More Efficient Oblivious Transfer and Extensions for Faster Secure Computation*

Gilad Asharov[1], Yehuda Lindell[1], Thomas Schneider[2], and Michael Zohner[2]

[1] Cryptography Research Group,
Bar-Ilan University, Israel,
asharog@cs.biu.ac.il, lindell@biu.ac.il
[2] Engineering Cryptographic Protocols Group,
TU Darmstadt, Germany,
{thomas.schneider,michael.zohner}@ec-spride.de

Abstract. Protocols for secure computation enable parties to compute a joint function on their private inputs without revealing anything but the result. A foundation for secure computation is oblivious transfer (OT), which traditionally requires expensive public key cryptography. A more efficient way to perform many OTs is to extend a small number of base OTs using OT extensions based on symmetric cryptography.
In this work we present optimizations and efficient implementations of OT and OT extensions in the semi-honest model. We propose a novel OT protocol with security in the standard model and improve OT extensions with respect to communication complexity, computation complexity, and scalability. We also provide specific optimizations of OT extensions that are tailored to the secure computation protocols of Yao and Goldreich-Micali-Wigderson and reduce the communication complexity even further. We experimentally verify the efficiency gains of our protocols and optimizations. By applying our implementation to current secure computation frameworks, we can securely compute a Levenshtein distance circuit with 1.29 billion AND gates at a rate of 1.2 million AND gates per second. Moreover, we demonstrate the importance of correctly implementing OT within secure computation protocols by presenting an attack on the FastGC framework.

Keywords: Secure computation; oblivious transfer extensions; semi-honest adversaries

1 Introduction

1.1 Background

In the setting of secure two-party computation, two parties P_0 and P_1 with respective inputs x and y wish to compute a joint function f on their inputs

*A preliminary version of this paper appears at ACM CCS 2013 [1].

without revealing anything but the output $f(x, y)$. This captures a large variety of tasks, including privacy-preserving data mining, anonymous transactions, private database search, and many more. In this paper, we consider semi-honest adversaries who follow the protocol, but may attempt to learn more than allowed via the protocol communication. We focus on semi-honest security as this allows construction of highly efficient protocols for many application scenarios. This model is justified e.g., for computations between hospitals or companies that trust each other but need to run a secure protocol because of legal restrictions and/or in order to prevent inadvertent leakage (since only the output is revealed from the communication). Semi-honest security also protects against potential misuse by some insiders and future break-ins, and can be enforced with software attestation. Moreover, understanding the cost of semi-honest security is an important stepping stone to efficient malicious security. We remark that also in a large IARPA funded project on secure computation on big data, IARPA stated that the semi-honest adversary model is suitable for their applications [31].

Practical secure computation. Secure computation has been studied since the mid 1980s, when powerful feasibility results demonstrated that any efficient function can be computed securely [19, 58]. However, until recently, the bulk of research on secure computation was theoretical in nature. Indeed, many held the opinion that secure computation will never be practical since carrying out cryptographic operations for every gate in a circuit computing the function (which is the way many protocols work) will never be fast enough to be of use. Due to many works that pushed secure computation further towards practical applications, e.g., [6, 7, 10, 15, 17, 25, 28, 35, 41–43, 50, 57], this conjecture has proven to be wrong and it is possible to carry out secure computation of complex functions at speeds that five years ago would have been unconceivable. For example, in FastGC [28] it was shown that AES can be securely computed with 0.2 seconds of preprocessing time and just 0.008 seconds of online computation. This has applications to private database search and also to mitigating server breaches in the cloud by sharing the decryption key for sensitive data between two servers and never revealing it (thereby forcing an attacker to compromise the security of two servers instead of one). In addition, [28] carried out a secure computation of a circuit of size *1.29 billion AND gates*, which until recently would have been thought impossible. Their computation took 223 minutes, which is arguably too long for most applications. However, it demonstrated that large-scale secure computation can be achieved. The FastGC framework was a breakthrough result regarding the practicality of secure computation and has been used in many subsequent works, e.g., [26, 27, 29, 30, 50]. However, it is possible to still do much better. The secure computation framework of [56] improved the results of FastGC [28] by a factor of 6-80, depending on the network latency. Jumping ahead, we obtain additional speedups for both secure computation frameworks [28] and [56]. Most notably, when applying our improved OT implementation to the framework of [56], we are able to evaluate the 1.29 billion AND gate circuit in just 18 minutes. We conclude that significant efficiency improvements can still be made, considerably broadening the tasks that can be solved using secure computation in practice.

Oblivious transfer and extensions. In an *oblivious transfer (OT)* [54], a sender with a pair of input strings (x_0, x_1) interacts with a receiver who inputs a choice bit σ. The result is that the receiver learns x_σ without learning anything about $x_{1-\sigma}$, while the sender learns nothing about σ. Oblivious transfer is an extremely powerful tool and the foundation for almost all efficient protocols for secure computation. Notably, Yao's garbled-circuit protocol [58] (e.g., implemented in FastGC [28]) requires OT for every input bit of one party, and the GMW protocol [19] (e.g., implemented in [10, 56]) requires OT for every AND gate of the circuit. Accordingly, the efficient instantiation of OT is of crucial importance as is evident in many recent works that focus on efficiency, e.g., [10, 20, 23, 26–28, 30, 40, 43, 49, 56]. In the semi-honest case, the best known OT protocol is that of [46], which has a cost of approximately 3 exponentiations per 1-out-of-2 OT. However, if thousands, millions or even billions of oblivious transfers need to be carried out, this will become prohibitively expensive. In order to solve this problem, *OT extensions* [4, 32] can be used. An OT extension protocol works by running a small number of OTs (say, 80 or 128) that are used as a base for obtaining many OTs via the use of cheap symmetric cryptographic operations only. This is conceptually similar to public-key encryption where instead of encrypting a large message using RSA, which would be too expensive, a hybrid encryption scheme is used such that only a single RSA computation is carried out to encrypt a symmetric key and then the long message is encrypted using symmetric operations only. Such an OT extension can actually be achieved with extraordinary efficiency; specifically, the protocol of [32] requires only three hash function computations on a single block per oblivious transfer (beyond the initial base OTs).

Related Work. There is independent work on the efficiency of OT extension with security against stronger malicious adversaries [21, 48, 49]. In the semi-honest model, [24] improved the implementation of the OT extension protocol of [32] in FastGC [28]. They reduce the memory footprint by splitting the OT extension protocol sequentially into multiple rounds and obtain speedups by instantiating the pseudo-random generator with AES instead of SHA-1. Their implementation evaluates 400,000 OTs (of 80-bit strings without precomputations) per second over WiFi; we propose additional optimizations and our fastest implementation evaluates more than 700,000 OTs per second over WiFi, cf. Tab. 4. A recent contribution [36] focuses on efficient OT extensions on short strings and achieves sublinear communication in the number of OTs. Our work is orthogonal to theirs, since our OT protocols maintain their efficiency when obliviously transferring long strings. Finally, [12] presented a protocol for large-scale private set-intersection using 1-out-of-2 OT on bit strings. We point out that our correlated OT (cf. §5.4) can be directly used in their semi-honest construction to improve communication complexity by factor two.

1.2 Our Contributions and Outline

In this paper, we present more efficient protocols for OT extensions. This is somewhat surprising since the protocol of [32] sounds optimal given that only three hash function computations are needed per transfer. Interestingly, our protocols do not lower the number of hash function operations. However, we observe that significant cost is incurred due to other factors than the hash function operations. We propose several algorithmic (§4) and protocol (§5) optimizations and obtain an OT extension protocol (General OT, G-OT §5.3) that has lower communication, faster computation, and can be parallelized. Additionally, we propose two OT extension protocols that are specifically designed to be used in secure computation protocols and which reduce the communication and computation even further. The first of these protocols (Correlated OT, C-OT §5.4) is suitable for secure computation protocols that require correlated inputs, such as Yao's garbled circuits protocol with the free-XOR technique [38, 58]. The second protocol (Random OT, R-OT §5.4) can be used in secure computation protocols where the inputs can be random, such as GMW with multiplication triples [3, 19] (cf. §5.1). We apply our optimizations to the OT extension implementation of [56] (which is based on [10]) and demonstrate the improvements by extensive experiments (§6).[1] A summary of the time complexity for 1-out-of-2 OTs on 80-bit strings is given in Fig. 1. While the original protocol of [32] as implemented in [56] evaluates 2^{23} OTs in 18.0 s with one thread and in 14.5 s with two threads, our improved R-OT protocol requires only 8.4 s with one thread and 4.2 s with two threads, which demonstrates the scalability of our approach.

Secure random number generation. In §3 we emphasize that when OT protocols are used as building block in a secure computation protocol, it is very important that random values are generated with a cryptographically strong random number generator. In fact, we show an attack on the latest version of the FastGC [28] implementation (version v0.1.1) of Yao's protocol which uses a weak random number generator. Our attack allows the full recovery of the inputs of both parties. To protect against our attack, a cryptographically strong random number generator needs to be used (which results in an increased runtime).

Faster semi-honest base OT without random oracle. In the semi-honest model, the OT of [46] is the fastest known with $2 + n$ exponentiations for the sender and $2n$ fixed-base exponentiations for the receiver, for n OTs. However, it is proven secure only in the random oracle model, which is why the authors of [46] also provide a slower semi-honest OT that relies on the DDH assumption, which has complexity $4n$ fixed-base + $2n$ double exponentiations for the sender and $1 + 3n$ fixed-base + n exponentiations for the receiver. In §5.2 we construct a protocol secure under the Decisional Diffie-Hellmann (DDH) assumption that is much faster when many transfers are run (as in the case of OT extensions where

[1] Our implementation is available online at `http://encrypto.de/code/OTExtension`.

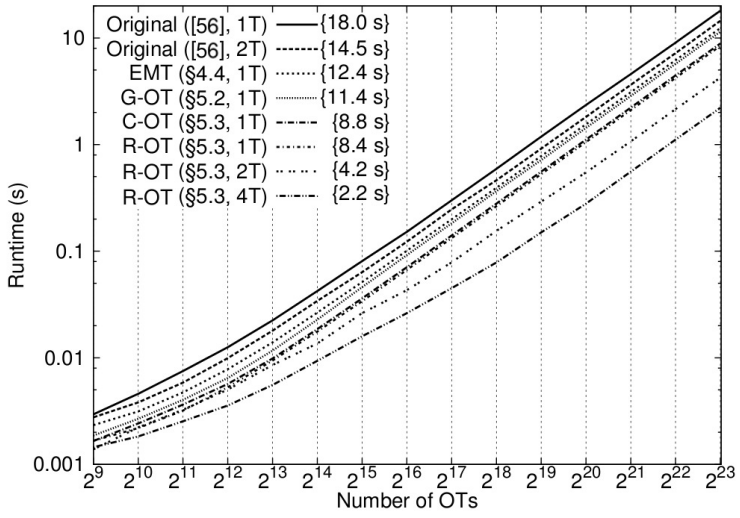

Fig. 1. Runtime for 1-out-of-2 OT extension optimizations on 80-bit strings. The reference and number of threads is given in (); the time for 2^{23} OTs is given in {}.

