Oblivious Outsourcing of Garbled Circuit Generation

Florian Kerschbaum
SAP
Karlsruhe, Germany
florian.kerschbaum@sap.com

ABSTRACT

Yao's garbled circuit technique is often used in outsourced computation. Current approaches divide the computation to two or more servers. The assumption is that the servers collaborate in offering the outsourced computation, but do not share data. This seems somewhat paradoxical in the current cloud economy. We therefore propose *oblivious outsourcing* where one server is unaware that other servers are involved. We present a garbled circuit generation outsourcing scheme built on lattice-based cryptography implementing this model. Our scheme does not increase the cost of circuit evaluation, but achieves a speed up of 98% (factor 55) for circuit generation.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*

Keywords

Secure Two-Party Computation; Garbled Circuits; Outsourcing; Homomorphism

1. INTRODUCTION

Yao's garbled circuit technique [32] offers a great opportunity to securely outsource sensitive computations. Unfortunately encryption of garbled circuits is less efficient than performing the computation locally. Therefore a number of protocols for outsourcing garbled circuits have been proposed [6, 7, 8, 9, 11, 19, 28].

Their common approach is to outsource the computation to two or more servers. On the one hand, in their setup the servers need to collaborate, e.g., by communicating according to a certain protocol, and agree on the function to be computed. In the cloud computing economy this implies jointly offering a service for the computation. On the other hand, in their setup the servers need to be mutually distrustful. If the servers collude and exchange secret information all confidentiality for the client is lost.

SAC'15, April 13–17 2015, Salamanca, Spain.

ACM 978-1-4503-3196-8/15/04 ...$15.00.
http://dx.doi.org/10.1145/2695664.2695665.

This paradoxical setup is very unlikely to be implemented in the cloud computing market in the near future. A customer who buys a service from two collaborating entities, but trusts that they will not exchange certain information, is probably hard to find. We therefore propose a different model for outsourcing garbled circuits.

In our model the servers offer independent services. They operate independently on the cloud market competing for customers. Two servers offer a service for encrypting a garbled circuit, but do not need to collaborate or otherwise exchange messages. One server offers a service for evaluating a garbled circuit. Neither of the servers ever directly communicate nor do they need to be aware of the existence of the other. They remain oblivious; hence oblivious outsourcing.

We achieve the following three types of obliviousness:

- *Input and Output Obliviousness*: Neither server will be able to infer anything about the client's input or output. This is the standard definition of obliviousness of garbled circuits.
- *Function Obliviousness*: The encrypting servers will not be able to infer anything about the function that is computed (except its complexity). We emphasize that the leaked side information for the encrypting servers is less than that for the evaluating server.
- *Outsourcing Obliviousness*: The evaluating server will not be able to decide whether another server was used to encrypt the circuit.

Our proposed scheme allows to implement this model with little overhead. We base on recent results in lattice-based cryptography and are as fast as current standard symmetric cryptography. Yet, our protocol still requires the client to perform work on the order of the function's complexity. This is a lower bound and a direct consequence of outsourcing obliviousness, i.e., no protocol with outsourcing obliviousness can be more efficient. Yet, our protocol significantly improves the constants and we are able to achieve an improvement factor of up to 55 in our implementation when generating a garbled circuit. Our protocols is therefore very suitable for generating garbled circuits in computationally weak devices, such as sensors or mobile phones. Furthermore, outsourcing sensitive computations can still be useful despite an encryption cost on the order of the function's complexity, in case of computations on joint inputs (see Section 7).

2. RELATED WORK

Yao introduced the garbled circuit technique in 1986 [32]. It has been extended, implemented, used and analyzed in a number of ways.

Outsourcing.

The first proposal for outsourcing garbled circuits by Feige et al. used a non-oblivious server [11]. There is only one server, but this server learns all inputs and outputs. It can therefore not be used in oblivious outsourcing. Kamara et al. recently formalized, improved and implemented the protocols for this model [19].

Later proposals, beginning with the work of Naor et al. [28] split the generation and evaluation between two servers and the clients providing input to both. This implements oblivious outsourcing where the servers learn nothing about input or output. Nevertheless this model suffers from economic drawbacks where the servers need to collaborate to offer the service, but not collude to exchange the data. Bugiel et al. recently built on this model [6].

Often only the circuit evaluation needs to be outsourced in order to offer a performance improvement already. Carter et al. recently proposed this for mobile devices [7, 8, 9]. We build on this model, but further allow circuit generation to be also outsourced, such that the second party can be also a computationally weak device. This significantly extends the reach of their approach.

An important application of outsourcing is the computation of collaborative statistics. Specific protocols can be found in [17, 20, 21].

Implementation.

