# CheapSMC: A Framework to Minimize Secure Multiparty Computation Cost in the Cloud

Erman Pattuk[1], Murat Kantarcioglu[1], Huseyin Ulusoy[1], and Bradley Malin[2]

[1] The Univ. of Texas at Dallas
erman.pattuk,muratk,huseyin.ulusoy@utdallas.edu
[2] Vanderbilt University
b.malin@vanderbilt.edu

**Abstract.** Secure multi-party computation (SMC) techniques are increasingly more efficient and practical, due in part, to various improvements. For instance, recent research has shown that different protocols that are implemented using different sharing mechanisms (e.g., boolean and arithmetic sharings) can have varying computational and communication costs. Although there are some approaches to automatically mix protocols of different sharing schemes to enhance execution efficiency, none provide a generic optimization framework to discover the least expensive mixed-protocol SMC execution for cloud deployment.
In this work, we introduce a generic SMC optimization framework CheapSMC that can invoke any mixed-protocol SMC circuit evaluation tool as a black box to uncover the cheapest SMC cloud deployment option. To do so, CheapSMC computes one-time benchmarks for the target cloud service and gathers performance statistics for basic circuit components. Relying on these statistics, an optimization layer of CheapSMC invokes several heuristics to find the cheapest mix-protocol circuit evaluation. Subsequently, the optimized circuit is passed to a mixed-protocol SMC tool for actual executable generation. Our empirical results, gathered by running cases studies on large range of complexity in data volume and functions for computation, show that significant cost savings can be achieved via our optimization framework in comparison to the state-of-the-art.

## 1 Introduction

Over the last couple of years, various two-party secure multi-party computation (SMC) protocols have been proposed to address different secure computation needs, ranging from privacy-preserving face recognition (e.g., [1]) to secure biometric identification (e.g., [3]). In addition, numerous generic two-party circuit evaluation platforms (e.g. [5, 12]) have been developed to improve the efficiency of existing secure protocols. Most of these platforms (e.g., [4]) also provide high-level programming language support that can automatically generate circuits from programs written in C-like languages. These recent advances have enabled two-party SMC protocols to be more practical and push towards actual deployment of such technologies.

At the same time, there remain several critical challenges to making these platforms practical. One challenge in particular that has received little attention is performance optimization. Recent research [6] has shown that different two-party SMC protocols can have different computational and communication cost profiles. For example, arithmetic sharing-based circuit evaluation protocols may be better for certain tasks in comparison to Yao's garbled circuit evaluation protocols. On the other hand, Boolean secret

sharing-based circuit evaluation techniques can achieve the best performance in certain situations. Based on such observations, it has been shown [6] that the combination of these techniques can perform much better than each in isolation. This begs the question: *How can we find the best combination of two-party SMC techniques for a given task?*

Most of the existing work to date fails to consider the problem of finding the optimal combination of different sharing- based protocols for a given task. Rather, they require the end user to manually specify the specific sub-protocols that must be invoked. There has been some investigation into optimization and automation of the selection process (e.g., [7]), but it is limited in scope with respect to the cost dimensions that the user can choose to optimize. For example, if one party leverages a cloud infrastructure for running the protocol, the network communication may significantly impact the overall cost (in terms of money) paid by the parties. As such, it is clear that we need an optimization framework that can automatically consider communication, computation and monetary costs when searching for the optimal two-party SMC protocol composition.

The goal of this paper is to introduce an optimization framework where the given two-party SMC task can be automatically optimized under a set of predefined cost constraints. In doing so, the optimal (or near optimal) combination of different sharing-based subprotocols (e.g., arithmetic, boolean, and Yao's secret sharing protocols) can be selected automatically. This goal is similar to other automatic task optimization frameworks associated with other systems. In our optimization framework, we especially focus on the cloud setting because it is being widely adopted by organizations in a wide range of application domains [8] due to its flexibility and low initial management cost. In the cloud setting, in addition to minimizing the overall run time of the system, we may need to balance the network traffic, and computation time to achieve overall lowest monetary cost. This makes the problem more challenging because it suggests we need to consider both communication and computation costs in the optimal mixed-protocol circuit generation.

