates a *toggle* for that wire (a bit) and two tokens (one *visible* and one *invisible*); ii. later, it runs across all the gates of the circuit and garbles each entry using the underlying DKC primitive. We define three EasyCrypt modules that will store the values of the random generated tokens, of the circuit and of the garbling process.

TOKEN GENERATION. The random tokens, that will subsequently be used to encode the inputs, are generated in two ways, one that will be used when in GameReal, GameFake and GameHybrid

```
module R = {
  var t : bool array
  var xx : tokens_t
}.

module C = {
  var f:fun_t
  var x:input_t
  var n:int
  var m:int
  var q:int
  var aa:int array
  var bb:int array
  var gg:bool gates_t
  var v:bool array
}.

module G = {
  var pp:word gates_t
  var yy:word array
  var randG: word gates_t
  var a:int
  var b:int
  var g:int
}.
```

Figure 5.8: Global values.

and another one that will be used by the adversary attacking the DKC security of the DKC primitive.

Regarding GameReal, GameFake and GameHybrid, one first chooses the *toggle trnd* of the token, that will correspond to its least significant bit. This value will either be random (if the wire is an input wire to any gate) or equivalent to the evaluation at the output of that gate (if the wire is an output wire). Then, two tokens are generated, a *visible* one (that the adversary will have access to) and an *invisible* one that is hidden from the adversary. The random generation module formalisation can be found in Figure 5.9, where Dword.dwordLsb is a distribution over bitstrings of fixed LSB and useVisible is a Boolean value that controls wether one should use the values of the evaluation of the circuit.

In what concerns the DKC adversary, the two tokens are generated in a simpler, *ad-hoc* way: for every value of $i \in [0..\text{bound}[$, then one token will be generated with LSB 0 and another will be generated witht LSB 1. We proved that both constructions produce equivalent tokens.

Real Game. GameReal is represented in Figure 5.11, where ^^ denotes the XOR operation. It makes use of two auxiliary functions garb and garb′ that are used to garble the entries of the truth table using the labels generated (Figure 5.10).

This proof is carried out by instantiating game IND-CPA with SomeGarble and then proving the equivalence between the two games. We note that game IND-CPA is instantiated with SomeGarble using a different token generator procedure of the type Rand as presented in Figure 5.12. This random generator can be seen as a specific case of the previous one when useVisible is false. The reason for this is that the garbling scheme construction is used in its pure form in the IND-CPA game, which means that wire values are encoded into the gates during the garbling procedure, and randomness generation is totally oblivious of wire values.

The intuition behind the proof is that the tokens generated in both experiments are identically distributed, despite the fact that they are generated differently. Consequently, the adversary will have the same information in both procedures.

Fake Game output is independent of the challenge bit. The independence proof for the output of GameFake was carried on in two steps, as suggested in [5]: i. rewrite GameFake

```
module RandomInit = {
  proc init(useVisible:bool): unit = {
    ...

    R.t = offun (fun x, false) (C.n + C.q);
    R.xx = map0;
    i = 0;
    while (i < C.n + C.q) {
      trnd = ${0,1};
      v = if useVisible then C.v[i] else false;
      trnd = if (i < C.n + C.q − C.m) then trnd else v;
      tok1 = $Dword.dwordLsb ( trnd);
      tok2 = $Dword.dwordLsb (!trnd);

      R.t[i] = trnd;

      R.xx[(i, v)] = tok1;
      R.xx[(i,!v)] = tok2;

      i = i + 1;
    }
  }
}.
```

Figure 5.9: Random generation module.

```
proc garb(yy : word, alpha : bool, bet : bool) : unit = {
  ...

  twe = tweak G.g (R.t[G.a] ^^ alpha) (R.t[G.b] ^^ bet);
  aa = oget R.xx[(G.a, C.v[G.a] ^^ alpha)];
  bb = oget R.xx[(G.b, C.v[G.b] ^^ bet)];
  G.pp[(G.g, R.t[G.a] ^^ alpha, R.t[G.b] ^^ bet)] = E twe aa bb yy;
}

proc garb'(rn : bool, alpha : bool, bet : bool) : word = {
  ...

  yy = $Dword.dword;
  yy = if rn then yy else oget R.xx[(G.g, oget C.gg[(G.g, C.v[G.a] ^^ alpha, C.v[G.b] ^^ bet)])];
  garb(yy, alpha, bet);
  return yy;
}
```