80 base OTs are needed) and is only slightly slower than the fastest OT in the random oracle model (§6.1).

Faster OT extensions. In §5.3 we present an improved version of the original OT extension protocol of [32] with reduced communication and computation complexity. Furthermore, we demonstrate how the OT extension protocol can be processed in independent blocks, allowing OT extension to be parallelized and yielding a much faster runtime (§4.1). In addition, we show how to implement the matrix transpose operation using a cache-efficient algorithm that operates on multiple entries at once (§4.2); this has a significant effect on the runtime of the protocol. Finally, we show how to reduce the communication by approximately one quarter (depending on the bit-length of the inputs). This is of great importance since local computations of the OT extension protocol are so fast that the communication is often the bottleneck, especially when running the protocol over the Internet or even wireless networks.

Extended OT functionality. Our improved protocol can be used in any setting that regular OT can be used. However, with a mind on the application of secure computation, we further optimize the protocol by taking into account its use in the protocols of Yao [58] and GMW [19] in §5.4. For Yao's garbled circuits protocol, we observe that the OT extension protocol can choose the first value randomly and output it to the sender while the second value is computed as a function of the first value. For the GMW protocol. we observe that the OT extension protocol can choose both values randomly and output them to the

sender. In both cases, the communication is reduced to a *half* (or even less) of the original protocol of [32].

Experimental evaluation and applications. In §6 we experimentally verify the performance improvements of our proposed optimizations for OT and OT extension. In §7 we demonstrate their efficiency gains for faster secure computation, by giving performance benchmarks for various application scenarios. For the Yao's garbled circuits framework FastGC [28], we achieve an improvement up to factor 9 for circuits with many inputs for the receiver, whereas we improve the runtime of the GMW implementation of [56] by factor 2, e.g., a Levenshtein distance circuit with 1.29 billion AND gates can now be evaluated at a rate of 1.2 million AND gates per second.

2 Preliminaries

In the following, we summarize the security parameters used in our paper (§2.1) and describe the OT extension protocol of [32] (§2.2), Yao's garbled circuits protocol (§2.3), and the GMW protocol (§2.4) in more detail. Standard definitions of security are given in Appendix A.

2.1 Security Parameters

Throughout the paper, we denote the symmetric security parameter by κ. Tab. 1 lists usage times (time frames) for different values of the symmetric security parameter κ (SYM) and corresponding field sizes for finite field cryptography (FFC) and elliptic curve cryptography (ECC) as recommended by NIST [51]. For FCC we use a subgroup of order $q = 2\kappa$. For ECC we use Koblitz curves which had the best performance in our experiments (cf. [13])[2].

Security (Time Frames)	**SYM**	**FFC**	**ECC**
Short (legacy)	80	1024	K-163
Medium (< 2030)	112	2048	K-243
Long (> 2030)	128	3072	K-283

Table 1. Security parameters and recommended key sizes.

2.2 Oblivious Transfer and OT Extension

The m-times 1-out-of-2 OT functionality for ℓ-bit strings, denoted $m \times OT_\ell$, is defined as follows: The sender S inputs m pairs of strings $\mathbf{x}_i^0, \mathbf{x}_i^1 \in \{0,1\}^\ell$

[2] The prime-field curves outperformed the Koblitz curves only for [46]-STD using the long-term security setting.

$(1 \leq i \leq m)$, the receiver R inputs a string $\mathbf{r} = (r_1, \ldots, r_m)$ of length m, and R obtains $\mathbf{x}_j^{r_j}$ $(1 \leq j \leq m)$ as output. OT ensures that S learns nothing about $\mathbf{r}$ and R learns nothing about $\mathbf{x}_j^{1-r_j}$.

An *OT extension protocol* implements the $m \times OT_\ell$ functionality using a small number of actual OTs, referred to as base OTs, and cheap symmetric cryptographic operations. In [32] it is shown how to implement the $m \times OT_\ell$ functionality using a single call to $\kappa \times OT_m$, and $3m$ hash function computations. Note that $\kappa \times OT_m$ can be implemented via a single call to $\kappa \times OT_\kappa$ in order to obliviously transfer symmetric keys, and then using a pseudo-random generator G to obliviously transfer the actual inputs of length m (cf. [30, 32]). In the first step of [32], S chooses a random string $\mathbf{s} \in_R \{0,1\}^\kappa$, and R chooses a random $m \times \kappa$ bit matrix $T = [\mathbf{t}^1 \mid \ldots \mid \mathbf{t}^\kappa]$, where $\mathbf{t}^i \in \{0,1\}^m$ denotes the i-th column of T. The parties then invoke the $\kappa \times OT_m$ functionality, where R plays the sender with inputs $(\mathbf{t}^i, \mathbf{t}^i \oplus \mathbf{r})$ and S plays the receiver with input $\mathbf{s}$. Let $Q = [\mathbf{q}^1 \mid \ldots \mid \mathbf{q}^\kappa]$ denote the $m \times \kappa$ matrix received by S. Note that $\mathbf{q}^i = (s_i \cdot \mathbf{r}) \oplus \mathbf{t}^i$ and $\mathbf{q}_j = (r_j \cdot \mathbf{s}) \oplus \mathbf{t}_j$ (where $\mathbf{t}_j$, $\mathbf{q}_j$ are the j-th rows of T and Q, respectively). S sends (y_j^0, y_j^1) where $y_j^0 = x_j^0 \oplus H(\mathbf{q}_j)$ and $y_j^1 = x_j^1 \oplus H(\mathbf{q}_j \oplus \mathbf{s})$, for $1 \leq j \leq m$. R finally outputs $z_j = y_j^{r_j} \oplus H(\mathbf{t}_j)$ for every j. The protocol is secure assuming that $H : \{0,1\}^m \mapsto \{0,1\}^\ell$ is a random oracle, or a correlation robust function as in Definition A2; see [32] for more details.

2.3 Yao's Garbled Circuits Protocol

Yao's garbled circuits protocol [58] allows two parties to securely compute an arbitrary function that is represented as Boolean circuit. The sender S encrypts the Boolean gates of the circuit using symmetric keys and sends the encrypted function together with the keys that correspond to his input bits to the receiver R. R then uses a 1-out-of-2 OT to obliviously obtain the keys that correspond to his inputs and evaluates the encrypted function by decrypting it gate by gate. To obtain the output, R sends the resulting keys to S or S provides a mapping from keys to output bits. We emphasize that Yao's garbled circuits protocol requires a 1-out-of-2 OT on κ-bit strings for *each input bit of R*. For our experiments we use the Yao's garbled circuits framework FastGC [28].

2.4 The GMW Protocol

The protocol of Goldreich, Micali, and Wigderson (GMW) [19] also represents the function to be computed as a Boolean circuit. Both parties secret-share their inputs using the XOR operation and evaluate the Boolean circuit as follows. An XOR gate is computed by locally XORing the shares while an AND gate is evaluated interactively with the help of a multiplication triple [3, 56] which can be precomputed by two random 1-out-of-2 OTs on bits (cf. §5.1). To reconstruct the outputs, the parties exchange their output shares. The performance of GMW depends on the number of OTs and on the depth of the evaluated circuit, since the evaluation of AND gates requires interaction. For our experiments we use

the GMW framework of [56], which is an optimization of the framework of [10] for the two party case.

3 Random Number Generation

The correct instantiation of primitives in implementations of cryptographic protocols is a challenging task, since various security properties have to be met. For instance, an important security property of a pseudo-random generator (PRG) is its unpredictability, i.e., given a sequence of pseudo-random bits $x_1...x_n$, the next bit x_{n+1} should not be predictable. If the security property of the primitive is not met, the security of the overall scheme can be compromised. We found that this was the case for the FastGC framework in version 0.1.1 [28] that uses the standard Java Random class in order to generate random values used in the base OTs, the random choices of vector $\mathbf{s}$ and matrix T in the OT extension, and the input keys of the garbled circuit. Overall, this enables an attack that allows each party to recover the inputs of the respective other party, as we will describe now.

3.1 The Java Random Class

The Java Random class implements a so-called *truncated linear congruential generator (T-LCG)* with secret seed $\psi \in \{0,1\}^{48}$. Random numbers can be generated by invoking the `next` method of an object of the Java Random class which takes as input the requested number of random bits b (for $1 \leq b \leq 32$), updates the seed $\psi' = (\alpha\psi + \beta)\ mod\ m$, and returns the topmost b bits of ψ, where $\alpha =$ 0x5DEECE66D, $\beta =$ 0xB, and $m = 2^{48}$ are public constants. If more than 32 random bits are needed, `next` is called repeatedly until a sufficient number of bits has been generated.
The security of T-LCGs was widely studied and they were shown to be predictable [22], even if the generated sequence is not directly output [5]. In case of the Java Random class, each iteration reveals b bits of the seed, leaving a remaining entropy of $48 - b$ bits. Furthermore, consecutive values can be used to build linear equations.
For our analysis, we assume that the generated random value has at least length 64 bits, i.e., it was generated by two consecutive calls to the `next` method with $b = 32$. This holds for the FastGC framework [28] which uses a Java Random object to generate symmetric keys and the columns of the T matrix (we use the first 64 bits only). To predict the output of the Java Random object, we recover its secret seed $\psi = \psi_1...\psi_{48}$ using the 64 bit output $d = d_1...d_{64}$. Since the topmost 32 bits are directly used as output, we have $\psi_{17}...\psi_{48} = d_1...d_{32}$. In addition, we have $\psi'_{17}...\psi'_{48} = d_{33}...d_{64}$. Now, the remaining lower 16 bits $\psi_1...\psi_{16}$ can be recovered using the linear equation $\psi' = (\alpha\psi + \beta) \bmod m$. Specifically, for each of the 2^{16} possible values of ψ we compute $(\alpha\psi + \beta) \bmod m - (\psi'_{17}...\psi'_{48}) \cdot 2^{16}$. Now, for the correct value of ψ the result will be zero in the 32 most-significant bits and so will be smaller than 2^{16}, whereas for all other values it will be larger

(with high probability). In practice, this suffices for finding the entire seed ψ in 2^{16} steps, which takes under a second. The recovered secret seed ψ can then be used to predict the output of the Java Random object.

3.2 Exploiting the Weak PRG in FastGC [28]

We demonstrate how the usage of the Java Random class in version v0.1.1 of the FastGC [28] framework can be exploited such that the sender can recover the input bits of the receiver using the T matrix generated in the OT extension protocol (cf. §2.2), and the receiver can recover the input bits of the sender using the sender's input keys to the garbled circuit. We implemented and verified both attacks on FastGC, which both run in less than a second. Note that both attacks are carried out on the honestly generated transcript, as required for the setting of semi-honest adversaries.