**Overview of the CheapSMC.** The objective of the system [3] is to make it easier for users to implement and execute SMC protocols, while minimizing the monetary cost of the SMC execution in the cloud. To ease the implementation phase and make CheapSMC extensible to available SMC tools, we partition it into three primary components. First, the *Programming API* acts as the frontend for the users, which enables implementation of SMC protocol using a C++ library. This layer is ultimately responsible for representing the user protocol as a circuit of atomic operations. It should be noted that CheapSMC can be further extended by designing a custom language (e.g., SFDL of *Fairplay* [4]), which uses our *Programming API* in the background. In this work, we do not focus on such integration and instead focus on the optimization aspects.

Second, the *Optimization Module* is responsible for assigning secret sharing schemes (e.g., Arithmetic, Boolean, Yao sharing as in the *ABY* framework [6]; Additively homomorphic, Yao sharing as in the *Tasty* framework [12]) to the nodes in the circuit, such that the total cost of executing the protocol in the cloud is minimized. Finding an optimal solution to this problem is NP-Hard, so this module provides heuristics to find near-optimal solutions.

Finally, the *SMC Layer* implements the *optimized* circuit given an existing SMC tool (e.g., *ABY* or *Sharemind* [5]). We recognize the development of efficient SMC tools is a vibrant research area, such that the design of CheapSMC does not focus on a single SMC tool that could limit its usefulness. Rather, we leverage a given SMC

[3] Please see `https://arxiv.org/abs/1605.00300` for the full version of our paper.

tool as a black box, provided that it is a mixed-protocol SMC tool (i.e., it allows for implementation using different sharing schemes).

CheapSMC relies on several atomic operations (e.g., addition, multiplication, binary xor, as well as binary and) that cover various application scenarios. In Section 4, we show the results of applying our system to several case studies, while further applications can be realized using our C++ library. Moreover, the process of optimizing the circuit is decoupled from the circuit generation and other layers, so that proposing a new heuristic and implementing it can be achieved with minimal effort.

## 2 The Optimal Partitioning Problem

### 2.1 Problem Definition

Let $\mathcal{S} = \{s_1, \ldots, s_n\}$ be the set of provided secret sharing mechanisms. Then, given a variable $x$ in some domain $\mathcal{I}$, let $[x]_{s_i}$ represent the secret sharing of $x$ using $s_i$.

Next, we define the set of operations $\mathcal{O} = \{o_1, \ldots, o_k\}$, such that each operation $o_i \in \mathcal{O}$ takes a set of parameters that are secretly shared in $s_j$ and outputs a single variable secretly shared in $s_j$. Note that the number of inputs that an operation takes is fixed, regardless of the secret sharing scheme. An operation $o_i$ is **supported** in $s_j$, if there exists an execution protocol that takes the input parameters to $o_i$ and outputs a result secretly shared in $s_j$. Let $\delta(o_i) \subseteq \mathcal{S}$ represent the secret sharing mechanisms that support operation $o_i \in \mathcal{O}$.

Since an operation can be executed in the cloud environment, one should approximate the monetary cost of performing the protocol execution in a certain setup. In order to achieve this goal, we focus on the processing and network transfer costs of executing a single operation in the pay-as-you-go cloud model. In this computing model, a customer of a cloud provider service is charged a constant amount per unit time for using a particular type of virtual machine (VM), while the prices vary depending on the processing capabilities of the VM. On the other hand, the monetary cost of transferring a single byte in and out of the VM is fixed based on the VM specifications. The unit cost of network transfer vary as the network capacity of the VM changes.

Under such circumstances, we define the processing and network transfer costs of executing an operation $o_i \in \mathcal{O}$ in the secret sharing scheme $s_j \in \delta(o_i)$ as $\mathtt{P}(o_i, s_j)$ and $\mathtt{N}(o_i, s_j)$, respectively. Furthermore, we define the processing and network transfer cost of converting a variable that is secretly shared in $s_i \in \mathcal{S}$ to $s_j \in \mathcal{S}$ as $\mathtt{CP}(s_i, s_j)$ and $\mathtt{CN}(s_i, s_j)$, respectively. Note that defined costs may vary based upon VM specifications.