The first implementation of garbled circuits (and secure computation) was provided by Malkhi et al. in the FairPlay project [25]. Since then a number of performance improvements have been achieved. Special circuit garbling techniques, such as the free-XOR technique [22] or garbled row reduction [31] improve the construction. Pipelining of the operations leads to effective parallelization of the parties [16]. This last implementation has itself already been tested on mobile devices *without outsourcing* [15]. On the one hand, one recent implementation by Bellare et al. [3] shows that improving the speed of cryptographic functions improves the overall performance. On the other hand, another recent implementation by Henecka and Schneider is bound by the network speed [14]. We did not implement a full garbling scheme, but only the functions necessary for our outsourcing scheme. The focus of our work is also not to improve the performance of garbled circuits, but to show that garbled circuit generation can be obliviously and efficiently outsourced.

Security.

The first security proof of Yao's protocol was given by Lindell and Pinkas [24]. A very good, recent formalization of garbled circuits was given by Bellare et al. [4]. We build on their security models and extend them to oblivious outsourcing.

Garbled circuits can be used with oblivious transfer in order to implement secure computation. In secure computation often a distinction is made between semi-honest and malicious security [12]. Commonly the cut-and-choose technique is used to build protocol using garbled circuits secure in the malicious model. It has already been shown that this can be efficiently implemented by Pinkas et al. [31] and recently for large circuits by Kreuter et al. [23]. We omit this distinction, since we focus on the security of the garbling scheme itself and therefore follow the definitions by Bellare et al. Nevertheless all of the techniques described in the related work above use garbled circuit generation and are therefore compatible with our approach. They would all benefit from its performance improvements. We outline an example in the semi-honest model in Section 7, but stress that many further use cases – also in stronger security models – are possible.

Jawurek at el. recently showed that garbled circuits can also be used to construct efficient zero-knowledge proofs [18]. Again, their technique would benefit from our outsourcing.

3. GARBLED CIRCUITS

3.1 Definitions

We follow the definition of garbled circuits by Bellare et al. [4]. A garbling scheme is defined by a tuple $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev})$ such that:

- The garbled circuit generation function Gb is a randomized algorithm that on input of a security parameter 1^κ and the description of a Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, with $n = \text{poly}(\kappa)$, $|f| = \text{poly}(\kappa)$ and $m = \text{poly}(\kappa)$ outputs a triple of strings (GC, e, d).
- The encoding function En is a deterministic function that uses e to map an input x to a *garbled input* X.
- The evaluation function Ev is a deterministic function that evaluates a garbled circuit GC on an encoded input X and produces an encoded output Z'.
- The decoding function De, using the string d, decodes the encoded output Z' into a plaintext output z.

The correctness and security definitions that we provide are very similar to those given by Bellare et al.

Definition 1 (Correctness). *Let $\mathcal{G}$ be a garbling scheme described as above. We say that $\mathcal{G}$ enjoys* correctness *if for all functions $f : \{0,1\}^n \to \{0,1\}^m$ with $n = \text{poly}(\kappa)$, $m = \text{poly}(\kappa)$, $|f| = \text{poly}(\kappa)$ and all $x \in \{0,1\}^n$ the following probability*

$$\Pr\left(\begin{array}{l}\mathsf{De}(d, \mathsf{Ev}(GC, \mathsf{En}(e, x)) \neq f(x) : \\ (GC, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)\end{array}\right)$$

is negligible in κ.

Intuitively, Definition 1 says that it is possible for the client to recover the result $f(x)$ from a server that is honestly evaluating a generated circuit with input keys corresponding to a value x.

Definition 2 (Input and Output Obliviousness). *Let $\mathcal{G}$ be a garbling scheme described as above. We say that $\mathcal{G}$ enjoys* input and output obliviousness *if for all polynomial-time, probabilistic adversaries $\mathcal{A}$, all functions $f : \{0,1\}^n \to \{0,1\}^m$ with $n = \text{poly}(\kappa)$, $m = \text{poly}(\kappa)$, $|f| = \text{poly}(\kappa)$ and all inputs $x_0, x_1 \in \{0,1\}^n$ the following probability*

$$\Pr\left(\begin{array}{l}\mathcal{A}(GC,X)=b:\\ X\leftarrow \mathsf{En}(e,x_b)\\ (GC,e,d)\leftarrow \mathsf{Gb}(1^\kappa,f)\\ b\xleftarrow{R}\{0,1\}\end{array}\right)$$

is negligible in κ.

Intuitively, Definition 2 says that it is not possible for the server to recover the input x or output $f(x)$ from the information sent to it, i.e. it remains oblivious.[1]

3.2 Implementation

We follow the original construction of Yao [32] which has been adopted in several recent implementations [14, 16, 25, 31]. We are going to describe in detail the function $\mathsf{Gb}(1^\kappa, f)$, since it is crucial for the understanding of our scheme. For the other functions of a garbling scheme we refer the reader to the literature. W.l.o.g. we assume that the function Gb is executed by the client.

Circuit Preparation.