Recovering the Receiver's Inputs. The sender can recover the receiver's input bits using the T matrix, which is chosen randomly by the receiver in the OT extension (cf. §2.2). Upon receiving the matrix $Q = [\mathbf{q}^1 \mid \ldots \mid \mathbf{q}^\kappa]$, the sender knows that $\mathbf{q}^i = \mathbf{t}^i$, if $s_i = 0$, and $\mathbf{q}^i = \mathbf{t}^i \oplus \mathbf{r}$, if $s_i = 1$. Hence, whenever the receiver has $s_i = 0$, the sender obtains $\mathbf{q}^i = \mathbf{t}^i$ and can recover an intermediate seed ψ of the Java Random object that was used to generate this column of T. Afterwards, the sender computes for $j > i$ consecutive random outputs $\mathbf{t}^j$ until he obtains a column $\mathbf{q}^j \neq t^j$ where $s_j = 1$ which occurs with overwhelming probability $1 - \frac{\kappa+1}{2^\kappa}$. Now, the sender can recover the receiver's input bits $\mathbf{r}$ by computing $\mathbf{q}^j \oplus \mathbf{t}^j = \mathbf{t}^j \oplus \mathbf{r} \oplus \mathbf{t}^j = \mathbf{r}$.

Recovering the Sender's Inputs. The receiver can recover the sender's input bits using the sender's input keys to the garbled circuit. In FastGC, the sender generates random symmetric keys $k_i \in \{0,1\}^\kappa$ for each of his ℓ input bits $b_i \in \{0,1\}$ using the same Random object. If $b_i = 0$, he sends $K_i = k_i$ to the receiver, else he sends $K_i = k_i \oplus (\Delta||0)$, where $\Delta \in \{0,1\}^{\kappa-1}$ is a constant global value [38]. In order to recover the sender's input bits, the receiver iteratively computes a candidate for the seed with which K_i was generated, computes the next $\ell - i$ keys k'_j $(i < j \leq \ell)$ and checks whether the candidate seed generates a consistent view for the observed values K'_j. If $b_i = 0$, then $K_i = k_i$ and the receiver knows that he has recovered the correct seed by finding either $k'_{i+1} \oplus k'_{i+2} = K_{i+1} \oplus K_{i+2}$ if there are at least two more input bits $b_{i+1} = b_{i+2} = 1$ or $k'_j = K_j$ if another input bit is $b_j = 0$. Once the receiver has found such a $b_i = 0$, he can recover all subsequent input bits by checking whether $k'_j = K_j$ $(\Rightarrow b_j = 0)$ or not $(\Rightarrow b_j = 1)$. If $b_i = 1$, then $K_i = k_i \oplus (\Delta||0)$ and the receiver recovers the wrong seed such that neither $K'_j = K_j$ nor $K'_{i+1} \oplus K'_{i+2} = K_{i+1} \oplus K_{i+2}$ hold with very high probability. Thus, the receiver knows that $b_i = 1$ and repeats the attack for $i+1$. Note that this attack fails if the sender has less than three input bits or all except the last two input bits of the sender are set to 1. In this case, however, the receiver can recover the input bits with high probability by using the remaining $\kappa - 64$ bits of the key to check if the candidate seed is correct.

Securing FastGC [28]. Securing the FastGC framework is relatively easy, since Java also provides a cryptographically strong random number generator, called SecureRandom, which by default is implemented based on SHA-1.[3] Replacing all usage of the Random class by SecureRandom increased the runtime of our experiments in §7 by around $0.5 - 4\%$, depending on the application. A complementary method to reduce the overhead in runtime is to use our correlated input OT extension of §5.4 which eliminates the need of generating a random T matrix s.t. our attack for reconstructing the receiver's inputs no longer works. Nevertheless, all randomness that is needed (even for our method) must be generated using SecureRandom.

4 Algorithmic Optimizations

In the following we describe algorithmic optimizations that improve the scalability and computational complexity of OT extension protocols. We identified computational bottlenecks in OT extension by micro-benchmarking the 1-out-of-2 OT extension implementation of [56].[4] We found that the combined computation time of S and R was mostly spent on three operations: the matrix transposition (43%), the evaluation of H, implemented with SHA-1 (33%), and the evaluation of G, implemented with AES (14%). To speed up OT extension, we propose to use parallelization (§4.1) and an efficient algorithm for bit-matrix transposition (§4.2). Note that these implementation optimizations are of general nature and can be applied to our, but also to other OT extension protocols with security against stronger active/malicious adversaries, e.g., [32, 49]. As we will show later in our experiments in §6.2, both algorithmic improvements result in substantially faster runtimes, but only in settings where the computation is the bottleneck, i.e., over a fast network such as a LAN.

4.1 Blockwise Parallelized OT Extension

Previous OT extension implementations [10,56] improved the performance of OT extension by using a *vertical* pipelining approach, i.e., one thread is associated to each step of the protocol: the first thread evaluates the pseudorandom generator G and the second thread evaluates the correlation robust function H (cf. §2.2). However, as evaluation of G is faster than evaluation of H, the workload between the two threads is distributed unequally, causing idle time for the first thread. Additionally, this method for pipelining is designed to run exactly two threads and thus cannot easily be scaled to a larger number of threads.
As observed in [24], a large number of OT extensions can be performed by *sequentially* running the OT extension protocol on blocks of fixed size. This

[3] In response to our findings, the usage of Random has been replaced with SecureRandom in version 0.1.2 of FastGC.

[4] Note that the implementation in [56] performs 1-out-of-4 OT, but we adapted their implementation since our protocol optimizations in §5 target 1-out-of-2 OT extension.

reduces the total memory consumption at the expense of more communication rounds.
We propose to use a *horizontal* pipelining approach that splits the matrices processed in the OT extension protocol into *independent* blocks that can be processed in parallel using multiple threads with equal workload, i.e., each of the N threads evaluates the OT extension protocol for $\frac{m}{N}$ inputs in parallel. Each thread uses a separate socket to communicate with its counterpart on the other party, s.t. network scheduling is done by the operating system.

4.2 Efficient Bit-Matrix Transposition

The computational complexity of cryptographic protocols is often measured by counting the number of invocations of cryptographic primitives, since their evaluation often dominates the runtime. However, non-cryptographic operations can also have a high impact on the overall run time of executions although they might seem insignificant in the protocol description. Matrix transposition is an example for such an operation. It is required during the OT extension protocol to transpose the $m \times \kappa$ bit-matrix T (cf. §2.2), which is created column-wise but hashed row-wise. Although transposition is a trivial operation, it has to be performed individually for each entry in T, making it a very costly operation.
We propose to efficiently implement the matrix transposition using Eklundh's algorithm [14], which uses a divide-and-conquer approach to recursively swap elements of adjacent rows (cf. Fig. 2). This decreases the number of swap operations for transposing a $n \times n$ matrix from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log_2 n)$. Additionally, since we process a bit-matrix, we can perform multiple swap operations in parallel by loading multiple bits into one register. Thereby, we again reduce the number of swap operations from $\mathcal{O}(n \log_2 n)$ to $\mathcal{O}(\lceil \frac{n}{r} \rceil \log_2 n)$, where r is the register size of the CPU ($r = 64$ for the machines used in our experiments). Jumping ahead to the evaluation in §6, this reduced the total time for the matrix transposition by approximately a factor of 9 from 7.1 s to 0.76 s per party.

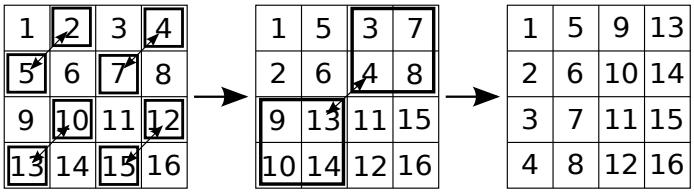

Fig. 2. Efficient matrix transposition of a 4×4 matrix using Eklundh's algorithm.

5 Protocol Optimizations

In this section, we show how to efficiently base the GMW protocol on random 1-out-of-2 OTs (§5.1), introduce a new OT protocol (§5.2), outline an optimized OT extension protocol (§5.3), and optimize OT extension for usage in secure computation protocols (§5.4).

5.1 GMW with Random 1-out-of-2 OTs

An AND gate in the GMW protocol can be computed efficiently using the multiplication triple functionality [3]: the parties hold no input, and the functionality chooses random bits $a_0, a_1, b_0, b_1, c_0, c_1$ under the constraint that $c_0 \oplus c_1 = (a_0 \oplus a_1)(b_0 \oplus b_1)$. Each P_i receives the shares labeled with i. To precompute the multiplication triples, previous works suggest to use 1-out-of-4 bit OT [10, 56].
In the following, we present a different approach for generating multiplication triples using two random 1-out-of-2 OTs on bits (R-OT). The R-OT functionality is exactly the same as OT, except that the sender gets two random messages as *outputs*. Later in §5.4, we will show that R-OT can be instantiated more efficiently than OT. In comparison to 1-out-of-4 bit OTs, using two R-OTs only slightly increases the computation complexity (one additional evaluation of G and H and two additional matrix transpositions), but improves the communication complexity by a factor of 2.
In order to generate a multiplication triple, we first introduce the f^{ab} functionality that is implemented in Algorithm 1 using R-OT. In the f^{ab} functionality, the parties hold no input and receive random bits $((a, u), (b, v))$, under the constraint that $ab = u \oplus v$. Now, note that for a multiplication triple $c_0 \oplus c_1 = (a_0 \oplus a_1)(b_0 \oplus b_1) = a_0b_0 \oplus a_0b_1 \oplus a_1b_0 \oplus a_1b_1$. The parties can generate a multiplication triple by invoking the f^{ab} functionality twice: in the first invocation P_0 acts as R to obtain (a_0, u_0) and P_1 acts as S to obtain (b_1, v_1) with $a_0b_1 = u_0 \oplus v_1$; in the second invocation P_1 acts as R to obtain (a_1, u_1) and P_0 acts as S to obtain (b_0, v_0) with $a_1b_0 = u_1 \oplus v_0$. Finally, each P_i sets $c_i = a_ib_i \oplus u_i \oplus v_i$. For correctness, observe that $c_0 \oplus c_1 = (a_0b_0 \oplus u_0 \oplus v_0) \oplus (a_1b_1 \oplus u_1 \oplus v_1) = a_0b_0 \oplus (u_0 \oplus v_1) \oplus (u_1 \oplus v_0) \oplus a_1b_1 = a_0b_0 \oplus a_0b_1 \oplus a_1b_0 \oplus a_1b_1 = (a_0 \oplus a_1)(b_0 \oplus b_1)$, as required. A proof sketch for security is given in Appendix B.

Algorithm 1 Random $(a, u), (b, v)$ with $ab = u \oplus v$

1: R chooses $a \in_R \{0, 1\}$.
2: S and R perform a R-OT with a as input of R.
   S obtains bits x_0, x_1 and R obtains bit x_a as output.
3: R sets $u = x_a$; S sets $b = x_0 \oplus x_1$ and $v = x_0$.
   [Note that $ab = u \oplus v$ as $ab = a(x_0 \oplus x_1) = (a(x_0 \oplus x_1) \oplus x_0) \oplus x_0 = x_a \oplus x_0 = u \oplus v$.]
4: R outputs (a, u) and S outputs (b, v).