Figure 5.10: Procedures garb and garb′.

```
module GarbleRealInit = {
  proc init() : unit = {

    ...

    G.g = C.n;
    while (G.g < C.n + C.q)
    {
      G.a = C.aa[G.g];
      G.b = C.bb[G.g];

      garb(oget R.xx[(G.g, C.v[G.g])], false, false);

      garb'(false, true, false);
      garb'(false, false, true);
      G.yy[G.g] = garb'(false, true, true);

      G.g = G.g + 1;
    }
  }
}.
```

Figure 5.11: Game GameReal.

```
module Rand : Rand = {
  proc gen(l:topo_t) : tokens_t = {
    var n, m, q, i : int;
    var aa, bb : int array;
    var t : bool;

    (n,m,q,aa,bb) = l;

    R.t = offun (fun x, false) (n+q);
    R.xx = map0;
    i = 0;
    while (i < n+q) {
      t = ${0,1};
      t = if (i < n+q−m) then t else false;
      R.t[i] = t;
      R.xx[(i,false)] = $Dword.dwordLsb t;
      R.xx[(i,true)] = $Dword.dwordLsb (!t);
      i = i+1;
    }

    return R.xx;
  }
}.
```

Figure 5.12: Random generator to use in the instantiation of game IND-CPA.

as a new game GameFake' in which proving the independence was trivial; and ii. proving the equivalence between GameFake and GameFake'. We omit the formalisation of GameFake because it is similar to the one of GameReal, with the exception of the call to garb' being made with the first parameter as *true*. GameFake and GameFake' differ in two main aspects:

- In GameFake', we consider two maps R'.vv and R'.ii instead of a simple map R.xx. They will store the visible and invisible tokens, respectively (Figure 5.13).
- GameFake' does not make any call to garb or garb', as it explicitly executes a similar code during the garbling process (Figure 5.14).

```
module RandomInit' = {
  proc init(useVisible:bool): unit = {
    ...

    i = 0;
    while (i < C.n + C.q) {
      trnd = ${0,1};
      v = if useVisible then C.v[i] else false;
      trnd = if (i < C.n + C.q − C.m) then trnd else v;
      tok1 = $Dword.dwordLsb ( trnd);
      tok2 = $Dword.dwordLsb (!trnd);

      R'.t[i] = trnd;

      R'.vv[i] = tok1;
      R'.ii[i] = tok2;

      i = i + 1;
    }
  }
}.
```

Figure 5.13: Random generator of GameFake'.

To prove the equivalence between the two games, we showed that:

- The visible values of map R.xx in GameFake are equivalent to the values of map R'.vv in GameFake'.

```
module GarbleInitFake' = {

  proc init() : unit = {
    ...

    G.g = C.n;
    while (G.g < C.n + C.q) {
      G.a = C.aa[G.g];
      G.b = C.bb[G.g];

      wa = oget R'.vv[G.a];
      wb = oget R'.vv[G.b];
      tok = oget R'.vv[G.g];
      twe = tweak G.g (getlsb wa) (getlsb wb);
      G.pp[(G.g, getlsb wa, getlsb wb)] = E twe wa wb tok;

      wa = oget R'.ii[G.a];
      wb = oget R'.vv[G.b];
      tok = $Dword.dword;
      twe = tweak G.g (getlsb wa) (getlsb wb);
      G.pp[(G.g, getlsb wa, getlsb wb)] = E twe wa wb tok;

      wa = oget R'.vv[G.a];
      wb = oget R'.ii[G.b];
      tok = $Dword.dword;
      twe = tweak G.g (getlsb wa) (getlsb wb);
      G.pp[(G.g, getlsb wa, getlsb wb)] = E twe wa wb tok;

      wa = oget R'.ii[G.a];
      wb = oget R'.ii[G.b];
      tok = $Dword.dword;
      twe = tweak G.g (getlsb wa) (getlsb wb);
      G.pp[(G.g, getlsb wa, getlsb wb)] = E twe wa wb tok;

      G.yy[G.g] = tok;

      G.g = G.g + 1;
    }
  }
}.
```

Figure 5.14: Garbling procedure of GameFake'.