Given the set of operations $\mathcal{O}$, the parties in the computation (i.e., the server and the client) implement a circuit $\mathcal{C}$ that is represented as a directed acyclic graph (DAG) and consists of $m$ nodes $c_1, \ldots, c_m$. Each node represents a single operation and takes input from other nodes, whereas the number of inputs is decided by the operation. To be more concrete, let $\alpha(c_i) \in \mathcal{O}$ represent the operation that is assigned to the node $c_i \in \mathcal{C}$, while $\beta(c_i) \subseteq \mathcal{C}$ is the set of nodes that supply input to $c_i$[4]. Furthermore, let $\gamma(c_i) \in \mathcal{S}$ be the secret sharing scheme assigned to $c_i$. Then the monetary cost of executing a node

[4] Note that this circuit representation of a computation can be provided by the programming language.

$c_i \in \mathcal{C}$ is simply:

$$\texttt{cost}(c_i) = \texttt{P}(\alpha(c_i), \gamma(c_i)) + \texttt{N}(\alpha(c_i), \gamma(c_i)) + \sum_{c_j \in \beta(c_i)} \texttt{CP}(\gamma(c_j), \gamma(c_i)) + \texttt{CN}(\gamma(c_j), \gamma(c_i)) \tag{1}$$

Using the above definitions, the optimal partitioning problem investigated in this paper is formally defined as follows:

**Definition 1.** *Given the processing and network transfer cost for a set of operations $\mathcal{O}$ using a set of secret sharing schemes $\mathcal{S}$, the cost of conversion between different secret sharing schemes, and a circuit $\mathcal{C} = \{c_1, \ldots, c_m\}$ of $m$ nodes, where each node $c_i$ is assigned an operation $\alpha(c_i) \in \mathcal{O}$, assign a secret sharing scheme to each node $c_i$, such that the total monetary cost of executing the circuit with the assigned secret sharing schemes is minimal.*

$$\begin{aligned} \textit{Given} : & \mathcal{O}, \mathcal{S}, \mathcal{C}, \texttt{P}(o,s) \textit{ and } \texttt{N}(o,s)\ \forall o \in \mathcal{O}, \forall s \in \delta(o), \\ & \texttt{CP}(s_i, s_j) \textit{ and } \texttt{CN}(s_i, s_j)\ \forall s_i, s_j \in \mathcal{S}, \alpha(c_i)\ \forall c_i \in \mathcal{C} \\ \textit{Minimize} : & \sum_{c_i \in \mathcal{C}} cost(c_i) \textit{ Subject to} : \gamma(c_i) \in \delta(\alpha(c_i))\ \forall c_i \in \mathcal{C} \end{aligned} \tag{2}$$

The partitioning problem is NP-Hard, and the reduction can be realized via the Integer Programming problem.

## 3 The Details of CheapSMC

### 3.1 Architecture

CheapSMC is composed three primary parts: (i) The programming interface (API), (ii) the optimization module, and (iii) the SMC layer. Each part is responsible for a different task: The programming interface morphs a user's program into its circuit representation. The optimization module heuristically assigns the secret sharing schemes to each node in the circuit using. The SMC layer generates the executables using state-of-the-art cryptographic primitives and techniques. The user of CheapSMC is expected to provide two inputs: (A) the specifications of the protocol that are going to be executed securely and (B) the unit costs for the operations and secret sharing schemes supported by the SMC framework. As discussed later, we provide a benchmark suite to automatically learn these unit costs for any target cloud service to assist the user.

**User Inputs.** It is assumed that the user knows the secure protocol for the application for CheapSMC will be invoked. One input to our system is the protocol specification, which can either be implemented using the associated C++ library or via a custom programming language, whose compiler turns the user input to an output compatible with our programming API. Next, the user must input the unit monetary cost of the operations for the hardware specifications that secure executables will work over. We have implemented a set of benchmark applications that can be executed by a user to discover the unit costs of each operation. It should be recognize that this is a one-time operation per tested cloud environment.

**Programming API.** We implemented an extensible library in C++ that allows a user to implement a secure protocol (e.g., set intersection or biometric matching). We provided several operations in the API that cover a variety of applications. Currently, the set of operations $\mathcal{O}$ include addition (`Add`), subtraction (`Sub`), multiplication (`Mul`), greater (`Ge`), equality (`Eq`), multiplexer (`Mux`), binary xor (`Xor`), and binary and (`And`). There are also two additional operations in the form of input (`In`) and output (`Out`), which allow the programmer to specify secret inputs to the circuit and to learn the outputs of the protocol execution.