Proof Sketch In Algortihm 1, a is generated randomly and used as input for R-OT. R gains no information on $b = x_0 \oplus x_1$, since x_0 and x_1 are randomly generated by the R-OT. By the definition of OT, S gains no information on a and R gains no information on v.

5.2 Optimized Oblivious Transfer

The best known protocols for oblivious transfer with security in the presence of semi-honest adversaries are those of Naor-Pinkas [46]. They present two protocols; a more efficient protocol that is secure in the random oracle model and a less efficient protocol that is secure in the standard model and under standard assumptions. In this section, we describe a new semi-honest OT protocol that is secure in the standard model and is essentially an optimized instantiation of the OT protocol of [16]. When implemented over elliptic curves, our protocol is about three times faster than the standard model OT of [46] and only two times slower than the random oracle OT of [46] (see §6.1 for a comparison of the protocol runtimes). Hence, our protocol is a good alternative for those preferring to not rely on random oracles.

Our $n \times OT_\ell$ protocol is based on the DDH assumption and uses a key derivation function (KDF); see Definition A1. We also assume that it is possible to sample a random element of the group, and the DDH assumption will remain hard even when the coins used to sample the element are given to the distinguisher (i.e., (g, h, g^a, h^a) is indistinguishable from (g, h, g^a, g^b) for random a, b, even given the coins used to sample h). This holds for all known groups in which the DDH problem is assumed to be hard and can be implemented as described next. For finite fields, one can sample a random element $h \in \mathbb{Z}_p$ of order q by choosing a random $x \in_R \mathbb{Z}_p$ and computing $h = x^{(p-1)/q}$ until $h \neq 1$. For elliptic curves, one chooses a random x-coordinate, obtains a quadratic equation for the y-coordinate and randomly chooses one of the solutions as h (if no solution exists, start from the beginning).

The computational complexity of our protocol for $n \times OT_\ell$ is $2n$ exponentiations for the sender $S$and $2n$ *fixed-base* exponentiations for the receiver R (in fixed-base exponentiations, the same "base" g is raised to the power of many different exponents; more efficient exponentiation algorithms exist for this case [44, Sec. 14.6.3]). In addition, S computes the KDF function $2n$ times, and R computes it n times. R samples n random group elements according to the above definition. See Protocol 51 for a detailed description of the protocol.

The protocol is secure in the presence of a semi-honest adversary (see Definition A3). The view of a corrupted sender consists of the pairs $\{(h_i^0, h_i^1)\}_{i=1}^n$ which are completely independent of the receiver's inputs, and therefore can be simulated perfectly. For the corrupted receiver, we need to show the existence of a simulator $\mathcal{S}_1$ that produces a computationally-indistinguishable view, given the inputs and outputs of the receiver, i.e., $\boldsymbol{\sigma}$ and $(x_1^{\sigma_1}, \ldots, x_n^{\sigma_n})$, without knowing the other sender values $(x_1^{1-\sigma_1}, \ldots, x_n^{1-\sigma_n})$. $\mathcal{S}_1$ works by running an execution of the protocol playing an honest S using inputs $x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}$ and using $x_i^{1-\sigma_i} = 0$ for all $1 \leq i \leq n$. The only difference between the view

PROTOCOL 51 (Optimized $n \times OT_\ell$ Protocol)

Inputs: S holds n pairs (x_i^0, x_i^1) of ℓ-bit strings, for every $1 \leq i \leq n$. R holds the selection bits $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_n)$. The parties agree on a group $\langle \mathbb{G}, q, g \rangle$ for which the DDH is hard, and a key derivation function KDF.

First Round (Receiver): Choose random exponents $\alpha_i \in_R \mathbb{Z}_q$ and random group elements $h_i \in_R \mathbb{G}$ for every $1 \leq i \leq n$. Then, for every i, set (h_i^0, h_i^1) as follows:

$$(h_i^0, h_i^1) \stackrel{\text{def}}{=} \begin{cases} (g^{\alpha_i},\ h_i) & \text{if } \sigma_i = 0 \\ (h_i,\ g^{\alpha_i}) & \text{if } \sigma_i = 1 \end{cases}$$

Send the pairs (h_i^0, h_i^1) to S.

Second Round (Sender): Choose a random element $r \in_R \mathbb{Z}_q$ and compute $u = g^r$. Then, for each pair (h_i^0, h_i^1) compute the keys: $(k_i^0, k_i^1) = \left((h_i^0)^r, (h_i^1)^r\right)$ and compute the pair of ciphertexts:

$$v_i^0 = x_i^0 \oplus \mathsf{KDF}(k_i^0) \quad \text{and} \quad v_i^1 = x_i^1 \oplus \mathsf{KDF}(k_i^1).$$

Send u together with the n pairs (v_i^0, v_i^1) to R.

Output Computation (Receiver): For every $1 \leq i \leq n$, set $k_i^{\sigma_i} = u^{\alpha_i}$ and $x_i^{\sigma_i} = v_i^{\sigma_i} \oplus \mathsf{KDF}(k_i^{\sigma_i})$. R outputs $(x_1^{\sigma_1}, \ldots, x_n^{\sigma_n})$; S has no output.

of the receiver generated by the simulator and in a real execution is regarding the values $\{v_i^{1-\sigma_i}\}_{i=1}^n$, which equal $x_i^{1-\sigma_i} \oplus \mathsf{KDF}(k_i^{1-\sigma_i})$ in a real execution and just $\mathsf{KDF}(k_i^{1-\sigma_i})$ in the simulation. From the security of the KDF with respect to DDH (see Definition A1), and using a standard hybrid argument, the values $(\mathsf{KDF}(k_1^{1-\sigma_1}), \ldots, \mathsf{KDF}(k_n^{1-\sigma_n})) = (\mathsf{KDF}(h_1^r), \ldots, \mathsf{KDF}(h_n^r))$ are indistinguishable from n uniform strings $z_1, \ldots, z_n$ each of size ℓ (even when the distinguisher sees $\langle \mathbb{G}, q, g, u = g^r \rangle$). This implies that the values $\{v_i^{1-\sigma_i}\}_{i=1}^n$ in the real execution are computationally indistinguishable from those in the simulation.

An additional optimization for random OT. When constructing OT extensions (see §2.2) the parties first run $\kappa \times OT_\kappa$ on random inputs (this holds for our optimized OT extension protocol, and also for the original protocol of [32] if $\kappa \times OT_m$ is implemented via $\kappa \times OT_\kappa$ as described in §2.2). Observe that in this case, the sender only needs to send $u = g^r$ to the receiver R; the parties can then derive the values locally (S by computing $x_i^0 = \mathsf{KDF}((h_i^0)^r)$ and $x_i^1 = \mathsf{KDF}((h_i^1)^r)$, and R by computing $x_i^{\sigma_i} = \mathsf{KDF}(u^{\alpha_i})$). This reduces the *communication* since the elements v_i^0 and v_i^1 do not have to be sent. In addition, this means that the messages sent by S and R are actually independent of each other, and so the protocol consists of a *single round* of communication. (As pointed out in [49], this optimization can also be carried out on the protocols of Naor-Pinkas [46]. However, those protocols still require two rounds of communication which can be a drawback in high latency networks.) The timings that appear in §7 are for an implementation that uses this additional optimization.[5]

[5] We remark that, in order to prove the security of this optimization in the standard model (without a random oracle), we need to change the ideal functionality for the random OT such that for every i, the output of the sender is $(\beta_i^0, x_i^0 = \mathsf{KDF}(g^{\beta_i^0}))$ and $(\beta_i^1, x_i^1 = \mathsf{KDF}(g^{\beta_i^1}))$, and the output of the receiver is $(\sigma_i, \beta_i^{\sigma_i}, \mathsf{KDF}(g^{\beta_i^{\sigma_i}}))$. That is,

5.3 Optimized General OT Extension

In the following, we optimize the $m \times OT_\ell$ extension protocol of [32], described in §2.2. Recall, that in the first step of the protocol in [32], R chooses a huge $m \times \kappa$ matrix $T = [\mathbf{t}^1 | \dots | \mathbf{t}^\kappa]$ while S waits idly. The parties then engage in a $\kappa \times OT_m$ protocol, where the inputs of the receiver are $(\mathbf{t}^i, \mathbf{t}^i \oplus \mathbf{r})$ where $\mathbf{r}$ is its input in the outer $m \times OT_\ell$ protocol (m selection bits). After the OT, S holds $\mathbf{t}^i \oplus (s_i \cdot \mathbf{r})$ for every $1 \leq i \leq \kappa$. As described in the appendices of [30, 32], the protocol can be modified such that R only needs to choose two small $\kappa \times \kappa$ matrices $K_0 = [\mathbf{k}_1^0 | \dots | \mathbf{k}_\kappa^0]$ and $K_1 = [\mathbf{k}_1^1 | \dots | \mathbf{k}_\kappa^1]$ of seeds. These seeds are used as input to $\kappa \times OT_\kappa$; specifically R's input as sender in the i-th OT is $(\mathbf{k}_i^0, \mathbf{k}_i^1)$ and, as in [32], the input of S is s_i. To transfer the m-bit tuple $(\mathbf{t}^i, \mathbf{t}^i \oplus \mathbf{r})$ in the i-th OT, R expands $\mathbf{k}_i^0$ and $\mathbf{k}_i^1$ using a pseudo-random generator G, sends $(\mathbf{v}_i^0, \mathbf{v}_i^1) = (G(\mathbf{k}_i^0) \oplus \mathbf{t}^i, G(\mathbf{k}_i^1) \oplus \mathbf{t}^i \oplus \mathbf{r})$, and S recovers $G(\mathbf{k}_i^{s_i}) \oplus \mathbf{v}_i^{s_i}$.

Our main observation is that, instead of choosing $\mathbf{t}^i$ randomly, we can set $\mathbf{t}^i = G(\mathbf{k}_i^0)$. Now, R needs to send only one m-bit element $\mathbf{u}^i = G(\mathbf{k}_i^0) \oplus G(\mathbf{k}_i^1) \oplus \mathbf{r}$ to S (whereas in previous protocols of [30, 32] two m-bit elements were sent). Observe that if S had input $s_i = 0$ in the i-th OT, then it can just define its output $\mathbf{q}^i$ to be $G(\mathbf{k}_i^0) = G(\mathbf{k}_i^{s_i})$. In contrast, if S had input $s_i = 1$ in the i-th OT, then it can define its output $\mathbf{q}^i$ to be $G(\mathbf{k}_i^1) \oplus \mathbf{u}^i = G(\mathbf{k}_i^{s_i}) \oplus \mathbf{u}^i$. Since $\mathbf{u}^i = G(\mathbf{k}_i^0) \oplus G(\mathbf{k}_i^1) \oplus \mathbf{r}$, we have that $G(\mathbf{k}_i^1) \oplus \mathbf{u}^i = G(\mathbf{k}_i^0) \oplus \mathbf{r} = \mathbf{t}^i \oplus \mathbf{r}$, as required. The full description of our protocol is given in Protocol 52. This optimization is significant in applications of $m \times OT_\ell$ extension where m is very large and ℓ is short, such as in GMW. In typical use-cases for GMW (cf. §7), m is in the size of several millions to a billion, while ℓ is one. Thereby, the communication complexity of GMW is almost reduced by half.