First, the client translates the function f into a circuit C. This circuit C consists of gates $G = g_1, \ldots, g_{|f|}$ and wires $W = w_1, \ldots, w_{|f|+n}$.

For simplicity of exposition we focus on one binary gate g_k. This gate g_k has input wires w_i and w_j and output wire w_k. It implements a Boolean, binary function (its type) τ_k, e.g., logical and or exclusive-or. Each gate consists of four entries – one for each possible input combination.

The wiring information of gate g_k is the tuple i, j, k. The wiring information of circuit C is the union of the wiring information of all gates $g \in G$. The type information of gate g_k is the tuple k, τ_k. The type information of circuit C is the union of the type information of all gates $g \in G$.

Key Generation.

The client generates for each wire w_k $(1 \le k \le |f| + n)$, and each bit $b \in \{0, 1\}$ a κ-bit string v_k^b. Let $\hat{v}_k^b$ be the least significant bit of v_k^b. The client ensures that $\hat{v}_k^0 \neq \hat{v}_k^1$, i.e., the least significant bits of the keys of each wire differ.

Encryption.

Let H be a cryptographic, collision-resistant, one-way hash function, e.g., SHA-3, mapping onto $\mathbb{Z}_q \supseteq \{0, 1\}^\kappa$. Let $||$ denote concatenation. Gate g_k consists of four entries of $x \in \{0, 1\}$, $y \in \{0, 1\}$ and $z = \tau_k(x, y)$. The client encrypts each entry x, y, z as follows[2]:

$$\hat{v}_i^x \ , \ \hat{v}_j^y \ , \ v_k^z + \mathsf{H}(v_i^x || v_j^y) \bmod q$$

The encryption of gate g_k is the union of all four entries and k.[3] The encryption of the circuit C is the union of all gates $g \in G$.

[1]Contrary to Bellare et al. we do not include function obliviousness, since the evaluating server in cloud computing most certainly knows the function. Furthermore, we are going to define a stricter side information function for the encrypting servers.

[2]We deal with gates with the same input wires by different hash functions.

[3]In a practical implementation this information can be further compressed. The least significant bits can be omitted by garbling, i.e., sorting, the entries. One entry can be omitted using the garbled row reduction technique [31].

4. AJTAI'S HASH FUNCTION

4.1 Lattice-Based Cryptography

Ajtai's seminal paper [1] started the field of lattice-based cryptography. An excellent introduction of lattice-based cryptography is given by Micciancio and Regev in [27]. We extract some material for the reader's understanding. A *lattice* is defined as the set of all integer combinations

$$\mathcal{L}(\vec{b_1}, \ldots, \vec{b_\nu}) = \left\{ \sum_{i=1}^{\nu} x_i \vec{b_i} : x_i \in \mathbb{Z} \right\}$$

of ν linearly independent vectors $\vec{b_1}, \ldots, \vec{b_\nu}$ in $\mathbb{R}^\nu$. The set of vectors $\vec{b_1}, \ldots, \vec{b_\nu}$ is called a basis for the lattice. A basis can be represented by the matrix $\vec{B} = [\vec{b_1}, \ldots, \vec{b_\nu}] \in \mathbb{R}^{\nu \times \nu}$ having the basis vectors as columns. Using matrix notation, the lattice generated by a matrix $\vec{B} \in \mathbb{R}^{\nu \times \nu}$ can be defined as $\mathcal{L}(\vec{B}) = \{\vec{B}\vec{x} : \vec{x} \in \mathbb{Z}^\nu\}$, where $\vec{B}\vec{x}$ is the usual matrix-vector multiplication.

Let N be a norm for the lattice $\mathcal{L}(\vec{B})$, e.g., the L^2 norm – Euclidean Distance in conventional geometry. Let $\vec{\lambda}$ be the shortest, non-zero vector in $\mathcal{L}(\vec{B})$ according to norm N. Then, the Approximate Shortest Vector Problem (aSVP)is assumed to be computational hard.

DEFINITION 3 (γ-aSVP). *Given $\vec{B}$ and norm N find a non-zero vector in $\mathcal{L}(\vec{B})$ of length at most $\gamma N(\vec{\lambda})$.*

4.2 Construction

Ajtai's hash function has the nice property that it is provably secure if the γ-aSVP is hard. Ajtai proved that inversion of the function implied solving *any* instance of the ν^c-aSVP problem [1]. Goldreich proved the function is also collision-resistant, a stronger property than one-wayness [13].

Ajtai's hash function is parameterized by integers ν, μ, q and δ. The hash function is keyed by a matrix $\vec{A}$ chosen uniformly from $\mathbb{Z}_q^{\nu \times \mu}$. The hash function $\mathsf{H}_{\vec{A}} : \{0, \ldots, \delta - 1\}^\mu \to \mathbb{Z}_q^n$ is given by

$$\mathsf{H}_{\vec{A}}(\vec{y}) = \vec{A}\vec{y} \bmod q.$$

Note that the key can be known to the adversary for the hash function to remain secure.