- Similarly, the invisible values of map R.xx in GameFake are equivalent to the values of map R'.vv in GameFake'.
- Every entry of the truth table will be garbled exactly the same way in both experiments, as the inlining of procedures garb and garb' will result in a similar code to the one of GameFake'.

Independence of GameFake' was easily proved considering the fact that the output of it does not depend on any decision bit.

HYBRID GAME BOUNDS. In order to complete the proof of security of the garbling scheme, there must be a bound on the distance between GameReal (identical to the IND-CPA game) and GameFake, where the adversary has no advantage. To relate the two experiments, Bellare et al [5] defined a hybrid argument represented by GameHybrid (Figure 5.15), parametrised by some hybrid value l.

```
1  module GarbleHybridInit = {
2    proc init(l : int) : unit = {
3
4      ...
5
6      G.g = C.n;
7      while (G.g < C.n + C.q) {
8        G.a = C.aa[G.g];
9        G.b = C.bb[G.g];
10
11       garb(oget R.xx[(G.g, C.v[G.g])], false, false);
12
13       tok = garb'(G.a ≤l, true, false);
14       tok = garb'(G.b ≤l, false, true);
15       G.yy[G.g] = garb'(G.a ≤l, true, true);
16
17       if (G.a ≤l < G.b ∧ C.gg[(G.g, !C.v[G.a], false)] = C.gg[(G.g, !C.v[G.a], true)]) {
18         garb(G.yy[G.g], true, false);
19       }
20
21       G.g = G.g + 1;
22     }
23   }
24 }.
```

Figure 5.15: Game GameHybrid.

The idea behind the hybrid argument is the following. In GameReal, every entry in the garbled row is filled with the encryption of the corresponding token. In GameFake, every hidden entry in the garble row is filled with a truly random bitstring. For $l \in \{0..n+q-m-1\}$, GameHybrid will iteratively replace the correct value of the garbled gate by truly random strings. Therefore, one can establish a *path* for going from the reference game IND-CPA (GameReal) to an independent game (GameFake).

When $l = 0$, it is easy to establish that GameHybrid and GameReal are equivalent, since all the Boolean values G.a $\leq$l and G.b $\leq$l are false. Likewise, GameHybrid and GameFake are equivalent, since all the Boolean values G.a $\leq$l and G.b $\leq$l are false.

The proof is completed by reducing the distance between two consecutive hybrids to the security of the underlying Dual Key Cypher scheme. The reduction is made by proving the following two results

$$\Pr\left[\mathsf{DKC}^{D} \mid b = 1 : \mathsf{res}\right] = \frac{1}{n+q} \sum_{l=1}^{n+q-m} \Pr\left[\mathsf{GameHybrid}_{l-1}^{\mathbf{Adv}} : \mathsf{res}\right]$$

$$\Pr\left[\neg\mathsf{DKC}^{D} \mid b = 0 : \mathsf{res}\right] = \frac{1}{n+q} \sum_{l=1}^{n+q-m} \Pr\left[\neg\mathsf{GameHybrid}_{l}^{\mathbf{Adv}} : \mathsf{res}\right]$$

We adopt a slightly different, yet sufficient for our instantiation, security model from the one described in [5]. As already mentioned, we consider a different token generation procedure for the DKC security (Figure 5.18). The DKC adversary has access to an oracle Encrypt that it will use to garble the Boolean circuit. The oracle is given the indices of the entry of the gate being garbled and it returns with a valid garbled value to this entry. This garbled value can be either the encryption of a token or the encryption of truly random value. Informally, the adversary should not be able to distinguish between these two values. The formalisation of this security definition in EasyCrypt can be found in Figures 5.16 and 5.17.

```
module Game(D:DKC_t, A:Adv_DKC_t) = {

  proc game(b : bool) : bool = {
    ...
    lsb = D.initialize(b);
    b' = A.get_challenge(lsb);
    return b' = b;
  }

  proc main() : bool = {
    var adv : bool;
    var b : bool;
    b = ${0,1};
    adv = game(b);
    return adv;
  }
}
```