In addition, as in [30], observe that unlike [32] the initial OT phase in Protocol 52 is completely independent of the actual inputs of the parties. Thus, the parties can perform the initial OT phase before their inputs are determined.

Finally, another problem that arises in the original protocol of [32] is that the entire $m \times \kappa$ matrix is transmitted together and processed. This means that the number of OTs to be obtained must be predetermined and, if m is very large, this results in considerable latency as well as memory management issues. As in [24], our optimization enables us to process small blocks of the matrix at a time, reducing latency, computation time, and memory management problems. In addition, it is possible to continually extend OTs, with no a priori bound on m. This is very useful in a secure computation setting, where parties may interact many times together with no a priori bound.

Theorem 53 *Assuming that G is a pseudorandom generator and H is a correlation-robust function (as in Definition A2), Protocol 52 privately-computes the $m \times OT_\ell$-functionality in the presence of semi-honest adversaries, in the $\kappa \times OT_\kappa$-hybrid model.*

---

in addition to receiving their input and output from the random OT functionality, the parties receive the "discrete log" of the pertinent values. This additional information is of no consequence in *our* applications of random OT.

PROTOCOL 52 (General OT extension protocol)

Inputs: S holds m pairs (x_j^0, x_j^1) of ℓ-bit strings, for every $1 \leq j \leq m$. R holds m selection bits $\mathbf{r} = (r_1, \ldots, r_m)$.

Initial OT Phase (base OTs):

1. S choose a random string $\mathbf{s} = (s_1, \ldots, s_\kappa)$ and R chooses κ pairs of κ-bits seeds $\{(\mathbf{k}_i^0, \mathbf{k}_i^1)\}_{i=1}^{\kappa}$.
2. The parties invoke the $\kappa \times OT_\kappa$-functionality, where S plays the *receiver* with input $\mathbf{s}$ and R plays the *sender* with inputs $(\mathbf{k}_i^0, \mathbf{k}_i^1)$ for every $1 \leq i \leq \kappa$.
3. For every $1 \leq i \leq \kappa$, let $\mathbf{t}^i = G(\mathbf{k}_i^0)$. Let $T = [\mathbf{t}^1 | \ldots | \mathbf{t}^k]$ denote the $m \times \kappa$ bit matrix where the i-th column is $\mathbf{t}^i$, and let $\mathbf{t}_j$ denote the j-th row of T, for $1 \leq j \leq m$.

OT extension Phase[a]**:**

1. R computes $\mathbf{t}^i = G(\mathbf{k}_i^0)$ and $\mathbf{u}^i = \mathbf{t}^i \oplus G(\mathbf{k}_i^1) \oplus \mathbf{r}$, and sends $\mathbf{u}^i$ to S for every $1 \leq i \leq \kappa$.
2. For every $1 \leq i \leq \kappa$, S defines $\mathbf{q}^i = (s_i \cdot \mathbf{u}^i) \oplus G(\mathbf{k}_i^{s_i})$. (Note that $\mathbf{q}^i = (s_i \cdot \mathbf{r}) \oplus \mathbf{t}^i$.)
3. Let $Q = [\mathbf{q}^1 | \ldots | \mathbf{q}^\kappa]$ denote the $m \times \kappa$ bit matrix where the i-th column is $\mathbf{q}^i$. Let $\mathbf{q}_j$ denote the j-th row of the matrix Q. (Note that $\mathbf{q}_j = (r_j \cdot \mathbf{s}) \oplus \mathbf{t}_j$.)
4. S sends (y_j^0, y_j^1) for every $1 \leq j \leq m$, where:
$$y_j^0 = x_j^0 \oplus H(j, \mathbf{q}_j) \quad \text{and} \quad y_j^1 = x_j^1 \oplus H(j, \mathbf{q}_j \oplus \mathbf{s})$$
5. For $1 \leq j \leq m$, R computes $x_j^{r_j} = y_j^{r_j} \oplus H(j, \mathbf{t}_j)$.

Output: R outputs $(x_1^{r_1}, \ldots, x_n^{r_n})$; S has no output.

[a] This phase can be iterated. Specifically, R can compute the next κ bits of $\mathbf{t}^i$ and $\mathbf{u}^i$ (by applying G to get the next κ bits from the PRG for each of the seeds and using the next κ bits of its input in $\mathbf{r}$) and send the block of $\kappa \times \kappa$ bits to S (κ bits from each of $\mathbf{u}^1, \ldots, \mathbf{u}^\kappa$).

Proof: We first show that the protocol implements the $m \times OT_\ell$-functionality. Then, we prove that the protocol is secure where the sender is corrupted, and finally that it is secure when the receiver is corrupted.

Correctness. We show that the output of the receiver is $(x_1^{r_1}, \ldots, x_m^{r_m})$ in an execution of the protocol where the inputs of the sender are $((x_1^0, x_1^1), \ldots, (x_m^0, x_m^1))$ and the input of the receiver is $\mathbf{r} = (r_1, \ldots, r_m)$. Let $1 \leq j \leq m$, we show that $z_j = x_j^{r_j}$. We have two cases:

1. $\boldsymbol{r_j = 0}$**:** Recall that $\mathbf{q}_j = (r_j \cdot \mathbf{s}) \oplus \mathbf{t}_j$, and so $\mathbf{q}_j = \mathbf{t}_j$. Thus:
$$\begin{aligned} z_j &= y_j^0 \oplus H(\mathbf{t}_j) = x_j^0 \oplus H(\mathbf{q}_j) \oplus H(\mathbf{t}_j) \\ &= x_j^0 \oplus H(\mathbf{t}_j) \oplus H(\mathbf{t}_j) = x_j^0 \end{aligned}$$

2. $\boldsymbol{r_j = 1}$**:** In this case $\mathbf{q}_j = \mathbf{s} \oplus \mathbf{t}_j$, and so:
$$\begin{aligned} z_j &= y_j^1 \oplus H(\mathbf{t}_j) = x_j^1 \oplus H(\mathbf{q}_j \oplus \mathbf{s}) \oplus H(\mathbf{t}_j) \\ &= x_j^1 \oplus H(\mathbf{t}_j) \oplus H(\mathbf{t}_j) = x_j^1 \end{aligned}$$

Corrupted Sender. The view of the sender during the protocol contains the output from the $\kappa \times OT_\kappa$ invocation and the messages $\mathbf{u}^1, \ldots, \mathbf{u}^\kappa$. The simulator $\mathcal{S}_0$ simply outputs a uniform string $\mathbf{s} \in \{0,1\}^\kappa$ (which is the only randomness that S chooses in the protocol, and therefore w.l.o.g. can be interpreted as the random tape of the adversary), κ random seeds $\mathbf{k}_1^{s_1}, \ldots, \mathbf{k}_\kappa^{s_\kappa}$, which are chosen uniformly from $\{0,1\}^\kappa$, and κ random strings $\mathbf{u}^1, \ldots, \mathbf{u}^\kappa$, chosen uniformly from $\{0,1\}^m$. In the real execution, $(\mathbf{s}, \mathbf{k}_1^{s_1}, \ldots, \mathbf{k}_\kappa^{s_\kappa})$ are chosen in exactly the same way. Each value $\mathbf{u}^i$ for $1 \leq i \leq \kappa$ is defined as $G(\mathbf{k}_i^0) \oplus G(\mathbf{k}_i^1) \oplus \mathbf{r}$. Since $\mathbf{k}_i^{1-s_i}$ is unknown to S (by the security of the $\kappa \times OT_\kappa$ functionality), we have that $G(\mathbf{k}_i^{1-s_i})$ is indistinguishable from uniform, and so each $\mathbf{u}^i$ is indistinguishable from uniform. Therefore, the view of the corrupted sender in the simulation is indistinguishable from its view in a real execution.

Corrupted Receiver. The view of the corrupted receiver consists of its random tape and the messages $((y_1^0, y_1^1), \ldots, (y_m^0, y_m^1))$ only. The simulator $\mathcal{S}_1$ is invoked with the inputs and outputs of the receiver, i.e., $\mathbf{r} = (r_1, \ldots, r_m)$ and $(x_1^{r_1}, \ldots, x_m^{r_m})$. $\mathcal{S}_1$ then chooses a random tape ρ for the adversary (which determines the $\mathbf{k}_i^0, \mathbf{k}_i^1$ values), defines the matrix T, and computes $y_j^{r_j} = x_j^{r_j} \oplus H(\mathbf{t}_j)$ for $1 \leq j \leq m$. Then, it chooses each $y_j^{1-r_j}$ uniformly and independently at random from $\{0,1\}^\ell$. Finally, it outputs $(\rho, (y_1^0, y_1^1), \ldots, (y_m^0, y_m^1))$ as the view of the corrupted receiver.
We now show that the output of the simulator is indistinguishable from the view of the receiver in a real execution. If $r_j = 0$, then $\mathbf{q}_j = \mathbf{t}_j$ and thus $(y_j^0, y_j^1) = (x_j^0 \oplus H(\mathbf{t}_j), x_j^1 \oplus H(\mathbf{t}_j \oplus \mathbf{s}))$. If $r_j = 1$, $\mathbf{q}_j = \mathbf{t}_j \oplus \mathbf{s}$ and therefore $(y_j^0, y_j^1) = (x_j^0 \oplus H(\mathbf{t}_j \oplus \mathbf{s}), x_j^1 \oplus H(\mathbf{t}_j))$. In the simulation, the values $y_j^{r_j}$ are computed as $x_j^{r_j} \oplus H(\mathbf{t}_j)$ and therefore are identical to the real execution. It therefore remains to show that the values $(y_1^{1-r_1}, \ldots, y_m^{1-r_m})$ as computed in the real execution are indistinguishable from random strings as output in the simulation. As we have seen, in the real execution each $y_j^{1-r_j}$ is computed as $x_j^{1-r_j} \oplus H(\mathbf{t}_j \oplus \mathbf{s})$. Since H is a correlation robust function, it holds that:

$$\{\mathbf{t}_1, \ldots, \mathbf{t}_m, H(\mathbf{t}_1 \oplus \mathbf{s}), \ldots, H(\mathbf{t}_m \oplus \mathbf{s})\} \stackrel{c}{\equiv} \{U_{m \cdot \kappa + m \cdot \ell}\}$$

for random $\mathbf{s}, \mathbf{t}_1, \ldots, \mathbf{t}_m \in \{0,1\}^\kappa$, where U_a defines the uniform distribution over $\{0,1\}^a$ (see Definition A2). In the protocol we derive the values $\mathbf{t}_1, \ldots, \mathbf{t}_m$ by applying a pseudorandom generator G to the seeds $\mathbf{k}_1^0, \ldots, \mathbf{k}_\kappa^0$ and transposing the resulting matrix. We need to show that the values $H(\mathbf{t}_1 \oplus \mathbf{s}), \ldots, H(\mathbf{t}_m \oplus \mathbf{s})$ are still indistinguishable from uniform in this case. However, this follows from a straightforward hybrid argument (namely, that replacing truly random $\mathbf{t}^i$ values in the input to H with pseudorandom values preserves the correlation robustness of H). We conclude that the ideal and real distributions are computationally indistinguishable. ∎

5.4 Optimized OT Extension in Yao & GMW

The protocol described in §5.3 implements the $m \times OT_\ell$ functionality. In the following, we present further optimizations that are specifically tailored to the use of OT extensions in the secure computation protocols of Yao and GMW.