As a proof of concept, we provided the interface as a C++ library that can be used to generate cost-optimal SMC executables. The interface can be bound with the compiler of a custom programming language, though which the system user can type the protocol specifications. Next, the compiler can generate the C++ program that uses the CheapSMC programming interface. In any scenario, the programming interface transforms the protocol into the circuit representation (See Section 2).

**Optimizer.** Given the circuit representation of the user protocol and the secret sharing schemes that are provided by the SMC layer, the optimizer module applies one of the heuristics (cf. Section 3.2) to assign secret sharing schemes to each node in the circuit. As discussed earlier, this module is responsible for finding the assignment that minimizes the monetary cost of executing the protocol securely.

Due to the fact that the optimal partitioning problem problem is NP-Hard, finding the optimal assignment may be impractical (even for a circuit of moderate size). As such, we introduced several heuristics (See Section 3.2) that are oriented to find a reasonable solution. In the background, CheapSMC applies each heuristic and chooses the one that gives the best result.

**SMC Layer.** Once the secret sharing schemes are assigned to each node in the circuit, CheapSMC passes the circuit to the SMC layer to generate the SMC executables. It should be recognized that there are several related investigations that provide mixed-protocol SMC tools, including the *ABY* framework by Demmler et al [6], the *TASTY* framework by Henecka et al. [12], and the *Sharemind* framework by Bogdanov et al. [5]. The SMC layer in CheapSMC is responsible for automatically implementing the optimized circuit from such existing tools. Note that since the selected SMC tool provides the low-level implementation of the cryptographic primitives (e.g., oblivious transfer [10, 11], multiplication triplets [2], and sharing and reconstructing secret inputs), we focus on optimizing the user protocol using sharing assignments.

### 3.2 Optimization Heuristics

In addition to two existing heuristics, we developed two new heuristics to solve the optimal partitioning problem. Here, we provide a high-level description of thee heuristics.

**Bottom-up Heuristic.** The key idea in this heuristic is to assign an *optimal* secret sharing scheme to the nodes in their topological order in the circuit. When a node $c_i \in \mathcal{C}$ is ready to be processed, the heuristic first assigns sharing schemes to the nodes that provide input to $c_i$. Based on the values assigned to the *children*, this heuristic selects the scheme that minimizes the expected monetary cost for $c_i$.

**Top-Down Heuristic.** This technique processes the nodes of the circuit in a manner quite the opposite of the Bottom-Up heuristic. Specifically, it assigns secret sharing schemes to the higher-level nodes first and iterates down to the lower levels. The idea in this heuristic is to assign the scheme that minimizes the cost of the current node,

given that the schemes for the nodes for which its input to are already known. Now, assume that the secret sharing scheme for the node $c_k$ is set to $s_k$ previously. When assigning the secret sharing scheme for the node $c_i$, this heuristic takes into account that the result of the node $c_i$ should be converted to $s_k$. The *optimal* decision is made with this consideration in mind.

**Fixed Secret Sharing.** In this optimization heuristic, each node in the circuit is assigned the same secret sharing scheme. However, in some SMC tools, certain secret sharing schemes may not necessarily support each single operation (e.g., Arithmetic sharing in *ABY* framework does not support `And`, `Xor`, and `Mux`). In such a case, this heuristic selects one scheme that supports each CheapSMC operation. One common secret sharing scheme that is included in almost all SMC tools and supports each operation is garbled circuit sharing (based on Yao's garbled circuit protocol [13]). Using this heuristic, we can measure the monetary cost of executing the user protocol by a single secret sharing scheme (e.g., pure SMC by garbled circuit sharing).

**Hill-Climbing.** This heuristic is based on the technique of Kerschbaum et al. [7]. The basic idea is to start by assigning a common secret sharing scheme (e.g., garbled circuit sharing) to each node in the circuit. Next, we check if the total cost can be reduced by changing the current secret sharing scheme of a node. This loop continues until the total cost cannot be improved by any further assignment.

# 4 Case Studies

## 4.1 Experiment Setup

We conducted empirical performance evaluations in two scenarios: (1) *Intra-Region*, where the parties are in the same Amazon EC2 region and (2) *Inter-Region*, where the parties are not in same region, for four different Amazon EC2 VM models. We tested each scenario and VM model with four techniques: the three optimization heuristics - (i.e., *Top-down*, *Bottom-up*, and *Hill Climbing*) - and *Pure-Yao*, which assigns Yao-style (i.e., garbled circuit) sharing to each node in the circuit.