A possible instantiation of the parameters is $\delta = 2$, $q = \nu^2$, and $\mu \approx 2\nu \log q / \log \delta$ The choice of ν then determines the security of the hash function. In terms of bits, the function maps $\mu \log \delta$ bits into $\nu \log q$ bits. Therefore we achieve a compression of roughly 2.

An important property of Ajtai's hash function is that it is homomorphic for addition if we let $\vec{y}, \vec{z} \in \mathbb{Z}_q^\mu$.

$$\begin{array}{rl} & \mathsf{H}_{\vec{A}}(\vec{y}) + \mathsf{H}_{\vec{A}}(\vec{z}) \\ = & \vec{A}\vec{y} + \vec{A}\vec{z} \\ = & \vec{A}(\vec{y} + \vec{z}) \\ = & \mathsf{H}_{\vec{A}}(\vec{y} + \vec{z}) \end{array}$$

5. OUTSOURCING

The idea of our construction is to replace the hash function in the garbled circuit generation with Ajtai's hash function. First, we create $t \ge 2$ shares of the circuit. Each share is then encrypted and the results combined into a garbled circuit.

5.1 Definition

Let $\mathcal{G}$ be a garbling scheme as defined above. We replace the function Gb with a garbling outsourcing scheme. A garbling outsourcing scheme $\mathcal{O}$ is defined by a tuple $\mathcal{O} = (\mathsf{Sh}, \mathsf{Ha}, \mathsf{Cb})$ such that

1. The share generation function Sh is a randomized algorithm that on input of a security parameter 1^κ and the description of a Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, with $n = \text{poly}(\kappa)$, $|f| = \text{poly}(\kappa)$ and $m = \text{poly}(\kappa)$ outputs a tuple of strings $(CS_1, \ldots, CS_t, e, d)$.
2. The share encryption function Ha is a deterministic algorithm that on input of share CS_s outputs a garbled share GS_s.
3. The share combination function Cb is a deterministic algorithm that on input of shares $GS_1, \ldots, GS_t$ outputs a string GC.

The following security definitions of an outsourcing scheme extend the ones for garbled circuits. Every garbled circuit (share) reveals some side information, most notably some information about the function, at least its circuit complexity $|f|$. Let $\Phi_\mathcal{O}$, $\Phi_\mathcal{G}$ be the function that computes the side information of a garbling or garbling outsourcing scheme, respectively. We use the computed function f as a parameter to either of the side information functions.

Definition 4 (Function Obliviousness). *Let $\mathcal{O}$ be a garbling outsourcing scheme described as above. We say that $\mathcal{O}$ enjoys* function obliviousness *if for all polynomial-time, probabilistic adversaries $\mathcal{A}$, all functions f_0, f_1 with $\Phi_\mathcal{O}(f_0) = \Phi_\mathcal{O}(f_1)$, $|f| = \text{poly}(\kappa)$ and all $s : 1 \le s \le t$ the following probability*

$$\Pr\left(\begin{array}{l}\mathcal{A}(CS_s) = b : \\ (CS_1, \ldots, CS_t, e, d) \leftarrow \mathsf{Sh}(1^\kappa, f_b) \\ b \xleftarrow{R} \{0,1\}\end{array}\right)$$

is negligible in κ.

Intuitively, Definition 4 says that it is not possible for the encrypting server to determine the computed function from the information sent to it. We specify the side information function in detail in our implementation. Furthermore, the encrypting server clearly remains oblivious to the client's input and output, since it receives no information about it.

Definition 5 (Outsourcing Obliviousness). *Let $\mathcal{O}$ be a garbling outsourcing scheme and $\mathcal{G}$ be a corresponding garbling scheme as described above. We say that $\mathcal{O}$ enjoys* outsourcing obliviousness *if for all polynomial-time, probabilistic adversaries $\mathcal{A}$ and all functions $f : \{0,1\}^n \to \{0,1\}^m$ with $n = \text{poly}(\kappa)$, $m = \text{poly}(\kappa)$, $|f| = \text{poly}(\kappa)$ the following probability*

$$\Pr\left(\begin{array}{l}\mathcal{A}(GC, e, d) = b : \\ (GC, e, d) \leftarrow (GC^b, e^b, d^b) \\ (GC^0, e^0, d^0) \leftarrow \mathsf{Gb}(1^\kappa, f) \\ (GC^1) \leftarrow \mathsf{Cb}(\mathsf{Ha}(CS_1), \ldots, \mathsf{Ha}(CS_t)) \\ (CS_1, \ldots, CS_t, e^1, d^1) \leftarrow \mathsf{Sh}(1^\kappa, f) \\ b \xleftarrow{R} \{0,1\}\end{array}\right)$$

is negligible in κ.