Figure 5.16: DKC security experiment.

```
module DKC : DKC_t = {
  ...
  proc encrypt(q:query_DKC) : answer_DKC = {
    ...
    ans = bad;
    (rn,ib,jb,lb,t) = q;

    if (!(mem DKCp.used t) ∧ ib.`1 < jb.`1 ∧ jb.`1 < lb.`1 ∧ lb ≠(l,DKCp.lsb)) {
      DKCp.used = DKCp.used `|` fset1 t;

      ki = oget DKCp.kpub[ib];
      kj = oget DKCp.kpub[jb];

      (aa,bb) = if (ib = (l,DKCp.lsb))
                  then (DKCp.ksec, kj)
                  else (if (jb = (l,DKCp.lsb))
                          then (ki, DKCp.ksec)
                          else (ki,kj));

      xx = oget DKCp.kpub[lb];

      if (((((l,DKCp.lsb) = ib) || ((l,DKCp.lsb) = jb)) ∧ !DKCp.b) || rn) {
        xx = $Dword.dword;
      }
      ans = (ki, kj, E t aa bb xx);
    }
    return ans;
  }
}
```

Figure 5.17: Oracle encrypt.

```
module DKC : DKC_t = {
  ...
  proc initialize(b : bool): bool = {
    ...
    while (i < bound) {
      if (i = l) {
        DKCp.lsb = ${0,1};
        DKCp.ksec = $Dword.dwordLsb (DKCp.lsb);
        DKCp.kpub[(i,DKCp.lsb)] = witness; (* can never return or encrypt this key *)
        DKCp.kpub[(i,!DKCp.lsb)] = $Dword.dwordLsb (!DKCp.lsb);
      }
      else {
        tok1 = $Dword.dwordLsb (false);
        tok2 = $Dword.dwordLsb (true);
        DKCp.kpub[(i, false)] = tok1;
        DKCp.kpub[(i, true)] = tok2;
      }
      i = i + 1;
    }

    return DKCp.lsb;
  }
}
```

Figure 5.18: Tokens generation.

The adversary queries Encrypt in the following way:

1. If the indices of both the left and right wires (a and b, respectively) of the gate being garbled are greater than l, then the gate should be garbled using a valid token.

2. On the opposite, if both indexes are smaller than l, then the gate should be garbled using a truly random word.

3. If l is greater than a but smaller than b, then only the entries corresponding to $(1, 0)$ and $(1, 1)$ should be garbled with a truly random value, whereas the other rows should be garbled with a valid token.

The reduction proof is completed by proving that, when $\mathsf{DKCp.b} = 1$, the DKC adversary will simulate $\mathsf{GameHybrid}_{l-1}$ and will simulate $\mathsf{GameHybrid}_l$ otherwise. The complete EasyCrypt code of the adversary is ommitted for brevity.

5.2.1.5 A concrete OT protocol: SomeOT

To instantiate the oblivious transfer protocol of the SFE theory, we have adopted an n-fold extension of the protocol by Bellare and Micali [6], in the hashed version presented by Naor and Pinkas [33] (n being the size of the selection string). The protocol requires a cyclic group $\mathcal{G}$ of prime order q and a generator g, and operates as described in Figure 5.19. The protocol is described in a purely functional manner, making any local state shared between the various stages of a given party explicit. For example, step_1 outputs the sender's local state st_s, for later use by step_3.

The security proof in the standard model is done via a reduction to the decisional Diffie-Hellman assumption and an entropy-smoothing assumption on the hash function.