In addition to the monetary cost of running CheapSMC, we measure the average running time of the four techniques. As mentioned in Section 3.2, our optimization problem can be enhanced by introducing performance constraints (e.g., the expected running time should be less than some threshold $t$). Given such a performance constraint, the heuristic solver may prune any solution that fails to satisfy the estimated performance constraint.

## 4.2 Biometric Matching

*Biometric matching* applications cover a two-party scenario, where (i) the server has a set of private entries and (ii) a client, holding its private entry, wants to learn the closest entry in the server's dataset based on some similarity measure. There are various problems related to this case study (e.g., biometric identification [3], fingercode authentication [9], and face recognition [1, 14]). One of the commonly used distance metric is squared Euclidean distance. In this protocol, the server and the client proceed over the server's dataset one by one, which results in the client learning the entry with the minimal distance to its private input. We implemented this case study using our Programming API for a dataset of 30 rows with 5 attributes of 32-bit numbers.

We performed tests on four different Amazon EC2 VM configurations. Detailed information on the specifications of each VM configuration can be found in the full version of this paper.

| | | Execution time ($ms$) | | | |
|---|---|---|---|---|---|
| | | Pure-GC | Hill | TD | BU |
| INTRA | m3.med | 1229.481 | 398.02 | **348.43** | 416.944 |
| | m3.large | 715.5147 | 355.2913 | **321.388** | 372.529 |
| | c4.large | 577.333 | 293.544 | **256.363** | 284.378 |
| | c4.xlarge | 561.602 | 352.197 | **292.522** | 326.393 |
| INTER | m3.med | **5518.04** | 6738.14 | 6237.37 | 6749.467 |
| | m3.large | **4801.94** | 6096.73 | 6103.647 | 6083.193 |
| | c4.large | **8731.577** | 35580.13 | 33267.77 | 35569.97 |
| | c4.xlarge | 9118.693 | 6916.357 | **6381.093** | 6826.727 |

**Table 1.** The average execution time for the *Biometric Matching* case study in the Amazon EC2 Cloud. The results are for two different scenarios, four different VM models, and four different techniques.

Table 1 shows the average running time for the *Biometric Matching* case study, two different scenarios, four VM models, and four secret-sharing assignment heuristics. As expected, the performance is much better in the Intra-Region scenario. In all cases, applying any of the heuristics yields lower running times compared to the Pure-GC assignment (i.e., where each node is assigned the Yao sharing). Moreover, our Top-Down heuristic yields the best running time for all VM models. Specifically, it is $15\%$ better than the Hill Climbing heuristic of Kerschbaum et al [7] in terms performance. For the Inter-Region scenario, we find that Pure-GC performs much better than the other techniques in terms of performance, except for the last model c4.xlarge. Since the physical distance between the two parties is large (i.e., between Tokyo and North Virginia), network latency plays a vital role in the overall performance. And it is shown that Yao sharing is much better than any other solution in high-latency networks. Since our primary optimization objective is to minimize cost (and not to minimize performance) the Inter-Region results are not surprising.

| | | Computation Cost (¢$10^{-3}$) | | | | Network Cost (¢$10^{-3}$) | | | | Total Cost (¢$10^{-3}$) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Pure-GC | Hill | TD | BU | Pure-GC | Hill | TD | BU | Pure-GC | Hill | TD | BU |
| INTRA | m3.med | 4.78 | 1.55 | **1.36** | 1.62 | 0.00 | 0.00 | 0.00 | 0.00 | 4.78 | 1.55 | **1.36** | 1.62 |
| | m3.large | 5.57 | 2.76 | **2.50** | 2.90 | 0.00 | 0.00 | 0.00 | 0.00 | 5.57 | 2.76 | **2.50** | 2.90 |
| | c4.large | 3.72 | 1.89 | **1.65** | 1.83 | 0.00 | 0.00 | 0.00 | 0.00 | 3.72 | 1.89 | **1.65** | 1.83 |
| | c4.xlarge | 7.24 | 4.54 | **3.77** | 4.21 | 0.00 | 0.00 | 0.00 | 0.00 | 7.24 | 4.54 | **3.77** | 4.21 |
| INTER | m3.med | **27.74** | 33.88 | 31.36 | 33.93 | 990.71 | 189.71 | **129.71** | 189.71 | 1018.46 | 223.58 | **161.07** | 223.64 |
| | m3.large | **45.75** | 58.09 | 58.15 | 57.96 | 990.71 | 189.71 | **129.71** | 189.71 | 1036.46 | 247.80 | **187.86** | 247.67 |
| | c4.large | **63.79** | 259.93 | 243.04 | 259.86 | 990.71 | 218.29 | **173.27** | 218.29 | 1054.50 | 478.23 | **416.31** | 478.15 |
| | c4.xlarge | 133.23 | 101.06 | **93.23** | 99.75 | 990.71 | 189.71 | **129.71** | 189.71 | 1123.95 | 290.76 | **222.94** | 289.45 |