Intuitively, Definition 5 says that it is not possible for the evaluating server to determine whether circuit garbling was outsourced.

5.2 Construction

W.l.o.g. we assume that the share generation and share combination are executed by the client and the share encryption by the encryption server.

Share Generation Sh.

Our share generation algorithm proceeds exactly as the circuit garbling algorithm in Section 3.2. The client creates circuit C and random keys $v_k^{\{0,1\}}$ for all wires $w_k \in W$. For each gate g_k each entry $x \in \{0,1\}$, $y \in \{0,1\}$, $z = \tau_g(x,y)$ is represented as

$$v_i^x \ , \ v_j^y \ , \ v_k^z$$

The client then creates for each $s : 1 \le s \le t-1$ the share CS_s as follows. For each entry of each gate g_k he uniformly selects $\vec{r}_{s,1}$ and $\vec{r}_{s,2}$ from $\mathbb{Z}_q^{\frac{1}{2}\mu}$ and $\vec{r}_{s,3}$ from $\mathbb{Z}_q^\nu$. These entries are then represented as

$$\vec{r}_{s,1} \ , \ \vec{r}_{s,2} \ , \ \vec{r}_{s,3}$$

The union of all entries of all gates $g \in G$ is circuit share CS_s.

The client creates the final t-th share as follows. He maps the keys v_i^x and v_j^y to vectors $\vec{v'}_i^x$ and $\vec{v'}_j^y$, respectively, in $\mathbb{Z}_\delta^{\frac{1}{2}\mu}$ by their bit representation: $\frac{1}{2}\mu\delta = \kappa$. He expands the vectors $\vec{v'}_i^x$ and $\vec{v'}_j^y$ to vectors $\vec{v}_i^x$ and $\vec{v}_j^y$ in $\mathbb{Z}_q^{\frac{1}{2}\mu}$ by keeping each coordinate the same: $\delta < q$. He maps the key v_k^z directly to a vectors $\vec{v}_k^z$ in $\mathbb{Z}_q^\nu$.

The client then computes the entries of the gates of the t-th share as

$$\vec{v}_i^x - \sum_{s=1}^{t-1} \vec{r}_{s,1} \ , \ \vec{v}_j^y - \sum_{s=1}^{t-1} \vec{r}_{s,2} \ , \ \vec{v}_k^z - \sum_{s=1}^{t-1} \vec{r}_{s,3} \bmod q$$

We refer to this tuple with the same notation $\vec{r}_{t,1}, \vec{r}_{t,2}, \vec{r}_{t,3}$ as above, since the vectors are independently, identically distributed in the same domain. The union of all entries of all gates $g \in G$ is circuit share CS_t.

Share Encryption Ha.

The encrypting server with circuit share CS_s $(1 \le s \le t)$ then performs the following encryption operation for each entry of each gate:

$$\vec{u}_s = \vec{r}_{s,3} + \mathsf{H}_{\vec{A}}(\vec{r}_{s,1} || \vec{r}_{s,2}) \bmod q$$

The encrypting server performs all cryptographic operations, namely hashing. The clients only need to create secret shares. The encrypting server returns the union of all $\vec{u}_s$ for circuit share CS_s as garbled share GS_s.

Share Combination Cb.

The client – after receiving all garbled shares GS_s from the encrypting servers – combines them into the garbled circuit GC. For each entry of each gate the client computes

$$
\begin{aligned}
& \sum_{s=1}^{t} \vec{u}_s \bmod q \\
= \; & \sum_{s=1}^{t} \vec{r}_{s,3} + \sum_{s=1}^{t} \mathsf{H}_{\vec{A}}(\vec{r}_{s,1} || \vec{r}_{s,2}) \\
= \; & \vec{v}_k^z + \mathsf{H}_{\vec{A}}(\sum_{s=1}^{t} (\vec{r}_{s,1} || \vec{r}_{s,2}) \bmod q) \\
= \; & \vec{v}_k^z + \mathsf{H}_{\vec{A}}(\vec{v}_i^x || \vec{v}_j^y)
\end{aligned}
$$

5.3 Security Proofs

Every garbling scheme reveals some side information, as does our outsourcing scheme. Following the definitions of Bellare et al. we captures this side information using the functions $\Phi_{\mathcal{O}}$ and $\Phi_{\mathcal{G}}$, respectively. We emphasize that

$$\Phi_{\mathcal{O}}(f) \subset \Phi_{\mathcal{G}}(f)$$

i.e., the garbling scheme reveals strictly more information than the outsourcing scheme. This implies that the evaluating server learns more information about the function computed than the encrypting server. This asymmetry coincides well with our economic motivation: The evaluating server offers a service to the client whereas the encrypting servers offer to speed up the input to this server. Furthermore, the evaluating server is still oblivious to input and output of the client as in Definition 2.