We let $\mathsf{Adv}^{\mathsf{DDH}}(\mathbf{Adv})$ and $\mathsf{Adv}^{\mathsf{ES}}(\mathbf{Adv})$ be the advantage of an adversary **Adv** breaking the DDH and the Entropy Smoothing assumptions, respectively.

```
op step1 (m : (msg * msg) array) (r:int array * G) =
  let (c, hkey) = r in
  let st_s = (m, g^c,hkey) in
  let m_1 = (hkey,g^c) in
  (st_s,m_1).

op step2 (b : bool array) (r:G array) m_1 =
  let (hkey,gc) = m_1 in
  let st_c = (b,hkey,r) in
  let m_2 = if b then gc / g^r else g^r in
  (st_c,m_2).

op step3 st_s (r:G) m_2 =
  let (m,gc,hkey) = st_s in
  let e = (H(hkey, m_2^r) ⊕ m_0,H(hkey,(gc/ m_2)^r) ⊕ m_1) in
  let m_3 = (g^r,e) in
  m_3.

op finalize st_c m_3 =
  let (b,hkey,x) = st_c in
  let (gr,e) = m_3 in
  let res = H(hkey,gr^x) ⊕ e_b in
  res.

clone OTProt as SomeOT with
  type rand_1 = G array,
  type rand_2 = (G array * G) * G,
  op prot (b:input_1) (r_c:rand_1) (m:input_2) (r_s:rand_2) =
    let (st_s,m_1) = step1 m (fst r_s) in
    let (st_c,m_2) = step2 b r_c m_1 in
    let m_3 = step3 st_s (snd r_s) m_2 in
    let res = finalize st_c m_3 in
    let conv = (m_1,m_2,m_3) in
    (conv,(res,())).
```

Figure 5.19: Our Concrete Oblivious Transfer Protocol.

Theorem 3 (OT-security of SomeOT) *For all $i \in \{1,2\}$ and OT^i adversary $\mathbf{Adv}_i$ against the SomeOT protocol, we can construct two efficient adversaries $\mathcal{D}^{\mathsf{DDH}}$ and $\mathcal{D}^{\mathsf{ES}}$, and a efficient simulator S such that*

$$\mathsf{Adv}^{\mathsf{OT}^i}_{\mathsf{SomeOT},\mathsf{S}}(\mathbf{Adv}_i) \leq \mathsf{n} \cdot \mathsf{Adv}^{\mathsf{DDH}}(\mathcal{D}^{\mathsf{DDH}}) + \mathsf{n} \cdot \mathsf{Adv}^{\mathsf{ES}}(\mathcal{D}^{\mathsf{ES}}).$$

The protocol is proven secure against malicious senders ($i = 1$) and malicious choosers ($i = 2$).

5.2.1.6 A concrete SFE protocol: Concrete

Finally, we combine SomeOT and SomeGarble using Yao's construction to obtain our Concrete SFE functionality. The security proof for this concrete construction immediately follows from Theorems 1, 3 and 2.

However, we take this opportunity to implement some instantiation-specific optimizations across abstraction boundaries and translate high-level programming constructs like maps and higher-order functions into more efficient data structures such as arrays. A separate proof that our efficient implementation is perfectly equivalent to the one on which we performed the security proof yields the final security theorem.

Theorem 4 (Security of the concrete SFE protocol) *For all SFE adversary $\mathbf{Adv}$ against the Concrete SFE protocol, we construct an efficient simulator S and efficient adversaries $\mathcal{B}_{\mathsf{DKC}}$,*

$\mathcal{B}_{\mathsf{DDH}}$ *and* $\mathcal{B}_{\mathsf{ES}}$*, such that the following inequalities hold:*

$$\mathsf{Adv}^{\mathsf{SFE}^1_{\Psi_{\mathsf{topo}}}}_{\mathsf{Concrete},\mathsf{S}}(\mathbf{Adv}) \leq (\mathsf{c}+1) \cdot \mathsf{Adv}^{\mathsf{DKC}}(\mathcal{B}_{\mathsf{DKC}}) + \varepsilon,$$
$$\mathsf{Adv}^{\mathsf{SFE}^2_{\Psi_{\mathsf{topo}}}}_{\mathsf{Concrete},\mathsf{S}}(\mathbf{Adv}) \leq \varepsilon,$$

where $\varepsilon = \mathsf{n} \cdot \mathsf{Adv}^{\mathsf{DDH}}(\mathcal{B}_{\mathsf{DDH}}) + \mathsf{n} \cdot \mathsf{Adv}^{\mathsf{ES}}(\mathcal{B}_{\mathsf{ES}})$.