Correlated OT (C-OT) for Yao. Before proceeding to the optimization, let us focus for a moment on Yao's protocol [58] with the free-XOR [38] and point-and-permute [43] techniques.[6] Using this techniques, the sender does not choose all keys for all wires independently. Rather, it chooses a global random value $\delta \in_R \{0,1\}^{\kappa-1}$, sets $\Delta = \delta||1$, and for every wire w it chooses a random key $k_w^0 \in_R \{0,1\}^\kappa$ and sets $k_w^1 = k_w^0 \oplus \Delta$. Later in the protocol, the parties invoke OT extension to let the receiver obliviously obtain the keys associated with its inputs. This effectively means that, instead of having to obliviously transfer two fixed independent bit strings, the sender needs to transfer two random bit strings with a fixed correlation. We can utilize this constraint on the inputs in order to save additional bandwidth in the OT extension protocol. Recall that in the last step of Protocol 52 for OT extension, S computes and sends the messages $y_j^0 = x_j^0 \oplus H(\mathbf{q}_j)$ and $y_j^1 = x_j^1 \oplus H(\mathbf{q}_j \oplus \mathbf{s})$. In the case of Yao, we have that $x_j^0 = k_w^0$ and $x_j^1 = k_w^1 = k_w^0 \oplus \Delta$. Since k_w^0 is just a random value, S can set $k_w^0 = H(\mathbf{q}_j)$ and can send the *single* value $y_j = \Delta \oplus H(\mathbf{q}_j) \oplus H(\mathbf{q}_j \oplus s)$. R defines its output as $H(\mathbf{t}_j)$ if $r_j = 0$ or as $y_j \oplus H(\mathbf{t}_j)$ if $r_j = 1$. Observe that if $r_j = 0$, then $\mathbf{t}_j = \mathbf{q}_j$ and R outputs $H(\mathbf{q}_j) = x_j^0 = k_w^0$, as required. In contrast, when $r_j = 1$, it holds that $\mathbf{t}_j = \mathbf{q}_j \oplus \mathbf{s}$ and thus $y_j \oplus H(\mathbf{q}_j \oplus \mathbf{s}) = \Delta \oplus H(\mathbf{q}_j) = \Delta \oplus k_w^0 = k_w^1$, as required. Thus, in the setting of Yao's protocol when using the free-XOR technique, it is possible to save bandwidth. As the keys k_w^0, k_w^1 used in Yao are also of length κ, the bandwidth is reduced from 3κ bits that are transmitted in every iteration of the extension phase to 2κ bits, effectively reducing the bandwidth by one third. Proving the security of this optimization requires assuming that H is a random oracle, in order to "program" the output to be as derived from the OT extension. In addition, we define a different OT functionality, called *correlated OT (C-OT)*, that receives Δ and chooses the sender's inputs uniformly under the constraint that their XOR equals Δ. Since Yao's protocol uses random keys under the same constraint, the security of Yao's protocol remains unchanged when using this optimized OT extension. Note that by using the correlated input OT extension protocol, the server needs to garble the circuit *after* performing the OT extension; this order is also needed for the pipelining approach used in many implementations, e.g., [28,40,42]. We remark that this optimization can be used in the more general case where in each pair one of the inputs is chosen uniformly at random and the other input is computed as a function of the first. Specifically, the sender has different functions f_j for every $1 \leq j \leq m$, and receives random values x_j^0 as output from the extension protocol, which defines $x_j^1 = f_j(x_j^0)$. E.g., for Yao's garbled circuits protocol, we have $x_j^1 = f_j(x_j^0) = \Delta \oplus x_j^0$.

[6] Our optimization is also compatible with the garbled row reduction technique of [53].

Random-OT (R-OT) for GMW. When using OT extensions for implementing the GMW protocol, the efficiency can be improved even further. In this case, the inputs for S in every OT are *independent random* bits b^0 and b^1 (see §5.1 for how to evaluate AND gates using two random OTs). Thus, the sender can allow the random OT extension protocol (functionality) R-OT to determine *both* of its inputs randomly. This is achieved in the OT extension protocol by having S define $b^0 = H(\mathbf{q}_j)$ and $b^1 = H(\mathbf{q}_j \oplus s)$. Then, R computes b^{r_j} just as $H(\mathbf{t}_j)$. The receiver's output is correct because $\mathbf{q}_j = (r_j \cdot \mathbf{s}) \oplus \mathbf{t}_j$, and thus $H(\mathbf{t}_j) = H(\mathbf{q}_j)$ when $r_j = 0$, and $H(\mathbf{t}_j) = H(\mathbf{q}_j \oplus \mathbf{s})$ when $r_j = 1$. With this optimization, we obtain that the entire communication in the OT extension protocol consists only of the initial base OTs, together with the messages $\mathbf{u}^1, \ldots, \mathbf{u}^\kappa$, and there are *no* y_j messages. This is a dramatic improvement of bandwidth. As above, proving the security of this optimization requires assuming that H is a random oracle, in order to "program" the output to be as derived from the OT extension. In addition, the OT functionality is changed such that the sender receives both of its inputs from the functionality, and the receiver just inputs $\mathbf{r}$ (see [49, Fig. 26]).

Summary. The original OT extension protocol of [32] and our proposed improvements for $m \times OT_\ell$ are summarized in Tab. 2. We compare the communication complexity of R and S for m parallel 1-out-of-2 OT extensions of ℓ-bit strings, with security parameter κ (we omit the cost of the initial $\kappa \times OT_\kappa$). We also compare the assumption on the function H needed in each protocol, where CR denotes Correlation Robustness and RO denotes Random Oracle.

Protocol	**Applicability**	R → S	S → R	H
Original [32]	All applications	$2m\kappa$	$2m\ell$	CR
G-OT §5.3	All applications	$m\kappa$	$2m\ell$	CR
C-OT §5.4	only x_j^0 random	$m\kappa$	$m\ell$	RO
R-OT §5.4	x_j^0, x_j^1 random	$m\kappa$	0	RO

Table 2. Sent bits for sender S and receiver R for m 1-out-of-2 OT extensions of ℓ-bit strings and security parameter κ.

6 Experimental Evaluation

In the following, we evaluate the performance of our proposed optimizations. In §6.1 we compare our base OT protocol (§5.2) to the protocols of [46] and in §6.2 we evalute the performance of our algorithmic (§4) and protocol optimizations (§5.3 and §5.4) for OT extension.

Benchmarking Environment. We build upon the C++ OT extension implementation of [56] which implements the OT extension protocol of [32] and is based on the implementation of [10]. We use SHA-1 to instantiate the random oracle and

the correlation robust function and AES-128 in counter mode to instantiate the pseudo-random generator and the key derivation function. Our benchmarking environment consists of two 2.5 GHz Intel Core2Quad CPU (Q8300) Desktop PCs with 4 GB RAM, running Ubuntu 10.10 and OpenJDK 6, connected by a Gigabit LAN.

6.1 Base OTs

In the following, we compare the performance of the OT protocols of Naor and Pinkas [46] in the random oracle (RO) and standard (STD) model to our STD model OT protocol of §5.2 for different libraries. We either use finite field cryptography (FFC) (based on the GNU-Multiprecision library v.5.0.5) or elliptic curve cryptography (ECC) (based on the Miracl library v.5.6.1). We measure the time for performing κ 1-out-of-2 base OTs on κ-bit strings, for symmetric security parameter κ, using the key sizes from Tab. 1. The runtimes are shown in Tab. 3.

Security	**[46]-RO**	**[46]-STD**	**§5.2-STD**
GMP (FFC)			
Short [ms]	18 (±0.9)	99 (±0.6)	41 (±3.3)
Medium [ms]	107 (±3.4)	629 (±3.3)	352 (±18)
Long [ms]	288 (±7.9)	1,681 (±4.7)	1,217 (±47)
Miracl (ECC)			
Short [ms]	39 (±1.6)	178 (±0.3)	61 (±2.5)
Medium [ms]	82 (±2.9)	418 (±0.6)	137 (±5.0)
Long [ms]	138 (±5.0)	763 (±0.8)	239 (±7.5)

Table 3. Performance results and standard deviations for base OTs.

For the short term security parameter, FFC using GMP outperforms ECC using Miracl by factor 2 for all protocols. However, starting from a medium term security parameter, ECC becomes increasingly more efficient and outperforms FCC by more than factor 2 for the long term security parameter. For ECC, we can observe that [46]-RO is about 5-6 times faster than [46]-STD but only 2 times faster than our §5.2-STD protocol. For FFC, our §5.2-STD protocol becomes more inefficient with increasing security parameter, since the random sampling requires nearly full-range exponentiations as opposed to the subgroup exponentiations in [46]-RO and [46]-STD.

6.2 OT Extension

To evaluate the performance of OT extension, we measure the time for generating the random inputs for the OT extension protocol and the overall OT extension protocol execution on 10,000,000 1-out-of-2 OTs on 80-bit strings for the short-term security setting, excluding the times for the base OTs. Tab. 4 summarizes

Network	Orig [56] (1 T)	Orig [56] (2 T)	EMT §4.2 (1 T)	G-OT §5.3 (1 T)	C-OT §5.4 (1 T)	R-OT §5.4 (1 T)	R-OT §5.4 (2 T, §4.1)	R-OT §5.4 (4 T, §4.1)
LAN [s]	20.61 (±0.07)	16.57 (±0.33)	14.43 (±0.05)	13.92 (±0.07)	10.60 (±0.03)	10.00 (±0.02)	5.03 (±0.08)	2.62 (±0.05)
WiFi [s]	30.69 (±0.18)	30.42 (±0.20)	30.45 (±0.24)	29.36 (±0.26)	14.39 (±0.14)	14.22 (±0.12)	14.23 (±0.18)	14.23 (±0.22)

Table 4. Performance results and standard deviations for 10,000,000 1-out-of-2 OTs on 80-bit strings using our optimizations in §4 and §5.

the resulting runtimes for the original version without (Orig [56] (1 T)) and with pipelining (Orig [56] (2 T)), the efficient matrix transposition (EMT §4.2), the general protocol optimization (G-OT §5.3), the correlated OT extension protocol (C-OT §5.4), the random OT extension protocol (R-OT §5.4), as well as a two and four threaded version of R-OT (2 T and 4 T, cf. §4.1). The line (x T) denotes the number of threads, running on each party. Since our optimizations target both, the runtime as well as the amount of data that is transferred, we assume two different bandwidth scenarios: LAN (Gigabit Ethernet with 1 GBit bandwidth) and WiFi (simulated by limiting the available bandwidth to 54 MBit and the latency to 2 ms). As our experiments in Tab. 4 show, the LAN setting benefits from computation optimizations (as computation is the bottleneck), whereas the WiFi setting benefits from communication optimizations (as the network is the bottleneck). All timings are the average of 100 executions with one party acting as sender and the other as receiver. Note that each version includes all prior listed optimizations.