Input and Output.

PROPOSITION 6 (INPUT AND OUTPUT OBLIVIOUSNESS). *The garbling scheme $\mathcal{G}$ is* input and output oblivious *as in Definition 2.*

The proof is actually quite complex and we refer the reader to the work of Lindell and Pinkas [24]. A condition of the proof using secure double encryption is that the cryptographic hash function is at least a secret-coin weak pseudo-random function [30]. This is still an assumption for Ajtai's hash function, but strong pseudo-randomness has been proven for the very closely related and also homomorphic hash functions of [2, 5]. We have not given any definition of input and output obliviousness of an outsourcing scheme. This is not a concern, since - differently from a garbling scheme – no information about the input or output, needs to be communicated. Therefore it is easy to see that it is input and output oblivious.

Function.

In order to judge the security of our scheme we specify the side information $\Phi_{\mathcal{G}}$ and $\Phi_{\mathcal{O}}$ in detail.
Side Information $\Phi_{\mathcal{G}}(f)$:

- Size n of the input
- Size m of the output
- Number $|f|$ of gates
- Wiring information i, j, k for each gate $g_k \in G$
- In case of free-XOR:
  - Type information k, τ_k for each gate $\{g_k | g_k \in G \wedge \tau_k = XOR\}$

Side Information $\Phi_{\mathcal{O}}(f)$:

- In case of free-XOR:
  - Number $|f'|$ of gates $\{g_k | g_k \in G \wedge \tau_k \neq XOR\}$
- In other cases:
  - Number $|f|$ of gates

THEOREM 7 (FUNCTION OBLIVIOUSNESS). *Let $\Phi_{\mathcal{O}}(f)$ as described above be our side information function. Then, our outsourcing scheme $\mathcal{O}$ is* function oblivious *as in Definition 4.*

PROOF. We prove by giving a simulator $\mathcal{S}(1^\kappa, \Phi_{\mathcal{O}}(f))$ for a circuit share GS_s (without input f_b) that is indistinguishable from a circuit share computed by $\mathsf{Sh}(1^\kappa, f_b)$. The simulator $\mathcal{S}(1^\kappa, \Phi_{\mathcal{O}}(f))$ uniformly selects for each entry of each gate (i.e. $4|f|$) random vectors $\vec{r}_1$ and $\vec{r}_2$ from $\mathbb{Z}_q^{\frac{1}{2}\mu}$ and $\vec{r}_3$ from $\mathbb{Z}_q^{\nu}$. For shares GS_s $(1 \leq s \leq t-1)$ the algorithm is the same as in algorithm $\mathsf{Sh}(1^\kappa, f_b)$. Share GC_t is independently, identically distributed, since in $\mathsf{Sh}(1^\kappa, f_b)$ it is the difference of uniform distributions. □

Outsourcing.

THEOREM 8 (OUTSOURCING OBLIVIOUSNESS). *Our outsourcing scheme $\mathcal{O}$ is* outsourcing oblivious *as in Definition 5.*

PROOF. The client computes in Cb the same encryption $\vec{v}_k^z + \mathsf{H}_{\vec{A}}(\vec{v}_i^x || \vec{v}_j^y)$ as in Gb. The vectors $\vec{v}_i^x$, $\vec{v}_j^y$ and $\vec{v}_k^z$ are chosen in Sh in the same way as in Gb. Therefore the result is indistinguishable. □

6. IMPLEMENTATION

We implemented the algorithms of our outsourcing scheme to measure its performance. The implementation was done in Java 1.6 and run on a 64-bit HotSpot Server virtual machine. All reported measurements are the median of the 50 last runs of 100 consecutive runs in order to counter effects of just-in-time compilation. All measurements were performed on an Intel Core i5 CPU with 2.4 GHz and 4GB RAM. The operating system was Windows 7 Enterprise.

6.1 Hash Function Performance

The bottleneck of each garbling scheme is its cryptographic functions, in our case the hash function. It is crucial for our proposed change of hash function that it does not increase the cost of evaluating a garbled circuit. Other homomorphic hash functions, such as the Chaum-van Heijst-Pfitzmann hash function [10], are very costly to compute. A clear advantage of lattice-based cryptography is its speed. We use the following parameters for Ajtai's hash function as recommended in [27].

$$
\begin{aligned}
\mu &= 4096 \\
\nu &= 256 \\
\delta &= 2 \\
q &= 256
\end{aligned}
$$

Hence, we achieve a compression factor of exactly 2. We compare Ajtai's hash function to SHA-3, the recent standard of NIST. We translated the code submitted to the competition to Java. The results for one invocation of the hash function are shown in Table 1.