5.2.2 Extracting an implementation

The EasyCrypt toolset uses the Why3 platform as a background interface with SMT solvers. Consequently, having access to the generated Why3 proof tasks, we were able to use the extraction mechanism of Why3 in order to obtain a verified OCaml implementation from our SFE protocol specification.

A Why3 proof task is extracted at the end of the definition of our Concrete SFE protocol. The task will contemplate every definition used by Concrete, from the previously defined characterisations of a projective garbling scheme and of an oblivious transfer protocol to the definition of EasyCrypt native types, like the integer type, resulting in a complex Why3 script.

The code extraction mechanism of Why3 uses some pre-defined libraries, such as the integer library and the real library. Therefore, every definition regarding these two types that was produced by the generation of the Why3 task needed to be syntactically changed to match the Why3 types. Another important aspect is that Why3 does not extract abstract operators, like, for example, our definition of a cyclic algebraic structure, dual-key cipher encryption scheme or entropy smoothing hash functions. This forced us to extract the code in a manual way, with minor syntactic adjustments, and then complete the implementation on the top of the `CryptoKit` library.

The OCaml implementation files are the following:

- *Cyclic_group_prime.ml* - abstract operations over a cyclic group of prime order q, in which the oblivious transfer relies on (instantiated using a multiplicative subgroup of integers modulo a large prime).
- *DKC.ml* - implementation of a dual-key cipher encryption scheme based on an abstract implementation of a pseudorandom function (instantiated using AES-128).
- *Hash.ml* - implementation of the SHA256 hash function.
- *Prime_field.ml* - implementation of a prime field corresponding to arithmetic modulo a prime, instantiated using q, used for randomness generation in the OT protocol.
- *Word.ml* - implementation of the operations over bit strings of a fixed length.
- *SFE.ml* - implementation of the concrete definition of the SFE in EasyCrypt. This file represents an OCaml implementation of a garbling scheme, an oblivious transfer protocol and a combination of both that constructs a SFE protocol.

Additionally, we produced files that represent OCaml instantiations for the EasyCrypt native types:

- *ecBool.ml* - OCaml interface for the EasyCrypt Boolean theory.

Table 5.1: Execution times (milliseconds)

Circuit	NGates	TTime	P2 S1 GT	P2 S1 OT	P1 S1 OT	P2 S2 OT	P1 S2 OT	P1 S2 ET
COMP32	301	299	1	60	59	121	59	1
ADD32	408	346	2	70	70	138	68	1
ADD64	824	692	4	141	139	277	135	1
MUL32	12438	376	55	115	61	126	61	13
AES	33744	1333	137	370	234	468	232	29
SHA1	106761	2993	484	963	479	967	482	101

- *ecIArray.ml* - OCaml interface for the EasyCrypt array theory.
- *ecInt.ml* - OCaml interface for the EasyCrypt integer theory.
- *ecPair.ml* - OCaml interface for the EasyCrypt pair theory.
- *ecPervasives.ml* - OCaml interface for the EasyCrypt pervasives theory.

5.3 Performance

In this section we present a performance evaluation of the SFE implementation generated from the EasyCrypt formalisation. Inputs are generated randomly using OCaml's Rand module, and the cryptographic randomness is generated using `CryptoKit`'s RNG. Our results show that, whilst being slower than optimised implementations of SFE [26, 4], the performance of the *extracted* program is compatible with real-world deployment, providing some evidence that the (unavoidable) overhead implied by our formal verification and code extraction approach is not prohibitive. We now present our experimental results in detail.

In addition to the overall execution time of the SFE protocol and the splitting of the processing load between the two involved parties, we also measure various speed parameters that permit determining the weight of the underlying components: the time spent in executing the OT protocol, and the garbling and evaluation speeds for the garbling scheme. Our measured execution times do not include serialisation and communication overheads (which are out of the scope of this work), nor do they include the time to sample the randomness (which can be pre-generated).

We run our experiments on an x86-64 Intel Core 2 Duo clocked at a modest 1.86 GHz with a 6MB L2 cache. The extracted code and parser are compiled with ocamlopt version 4.00.1. The tests are run in isolation, using the OCamlSys.time operator for time readings. We run tests in batches of 100 runs each, noting the median of the times recorded in the runs.