**Table 2.** The average computational, network, and total cost of running the *Biometric Matching* case study in the Amazon EC2 Cloud. The results are for two different scenarios, four different VM models, and four different techniques.

Table 2 shows the monetary cost for the *Biometric Matching* case study with the aforementioned setup. In the Intra-Region scenario, we see that the Top-Down heuristic performs better than any other technique in all VM models. This is due to a better assignment of secret sharing schemes to the nodes in the circuit for this particular case study. Note that the network communication cost within the same region is $0$ in Amazon EC2, which is why the network cost in Table 2 is simply $0$. In the Inter-Region scenario,

the Top-Down heuristic once again delivers the cheapest assignments for all VM models. It performs $30\%$ better than the Hill Climbing heuristic. In terms of computation cost, we see that Pure-GC performs better due to the reasons discussed earlier (i.e., high network latency). However, in terms of total cost, Top-Down heuristic induces up to $80\%$ reduction.

## 5 Conclusion

This paper introduced CheapSMC, an SMC framework that aims to minimize the cost of executing SMC protocols in the cloud. We performed extensive cost profiling for the Amazon EC2 cloud service leveraged the gathered statistics and applied our system two case studies (i.e., biometric matching and matrix multiplication). We evaluated CheapSMC using four VM models and two scenarios (i.e., Inter-Region and Intra-Region). and we showed that the cost of executing SMC using our heuristics is up to $96\%$ and $30\%$ less than pure garbled circuit and Hill-Climbing methods, respectively. The evidence suggests that purchasing faster and more expensive VM models from Amazon EC2 do not necessarily reduce the total monetary cost of executing SMC protocols. In general, compute-optimized VMs can result in more expenses, while those which are memory-optimized can produce cheaper SMC executions.

## 6 Acknowledgement

The research was supported by grants from the NIH (R01LM009989, R01HG006844, & 1U01HG008701) and NSF (CNS-1111529, CNS-1228198, & CICI-1547324).

## References

1. Ahmed-Reza Sadeghi et al. Efficient privacy-preserving face recognition. In *ICISC*, pages 229–244. Springer, 2010.
2. D. Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology*, pages 420–432. Springer, 1992.
3. D Evans et al. Efficient privacy-preserving biometric identification. In *NDSS*, 2011.
4. Dahlia Malkhi et al. Fairplay-secure two-party computation system. In *USENIX Security*, 2004.
5. Dan Bogdanov et al. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206. Springer, 2008.
6. Daniel Demmler et al. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.
7. Florian Kerschbaum et al. Automatic protocol selection in secure two-party computations. In *ACNS*, pages 566–584. Springer, 2014.
8. Forbes. Cloud computing: United states businesses will spend $13 billion on it. http://www.forbes.com/sites/tjmccue/2014/01/29/cloud-computing-united-states-businesses-will-spend-13-billion-on-it, 2014.
9. Mauro Barni et al. Privacy-preserving fingercode authentication. In *ACM workshop on Multimedia and security*, pages 231–240. ACM, 2010.
10. Moni Naor et al. Efficient oblivious transfer protocols. In *SIAM*, pages 448–457, 2001.
11. M. O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.
12. Wilko Henecka et al. Tasty: tool for automating secure two-party computations. In *ACM CCS*, pages 451–462, 2010.
13. A. C. Yao. Protocols for secure computations. In *IEEE ASFCS*, pages 160–164. IEEE, 1982.
14. Zekeriya Erkin et al. Privacy-preserving face recognition. In *PETS*, pages 235–253, 2009.