Ajtai	0.85 ms
SHA-3	3.5 ms

Table 1: Performance of hash functions

Both garbled circuit evaluation Ev and share encryption Ha perform linearly many hash functions in the number of (non-XOR) gates. For large functions f their run-time is therefore dominated by hash computations. We can see that the proposed parameters for Ajtai's hash function are almost four times faster than for SHA-3. The goal of our comparison is not to optimize implementation or parameters, but show that the performance of the hash functions is "comparable". A particular choice of parameters or code optimizations need to be done with respect to the application. Still, we show that our proposed replacement of hash function does not imply unreasonably large costs on the evaluating server.

6.2 Share Generation and Combination

The client needs to perform share generation Sh and combination Cb. Since we claim that our outsourcing scheme is suitable for computationally weak devices, this operation needs to be fast. Most notably, it needs to be *significantly* faster than generating the circuit on the device. Share generation involves choosing many random numbers. Fortunately, these are independent of the function to be computed and can be easily pre-generated. When perform an outsourced computation we therefore only need to compute the secret shares.

We set $t = 2$ outsourcing servers in our evaluation and use otherwise the same parameters as above. We perform both share generation and combination for 100 gates. The results are shown in Table 2.

Generation & Combination	6.2 ms

Table 2: Performance of 100 gates share generation and combination

The lower bound for garbled circuit generation is performing four hash functions per gate – one for each entry. Therefore generating a circuit with 100 gates takes at least 340 ms. Hence, our outsourcing scheme reduces the workload on the client by a factor of 55. Again, our implementation does not perform optimization of parameters or code, but shows that there can be a significant advantage in using our outsourcing scheme.

7. MULTI-USER SCENARIO

So far we have focused on a single client outsourcing its computation, but it is easy to see that circuit share generation and combination scales linearly in the number of gates. The client therefore does not save much computational effort compared to locally executing the function f. It is also easy to see that *outsourcing obliviousness* implies this lower bound in any garbling scheme. Since the client needs to communicate the entire garbled circuit GC to the evaluating server, it always has communication complexity and hence computation complexity in the size of the circuit and consequently the number of gates. If the client would not communicate the entire circuit, the evaluating server could determine that outsourcing has taken place.[4]

The situation changes when two computationally weak devices collaborate using their inputs or one device stores data on the evaluating server. We elaborate on the first case of two securely collaborating devices as in [15, 26, 29]. Assume Alice and Bob each have a device and input a and b, respectively. They want to compute the function $(y, z) = f(a, b)$ on their joint inputs. They could use a garbling scheme and oblivious transfer in order to execute this secure computation, as done in [15].

Still, for larger computations they may want to outsource the computation to a larger computer, say the cloud Charlie. This is envisioned in [7, 8], but only achieved for one computationally weak device – the one that does not generate the garbled circuit. They do this using any garbling scheme as defined by Bellare et al. by sharing the encoding and decoding keys e, d. The cloud would then perform the evaluation function Ev. In [9] the authors reverse the roles of circuit generator and evaluator, but still they do not achieve outsourcing obliviousness and a computationally weak device has to evaluate the garbled circuit (instead of the cloud).

Yet, the most cryptographic operations are performed in garbled circuit generation Gb. This must still be performed on the devices – usually by one device. Using our outsourcing scheme this operation can be also outsourced. We describe this scenario in detail for $t = 2$ outsourcing servers – Sally and Tom.

1. Alice executes $(CS_1, CS_2, e, d) \leftarrow \mathsf{Sh}(1^\kappa, f)$. She sends e and d to Bob, CS_1 to Sally and CS_2 to Tom.
2. Sally and Tom execute $GS_1 \leftarrow \mathsf{Ha}(CS_1)$ or $GS_2 \leftarrow \mathsf{Ha}(CS_2)$, respectively. They return GS_1 or GS_2 to Alice.
3. Alice executes $GC \leftarrow \mathsf{Cb}(GS_1, GS_2)$. Alice sends GC and $A \leftarrow \mathsf{En}(e, a)$ to Charlie.
4. Bob sends $B \leftarrow \mathsf{En}(e, b)$ to Charlie.
5. Charlie executes $(Y, Z) \leftarrow \mathsf{Ev}(GC, A, B)$. He sends Y to Alice and Z to Bob.
6. Alice and Bob execute $y \leftarrow \mathsf{De}(d, Y)$ or $z \leftarrow \mathsf{De}(d, Z)$, respectively.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments on improving the paper.

8. REFERENCES

[1] M. Ajtai. Generating Hard Instances of Lattice Problems (Extended Abstract). In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC)*, 1996.

[4] Anonymous communication may change this, but has other implications we deem detrimental to our scenario.

[2] A. Banerjee, and C. Peikert. New and Improved Key-Homomorphic Pseudorandom Functions. In *Advances in Cryptology (CRYPTO)*, 2014.