Our measurements are conducted over circuits made publicly available by the cryptography group at the University of Bristol[1], precisely for the purpose of enabling the testing and benchmarking of multiparty computation and homomorphic encryption implementations. A simple conversion of the circuit format is carried out to ensure that the representation matches the conventions adopted in the formalisation.

A subset of our results are presented in Table 5.1, for circuits COMP32 (32-bit signed number less-than comparison), ADD32 (32-bit number addition), ADD64 (64-bit number addition), MUL32 (32-bit number multiplication), AES (AES block cipher) and SHA1 (SHA-1 hash algorithm), with respect to the garbling (S1 GT), the two stages of oblivious transfer (S1/S2 OT) and the evaluation stage (S2 ET). The semantics of the evaluation of the arithmetic circuits is that each party holds one of the operands. In the AES evaluation we have that P1 holds the

[1] `http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/`

128-bit input block, whereas P2 holds the 128-bit secret key. Finally, in the SHA1 example we model the (perhaps artificial) scenario where each party holds half of a 512-bit input string.

We present the number of gates for each circuit as well as the execution times in milliseconds. A rough comparison with results presented in, for example [26], where an execution of the AES circuit takes roughly 1.6 seconds (albeit including communications overhead and randomness generation time) allows us to conclude that real-world applications are within the reach of the implementations generated using the formally verification approach that we are exploring within PRACTICE. Furthermore, additional optimisation effort can lead to significant performance gains, e.g., by resorting to hardware support for low-level cryptographic implementations as in [4], or implementing garbled-circuit optimisations such as those allowed by XOR gates [30].

Chapter 6

Conclusion

In this report we have described four implementations of protocol for secure computation. First we described an optimized protocol for generating Beaver triples, which is a subprotocol that are used in many secure computation protocols. Then we presented a two-party protocol for securely computing boolean circuits, called TinyTables. Then we presented the mixed protocol of the ABY framework which allows an application to switch between different secure computation protocols in order to optimize performance. Finally, we presented a formally verified implementation of Yao's garbled circuits, in particular the efforts to obtain a mechanised proof of the security and correctness of the protocol.

For each protocol we have first given a short theoretical overview of the protocol, and provided details on how it has been implemented. Then we have measured the performance of the protocols with regard to timing and amount of network communications, and we have for each protocol discussed bottlenecks and possible optimizations. The results of the measurements can be found in the individual chapters, but they include that the protocol for generating Beaver triples can generate 100,000 32-bit triples in 15.45 seconds, and that the ABY framework can compute private set intersection on 4096 elements of length 32-bits in 30.0 seconds. Both these measurements have been done are in a setting with an average network latency of 170 ms.

Based on the quality of the implementations and the performance measurements we have conducted, we belive that all the presented protocol implementations are feasible to use by software developers in building applications based on secure computation.

Bibliography

[1] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 535–548, 2013.

[2] Elaine B. Barker and John M. Kelsey. Sp 800-90a. recommendation for random number generation using deterministic random bit generators. Technical report, Gaithersburg, MD, United States, 2012.

[3] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella-Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 71–90, Heidelberg, 2011. Springer.

[4] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy (S&P'13)*, pages 478–492. IEEE, 2013.

[5] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 784–796, New York, NY, USA, 2012. ACM.

[6] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO' 89 Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 547–557. Springer New York, 1990.

[7] Pille Pullonen Florian Kerschbaum Florian Hahn Agnes Kiss Thomas Schneider Michael Zohner Benny Pinkas, Claudio Orlandi. PRACTICE Deliverable D13.3: the full set of new protocols, 2016.

[8] Thomas Schneider Michael Zohner Benny Pinkas, Agnes Kiss. PRACTICE Deliverable D13.4: prototype implementations of key protocols, 2016.

[9] Dan Bogdanov. *Sharemind: programmable secure computations with practical applications*. PhD thesis, University of Tartu, 2013.

[10] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the Estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, volume 8975 of *LNCS*, pages 227–234. Springer, 2015.

[11] Dan Bogdanov, Marko J oemets, Sander Siim, and Meril Vaht. Privacy-preserving tax fraud detection in the cloud with realistic data volumes. Technical Report T-4-24, Cybernetica AS, `http://research.cyber.ee/`., 2016.