LAN setting. The original OT extension implementation of [56] has a runtime of 20.61 s without pipelining, which is reduced to only 80% (16.57 s) when using pipelining. Implementing the efficient matrix transposition of §4.2 decreases the runtime to 70% of the one-threaded original version (14.43 s) and already outperforms the pipelined version even though only one thread is used. The general improved OT extension protocol of §5.3 removes the need to generate the random matrix T, which reduces the runtime to 13.92 s. The C-OT extension of §5.4 decreases the runtime to 10.60 s, since the protocol generates the random input values for the sender. The R-OT extension of §5.4 further decreases the runtime to 10.00 s, since the last communication step is eliminated. Finally, the parallelized OT extension of §4.1 results in a nearly linear decrease in runtime to 50% (5.03 s) for two threads and to 26% (2.62 s) for four threads. Overall, using two threads, we decreased the runtime in the LAN setting by a factor of 3 compared to the two-threaded original implementation.

WiFi setting. In the WiFi setting, we observe that the one and two threaded original implementation is already slower compared to the LAN setting. Moreover, all optimizations that purely target the runtime have little effect, since the network has become the bottleneck. We therefore focus on the optimizations for the communication complexity. The G-OT optimization of §5.3 only

slightly decreases the runtime since both parties have the same up and download bandwidth and the channel from sender to receiver becomes the bottleneck (cf. Tab. 2).[7] The C-OT extension of §5.4 reduces the runtime by a factor of 2, corresponding to the reduced communication from sender to receiver which is now equal to the communication in the opposite direction. The R-OT extension of §5.4 only slightly decreases the runtime, since now the channel from receiver to sender has become the bottleneck. Finally, the multi-threading optimization of §4.1 does not reduce the runtime as the network is the bottleneck.

7 Application Scenarios

OT extension is the foundation for efficient implementations of many secure computation protocols, including Yao's garbled circuits implemented in the FastGC framework [28] and GMW implemented in the framework of [10, 56]. To demonstrate how both protocols benefit from our improved OT extensions, we apply our implementations to both frameworks and consider the following secure computation use-cases: Hamming distance (§7.1), set-intersection (§7.2), minimum (§7.3), and Levenshtein distance (§7.4). The overall performance results are summarized in Tab. 5 and discussed in §7.5. All experiments were performed under the same conditions as in §6 (LAN setting) using the random-oracle protocol of [46] as base OT. We extended the FastGC framework [28] to call our C++ OT implementation using the Java Native Interface (JNI). We stress that the goal of our performance measurements is to highlight the efficiency gains of our improved OT protocols, but not to provide a comparison between Yao's garbled circuits and the GMW protocol.

Implementation	Base-OTs	Hamming §7.1	Set-Intersect. §7.2	Minimum §7.3	Levenshtein §7.4
FastGC [28]	470 ms	149 ms (86.8 ms)	249 s (227 s)	1094 s (552 s)	265 min (148 ms)
FastGC [28] fixed with CPRG	482 ms	155 ms (87.6 ms)	253 s (227 s)	1106 s (554 s)	266 min (157 ms)
FastGC [28] with C-OT (4 T)	69 ms	85 ms (4.4 ms)	27 s (0.96 s)	593 s (15 s)	266 min (15 ms)
GMW [56]	142 ms	79 ms (46.5 ms)	1.91 s (1.34 s)	44 s (41 s)	—
GMW [56] with R-OT (4 T)	28 ms	30 ms (11.3 ms)	0.93 s (0.51 s)	21 s (19 s)	18 min (11 min)
AND gates	-	896	1,048,576	39,999,960	1,290,653,042
Client input bits	-	900	1,048,576	10,000,000	2,000

Table 5. Performance results for the frameworks of [28] and [56] with and without our optimized OT implementation. The time spent in the OT extensions is given in ().

[7] For shorter strings or if the channel would have a higher bandwidth from sender to receiver (e.g., a DSL link), the runtime would decrease already for the G-OT optimization.

7.1 Hamming Distance

The Hamming distance between two ℓ-bit strings is the number of positions that both strings differ in. Applications of secure Hamming distance computation include privacy-preserving face recognition [52] and private matching for cardinality threshold [33]. As shown in [28, 56], using a circuit-based approach is a very efficient way to securely compute the face recognition algorithm of [52] which uses $\ell = 900$. We use the compact Hamming distance circuit of [8] with size $\ell - HW(\ell)$ AND gates and ℓ input bits for the client, where $HW(\ell)$ is the Hamming weight of ℓ.

7.2 Set-Intersection

Privacy-preserving set-intersection allows two parties, each holding a set of σ-bit elements, to learn the elements they have in common. Applications include governmental law enforcement [11], sharing location data [47], and botnet detection [45]. Several Boolean circuits for computing the set-intersection were described and evaluated in [27]. The authors of [27] state that for small σ (up to $\sigma = 20$ in their experiments), the bitwise AND (BWA) circuit achieves the best performance. This circuit treats each element $e \in \{0,1\}^\sigma$ as an index to a bit-sequence $\{0,1\}^{2^\sigma}$ and denotes the presence of e by setting the respective bit to 1. The parties then compute the set-intersection as the bitwise AND of their bit-sequences. We build the BWA circuit for $\sigma = 20$, resulting in a circuit with $2^\sigma = 1{,}048{,}576$ AND gates and input bits for the client. To reduce the memory footprint of the FastGC framework [28], we split the overall circuit and the OTs on the input bits into blocks of size $2^{16} = 65{,}536$.

7.3 Secure Minimum

Securely computing the minimum of a set of values is a common building block in privacy-preserving protocols and is used to find best matches, e.g., for face recognition [15, 55], finger code authentication [2], or online marketplaces [10]. We use the scenario considered in [42] that securely computes the minimum of $N = 1{,}000{,}000$ $\ell = 20$-bit values, where each party holds 500,000 values. Using the minimum circuit construction of [37], our circuit has $2\ell N - 2\ell \approx 40{,}000{,}000$ AND gates and the client has $\frac{N}{2}\ell = 10{,}000{,}000$ input bits. We note that the performance of the garbled circuit implementation of [42] is about the same as that of FastGC [28] – their circuit has twice the size and takes about twice as long to evaluate. For the FastGC framework we again evaluate the overall circuit by iteratively computing the minimum of at most 2,048 values.

7.4 Levenshtein Distance

The Levenshtein distance denotes the number of operations that are needed to transform a string a into another string b using an alphabet of bit-size σ. It can be used for privacy-preserving matching of DNA and protein-sequences [28, 34].

We use the same circuit and setting as [28] with $\sigma = 2$ to compare strings a and b of size $|a| = 2{,}000$ and $|b| = 10{,}000$. The resulting circuit has 1.29 billion AND gates and $\sigma|a| = 4{,}000$ input bits for the client. The GMW framework of [56] was not able to evaluate the Levenshtein circuit since their OT extension implementation tries to process all OTs at once and their framework tries to store the whole circuit in memory, thereby exceeding the available memory of our benchmarking environment. Hence, we changed their underlying circuit structure to support large-scale circuits by deleting gates that were used and building the circuit iteratively.

7.5 Discussion

We discuss the results of our experiments in Tab. 5 next. For the FastGC framework [28], our improved OT extension implementation written in C++ and using 4 threads is more than one order of magnitude faster than the corresponding single-threaded Java routine of the original implementation. The improvements on total time depend on the ratio between the number of client inputs and the circuit size: for circuits with many client inputs (§7.1, §7.2, §7.3), we obtain a speedup by factor 2 to 9, whereas for large circuits with few inputs (§7.4) the improvement for OTs has a negligible effect on the total runtime. To further improve the runtime of large circuits, a faster engine for circuit garbling, e.g., [6], could be combined with our improved OT implementation. For the GMW framework [56], the total runtime is dominated by the time for performing OT extension, which we reduce by factor 2.

Acknowledgements We thank David Evans and the anonymous reviewers of ACM CCS for their helpful comments on our paper. The first two authors were funded by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 239868. The third and fourth author were supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

References

1. G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *Computer and Communications Security (CCS'13)*. ACM, 2013. To appear.
2. M. Barni, T. Bianchi, D. Catalano, M. Di Raimondo, R. Donida Labati, P. Failla, D. Fiore, R. Lazzeretti, V. Piuri, F. Scotti, and A. Piva. Privacy-preserving fingercode authentication. In *Multimedia and Security (MM&SEC'10)*, pages 231–240. ACM, 2010.
3. D. Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology – CRYPTO'91*, volume 576 of *LNCS*, pages 420–432. Springer, 1991.

4. D. Beaver. Correlated pseudorandomness and the complexity of private computations. In *Symposium on Theory of Computing (STOC'96)*, pages 479–488. ACM, 1996.
5. M. Bellare, S. Goldwasser, and D. Micciancio. "pseudo-random" number generation within cryptographic algorithms: The DDS case. In *Advances in Cryptology – CRYPTO'97*, volume 1294 of *LNCS*, pages 277–291. Springer, 1997.
6. M. Bellare, V. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *Symposium on Security and Privacy*, pages 478–492. IEEE, 2013.
7. A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multiparty computation. In *Computer and Communications Security (CCS'08)*, pages 257–266. ACM, 2008.
8. J. Boyar and R. Peralta. The exact multiplicative complexity of the Hamming weight function. *Electronic Colloquium on Computational Complexity (ECCC'05)*, (049), 2005.
9. R. Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.
10. S. G. Choi, K.-W. Hwang, J. Katz, T. Malkin, and D. Rubenstein. Secure multiparty computation of Boolean circuits with applications to privacy in on-line marketplaces. In *Cryptographers' Track at the RSA Conference (CT-RSA'12)*, volume 7178 of *LNCS*, pages 416–432. Springer, 2012.
11. E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security (FC'10)*, volume 6052 of *LNCS*, pages 143–159. Springer, 2010.
12. C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: An efficient and scalable protocol. In *Computer and Communications Security (CCS'13)*. ACM, 2013. To appear.
13. Y. Ejgenberg, M. Farbstein, M. Levy, and Y. Lindell. SCAPI: The secure computation application programming interface. *IACR Cryptology ePrint Archive*, 2012:629, 2012.
14. J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, C-21(7):801–803, 1972.
15. Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In *Privacy Enhancing Technologies Symposium (PETS'09)*, volume 5672 of *LNCS*, pages 235–253. Springer, 2009.
16. S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communmunications of the ACM*, 28(6):637–647, 1985.
17. K. Frikken, M. Atallah, and C. Zhang. Privacy-preserving credit checking. In *Electronic Commerce (EC'05)*, pages 147–154. ACM, 2005.
18. O. Goldreich. *Foundations of Cryptography*, volume 2: Basic Applications. Cambridge University Press, 2004.
19. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *Symposium on Theory of Computing (STOC'87)*, pages 218–229. ACM, 1987.
20. S. D. Gordon, J. Katz, V. Kolesnikov, F. Krell, T. Malkin, M. Raykova, and Y. Vahlis. Secure two-party computation in sublinear (amortized) time. In *Computer and Communications Security (CCS'12)*, pages 513–524. ACM, 2012.
21. D. Harnik, Y. Ishai, E. Kushilevitz, and J. B. Nielsen. OT-combiners via secure computation. In *Theory of Cryptography (TCC'08)*, volume 4948 of *LNCS*, pages 393–411. Springer, 2008.