[3] M. Bellare, V. Hoang, S. Keelveedhi, and P. Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (SP)*, 2013

[4] M. Bellare, V. Hoang, and P. Rogaway. Foundations of Garbled Circuits. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[5] D. Boneh, K. Lewi, H. Montgomery, and A. Raghunathan. Key Homomorphic PRFs and Their Applications. In *Advances in Cryptology (CRYPTO)*, 2013.

[6] S. Bugiel, S. Nürnberger, A. Sadeghi, and T. Schneider. Twin Clouds: An Architecture for Secure Cloud Computing. *Workshop on Cryptography and Security in Clouds (CSC)*, 2011.

[7] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For Your Phone Only: Custom Protocols for Efficient Secure Function Evaluation on Mobile Devices. *Journal of Security and Communication Networks (JSCN)*, 2013.

[8] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *Proceedings of the 22nd USENIX Security Symposium (SECURITY)*, 2013.

[9] H. Carter, C. Lever, and P. Traynor. Whitewash: Outsourcing Garbled Circuit Generation for Mobile Devices. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC)*, 2014.

[10] D. Chaum, E. van Heijst, and B. Pfitzmann. Cryptographically Strong Undeniable Signatures, Unconditionally Secure for the Signer. In *Advances in Cryptology (CRYPTO)*, 1991.

[11] U. Feige, J. Kilian, and M. Naor. A Minimal Model for Secure Computation (Extended Abstract). In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing (STOC)*, 1994.

[12] O. Goldreich. The Foundations of Cryptography - Volume 2. *Cambridge University Press*, 2004.

[13] O. Goldreich, S. Goldwasser, and S. Halevi. Collision-Free Hashing from Lattice Problems. *Technical Report TR96-056, Electronic Colloquium on Computational Complexity (ECCC)*, 1996.

[14] W. Henecka, and T. Schneider. Faster Secure Two-Party Computation with Less Memory. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2013.

[15] Y. Huang, P. Chapman, and D. Evans. Privacy-Preserving Applications on Smartphones. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Security (HotSec)*, 2011.

[16] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proceedings of the 20th USENIX Security Symposium (SECURITY)*, 2011.

[17] M. Jawurek, and F. Kerschbaum. Fault-Tolerant Privacy-Preserving Statistics. In *Proceedings of the 12th Symposium on Privacy Enhancing Technologies (PETS)*, 2012.

[18] M. Jawurek, F. Kerschbaum, and C. Orlandi. Zero-Knowledge Using Garbled Circuits: How To Prove Non-Algebraic Statements Efficiently. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[19] S. Kamara, P. Mohassel, and B. Riva. Salus: A System for Server-Aided Secure Function Evaluation. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[20] F. Kerschbaum. Building a Privacy-Preserving Benchmarking Enterprise System. *Enterprise Information Systems 2 (4)*, 2008.

[21] F. Kerschbaum, and O. Terzidis. Filtering for Private Collaborative Benchmarking. In *Proceedings of the Conference on Emerging Trends in Information and Communication Security (ETRICS)*, 2006.

[22] V. Kolesnikov, and T. Schneider. Improved Garbled Circuits: Free XOR Gates and Applications. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming*, 2008.

[23] B. Kreuter, A. Shelat, and C. Shen. Billion-Gate Secure Computation with Malicious Adversaries. In *Proceedings of the 21st USENIX Security Symposium (SECURITY)*, 2012.

[24] Y. Lindell, and B. Pinkas. A Proof of Yao's Protocol for Secure Two-Party Computation. *Journal of Cryptology 22(2)*, 2009.

[25] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - A Secure Two-Party Computation System. In *Proceedings of the 13th USENIX Security Symposium (SECURITY)*, 2004.

[26] D. Mayer, D. Teubert, S. Wetzel, U. Meyer, and G. Neugebauer. appoint - A Distributed Privacy-Preserving iPhone Application. *3rd ACM Conference on Wireless Security (WISEC)*, Poster Session, 2010.

[27] D. Micciancio, and O. Regev. Lattice-based Cryptography. In D. Bernstein, and J. Buchmann (eds.), *Post-quantum Cryptography*, Springer, 2008.

[28] M. Naor, B. Pinkas, and R. Sumner. Privacy Preserving Auctions and Mechanism Design. In *Proceedings of the 1st ACM Conference on Electronic Commerce (EC)*, 1999.

[29] G. Neugebauer, L. Brutschy, U. Meyer, and S. Wetzel. Design and Implementation of Privacy-Preserving Reconciliation Protocols. In *Proceedings of the 6th ACM International Workshop on Privacy and Anonymity in the Information Society (PAIS)*, 2013.

[30] K. Pietrzak, and J. Sjödin. Weak Pseudorandom Functions in Minicrypt. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP)*, 2008.

[31] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure Two-Party Computation is Practical. In *Advances in Cryptology (ASIACRYPT)*, 2009.

[32] A. Yao. How to Generate and Exchange Secrets (Extended Abstract). In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science (FOCS)*, 1986.