[12] Niklas Buescher, Peter Nordholt Dan Bogdanov, Roman Jagomägis, Jaak Randmets, José Bacelar Almeida, Bernardo Portela, and Hugo Pacheco. PRACTICE Deliverable D12.3: formal verification requirements, 2015. Available from `http://www.practice-project.eu`.

[13] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145, 2001.

[14] I. Damgård, M. Geisler, and M. Krøigaard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 1(1):22–31, 2008.

[15] I. Damgård, M. Geisler, and M. Krøigaard. A correction to 'Efficient and secure comparison for on-line auctions'. *International Journal of Applied Cryptography*, 1(4):323–324, 2009.

[16] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In *PKC'01*, volume 1992, pages 119–136, 2001.

[17] I. Damgård, M. Jurik, and J. B. Nielsen. A generalization of Paillier's public-key system with applications to electronic voting. *International Journal of Information Security*, 9(6):371–385, 2010.

[18] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samiel Ranellucci. Gate-scrambling revisited - or: The tinytable protocol for 2-party secure computation. 2016.

[19] Kasper Damgård, Peter Nordholt, Marko Jõemets, Peeter Laud, Sander Siim, Ville Sokk, Sander Valvas, Kurt Nielsen, and Tomas Toft. PRACTICE Deliverable D23.3: an online portal providing secure computation capabilities, 2016.

[20] Kasper Lyneborg Damgård, Thomas Jakobsen, and Peter Sebastian Nordholdt. PRACTICE Deliverable D14.2: platform for secure computation, 2013. Available from `http://www.practice-project.eu`.

[21] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security (NDSS'15)*. The Internet Society, 2015. Code: `http://encrypto.de/code/ABY`.

[22] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, C-21(7):801–803, July 1972.

[23] Niv Gilboa. Two party RSA key generation. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 116–129, 1999.

[24] Shay Gueron, Yehuda Lindell, Ariel Nof, and Benny Pinkas. Fast garbling of circuits under standard assumptions. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 567–578, 2015.

[25] Isabelle Hang, Ferdinand Brasser, Niklas Buescher, Stefan Katzenbeisser, Ahmad Sadeghi, Kai Samelin, Thomas Schneider, Jakob Pagter, Peter Sebastian Nordholt Janus Dam Nielson, Kurt Nielsen, Johannes Ulfkjaer Jensen, Dan Bogdanov, Roman Jagomägis, Liina Kamm, Jaak Randmets, Jaak Ristioja, Reimo Rebane, Jaak Ristioja, Sander Siim, Riivo Talviste, Manuel Barbosa, Bernardo Portela, Rui Oliveira, Stelvio Cimato, and Ernesto Damiani. PRACTICE Deliverable D22.1: tools: State-of-the-art analysis, 2013. Available from `http://www.practice-project.eu`.

[26] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[27] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, pages 145–161, 2003.

[28] Florian Kerschbaum, Florian Hahn, Thomas Schneider, Michael Zohner, Pille Pullonen, and Claudio Orlandi. PRACTICE Deliverable D13.1: a set of new protocols, 2015. Available from `http://www.practice-project.eu`.

[29] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, pages 54–70, 2013.

[30] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, Part II*, ICALP '08, pages 486–498, Berlin, Heidelberg, 2008. Springer-Verlag.

[31] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

[32] Tobias Mueller, Niklas Buescher, Hiva Mahmoodi, Janus Dam Nielsen, Peter S. Nordholt, Dan Bogdanov, Manuel Barbosa, Johannes U Jensen, and Kurt Nielsen. PRACTICE Deliverable D22.2: Tools design document, 2014.

[33] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 448–457, 2001.

[34] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT'99*, volume 1592, pages 223–238, 1999.

[35] Reimo Rebane. PRACTICE Deliverable D22.4: software development kit and tools prototype (final version), 2016.

[36] Sander Siim. A Comprehensive Protocol Suite for Secure Two-Party Computation. Master's thesis, Institute of Computer Science, University of Tartu, 2016.

[37] Why3 user manual. `http://why3.lri.fr/download/manual-0.81.pdf`.