22. J. Håstad and A. Shamir. The cryptographic security of truncated linearly related variables. In *Symposium on Theory of Computing (STOC'85)*, pages 356–362. ACM, 1985.
23. W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *Computer and Communications Security (CCS'10)*, pages 451–462. ACM, 2010.
24. W. Henecka and T. Schneider. Faster secure two-party computation with less memory. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'13)*, pages 437–446. ACM, 2013.
25. A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *Computer and Communications Security (CCS'12)*, pages 772–783. ACM, 2012.
26. Y. Huang, P. Chapman, and D. Evans. Privacy-preserving applications on smartphones. In *Hot topics in security (HotSec'11)*. USENIX, 2011.
27. Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Network and Distributed Security Symposium (NDSS'12)*. The Internet Society, 2012.
28. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *Security Symposium*. USENIX, 2011.
29. Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *Symposium on Security and Privacy*, pages 272–284. IEEE, 2012.
30. Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient privacy-preserving biometric identification. In *Network and Distributed Security Symposium (NDSS'11)*. The Internet Society, 2011.
31. Intelligence Advanced Research Projects Activity (IARPA). Security and Privacy Assurance Research (SPAR) Program, 2010.
32. Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.
33. A. Jarrous and B. Pinkas. Secure hamming distance based computation and its applications. In *Applied Cryptography and Network Security (ACNS'09)*, volume 5536 of *LNCS*, pages 107–124. Springer, 2009.
34. S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *Symposium on Security and Privacy*, pages 216–230. IEEE, 2008.
35. F. Kerschbaum. Automatically optimizing secure computation. In *Computer and Communications Security (CCS'11)*, pages 703–714. ACM, 2011.
36. V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *Advances in Cryptology – CRYPTO'13 (2)*, volume 8043 of *LNCS*, pages 54–70. Springer, 2013.
37. V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology And Network Security (CANS'09)*, volume 5888 of *LNCS*, pages 1–20. Springer, 2009.
38. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages and Programming (ICALP'08)*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.
39. H. Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology – CRYPTO'10*, volume 6223 of *LNCS*, pages 631–648. Springer, 2010.
40. B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Security Symposium*. USENIX, 2012.

41. P. MacKenzie, A. Oprea, and M. K. Reiter. Automatic generation of two-party computations. In *Computer and Communications Security (CCS'03)*, pages 210–219. ACM, 2003.
42. L. Malka. VMCrypt - modular software architecture for scalable secure computation. In *Computer and Communications Security (CCS'11)*, pages 715–724. ACM, 2011.
43. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *Security Symposium*, pages 287–302. USENIX, 2004.
44. A. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.
45. S. Nagaraja, P. Mittal, C.-Y. Hong, M. Caesar, and N. Borisov. Botgrep: Finding P2P bots with structured graph analysis. In *Security Symposium*, pages 95–110. USENIX, 2010.
46. M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *ACM-SIAM Symposium On Discrete Algorithms*, SODA '01, pages 448–457. Society for Industrial and Applied Mathematics, 2001.
47. A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *Network and Distributed Security Symposium (NDSS'11)*. The Internet Society, 2011.
48. J. B. Nielsen. Extending oblivious transfers efficiently - how to get robustness almost for free. Cryptology ePrint Archive, Report 2007/215, 2007.
49. J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO'12*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.
50. V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Symposium on Security and Privacy*, pages 334–348. IEEE, 2013.
51. NIST. NIST Special Publication 800-57, Recommendation for Key Management Part 1: General (Rev. 3). Technical report, 2012.
52. M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI - a system for secure face identification. In *Symposium on Security and Privacy*, pages 239–254. IEEE, 2010.
53. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT'09*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.
54. M. O. Rabin. *How to exchange secrets with oblivious transfer*, TR-81 edition, 1981. Aiken Computation Lab, Harvard University.
55. A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In *International Conference on Information Security and Cryptology (ICISC'09)*, volume 5984 of *LNCS*, pages 229–244. Springer, 2009.
56. T. Schneider and M. Zohner. GMW vs. Yao? Efficient secure two-party computation with low depth circuits. In *Financial Cryptography and Data Security (FC'13)*, LNCS. Springer, 2013.
57. A. Schröpfer and F. Kerschbaum. Demo: secure computation in JavaScript. In *Computer and Communications Security (CCS'11)*, pages 849–852. ACM, 2011.
58. A. C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science (FOCS'86)*, pages 162–167. IEEE, 1986.

A Definitions

We let κ denote the security parameter. A function $\mu(\cdot)$ is negligible if for every positive polynomial $p(\cdot)$ and all sufficiently large n it holds that $\mu(n) < 1/p(n)$. A distribution ensemble $X = \{X(a,n)\}_{a\in\mathcal{D}_n, n\in\mathbb{N}}$ is an infinite sequence of random variables indexed by $a \in \mathcal{D}_n$ and $n \in \mathbb{N}$. Two distribution ensembles X, Y are *computationally indistinguishable*, denoted $X \stackrel{c}{\equiv} Y$ if for every non-uniform polynomial time algorithm D there exists a negligible function $\mu(\cdot)$ such that for every n, and every $a \in \mathcal{D}_n$:

$$|\Pr[D(X(a,n),a,n) = 1] - \Pr[D(Y(a,n),a,n]| \leq \mu(n).$$

Key Derivation Function. The following definition is an adaptation of the general definition of [39] for the case of the DDH problem. Intuitively, the adversary should not be able to distinguish between an output of the KDF function and a uniform string. Let $\mathsf{Gen}(1^\kappa)$ be a function that produces a group $(\mathbb{G}, q, g)$ for which the DDH problem is believed to be hard. We define:

Definition A1 (Key-Derivation Function) *A key derivation function* KDF *with ℓ-bit output is said to be* secure with respect to DDH *if for any* PPT *attacker $\mathcal{A}$ there exists a negligible function $\mu(\cdot)$ such that:*

$$\begin{aligned}|\Pr[\mathcal{A}(\mathbb{G}, q, g, g^r, h, \mathsf{KDF}(h^r)) = 1] \\ - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^r, h, z) = 1]| \leq \mu(\kappa)\end{aligned}$$

where $(\mathbb{G}, q, g) = \mathsf{Gen}(1^\kappa)$, r is distributed uniformly in $\mathbb{Z}_q$ and z is distributed uniformly in $\{0,1\}^\ell$.

Correlation Robust Function. We present a definition for correlation robust function. The definition is based on the definition in [32].

Definition A2 *[Correlation Robustness] An efficiently computable function $H : \{0,1\}^\kappa \to \{0,1\}^\ell$ is said to be* correlation robust *if it holds that:*

$$\{\mathbf{t}_1, \ldots, \mathbf{t}_m, H(\mathbf{t}_1 \oplus \mathbf{s}), \ldots, H(\mathbf{t}_m \oplus \mathbf{s})\} \stackrel{c}{\equiv} \{U_{m\cdot\kappa+m\cdot\ell}\}$$

where $\mathbf{t}_1, \ldots, \mathbf{t}_m, \mathbf{s}$ are chosen uniformly and independently at random from $\{0,1\}^\kappa$, and $U_{m\cdot\kappa+m\cdot\ell}$ is the uniform distribution over $\{0,1\}^{m\cdot\kappa+m\cdot\ell}$.

Secure Two-Party Computation. We give a formal definition for security of a two party protocol in the presence of a semi-honest adversary. The definition is the standard definition, see [9,18].The view of the party P_0 during an execution of a protocol π on inputs (x,y), denoted $\text{VIEW}_i^\pi(x,y)$, is defined to be $(x, r; \boldsymbol{m})$ where x is P_0's private input, r its internal coin tosses, and $\boldsymbol{m}$ are the messages it has received in the execution. The view of P_1 is defined analogously. Let $\text{OUTPUT}^\pi(x,y)$ denote the output pair of both parties in a real execution of the protocol. We are now ready to security definition:

Definition A3 *Let $f : (\{0,1\}^*)^2 \to (\{0,1\}^*)^2$ be a (possible randomized) two–party functionality, and let $f_i(x,y)$ denotes the ith element of $f(x,y)$. Let π be a protocol. We say that π* privately–computes *f if for every $(x,y) \in (\{0,1\}^*)^2$:* $\text{OUTPUT}^\pi(x,y) = f(x,y)$ *and there exists a pair of probabilistic polynomial-time* PPT *algorithms $\mathcal{S}_0, \mathcal{S}_1$:*

$$\{\mathcal{S}_0(x, f_0(x,y)), f(x,y)\}_z \stackrel{c}{\equiv} \{\text{VIEW}_0^{\Pi}(x,y), \text{OUTPUT}^\pi(x,y)\}_z$$
$$\{\mathcal{S}_1(y, f_1(x,y)), f(x,y)\}_z \stackrel{c}{\equiv} \{\text{VIEW}_1^{\Pi}(x,y), \text{OUTPUT}^\pi(x,y)\}_z$$

where $z = (x,y) \in (\{0,1\}^*)^2$.

In case the function f is *deterministic* (like in the OT functionality), there is no need to consider the joint distribution of the outputs and the view, and it is enough to show that the output of the simulator $\mathcal{S}_i$ is indistinguishable from the view of the party P_i.

B Multiplication Triple Protocol

In this section, we show that the protocol presented in §5.1 privately computes the multiplication triple functionality.

First, we consider the f^{ab} functionality. The protocol implements the functionality since any random $(b,v),(a,u)$, for which $ab = u \oplus v$, can be written as $(b,v) = (x_0 \oplus x_1, x_0)$ and $(a,u) = (a, ab \oplus v) = (a, x_a)$, since it holds that: $ab \oplus v = ab \oplus x_0 = a(x_0 \oplus x_1) \oplus x_0 = x_a$. The inputs and outputs of each party fully determine its view, and therefore simulators are trivial and just re–arrange their inputs. Consistency of the generated view with the output of the parties holds trivially.

We turn to the multiplication triple functionality. It is easy to verify that the protocol implements the functionality. Regarding simulation, a simulator $\mathcal{S}_0$ is given (a_0, b_0, c_0), chooses random u_0 and defines: $v_0 = c_0 \oplus a_0 b_0 \oplus u_0$. Since u_0, v_0 are random and hidden from the distinguisher, the view is consistent with (a_1, b_1, c_1). A simulator for $\mathcal{S}_1$ works the same, and security holds from the same reasoning (i.e., $v_1 = a_0 b_1 \oplus u_0$ is random since u_0 is hidden from the distinguisher, and v_1 is fully determined from c_1, a_1, b_1, u_1).