Choose Wisely: A Comparison of Secure Two-Party Computation Frameworks

Jan Henrik Ziegeldorf, Jan Metzke, Martin Henze, Klaus Wehrle
Communication and Distributed Systems (COMSYS), RWTH Aachen University, Germany
{ziegeldorf, henze, wehrle}@comsys.rwth-aachen.de, jan.metzke@rwth-aachen.de

***Abstract*—Secure Two-Party Computation (STC), despite being a powerful tool for privacy engineers, is rarely used practically due to two reasons: i) STCs incur significant overheads and ii) developing efficient STCs requires expert knowledge. Recent works propose a variety of frameworks that address these problems. However, the varying assumptions, scenarios, and benchmarks in these works render results incomparable. It is thus hard, if not impossible, for an inexperienced developer of STCs to choose the best framework for her task. In this paper, we present a thorough quantitative performance analysis of recent STC frameworks. Our results reveal significant performance differences and we identify potential for optimizations as well as new research directions for STC. Complemented by a qualitative discussion of the frameworks' usability, our results provide privacy engineers with a dependable information basis to take the decision for the right STC framework fitting their application.**

I. Introduction

In recent years, the societal demand for systems which inherently respect privacy and protect data has significantly increased [1], especially due to press coverage about global surveillance programs and data breaches. One important tool which allows privacy engineers to build such inherently privacy-preserving and data-protection compliant systems is Secure Two-Party Computation (STC). STC allows two mutually distrusting parties to compute a shared functionality without having to reveal their private inputs to the other or any third party. It has, e.g., been applied for secure face recognition [2], [3] where a facial picture held by one party is matched against the private database held by the other party without either party learning the other's private input. Other example applications of STC include secure recommendation systems [4] or secure genetic testing [5]. Theoretically, any computable functionality can also be computed securely under STC.

However, STC is still rarely used beyond research. The reasons for this are twofold: First and foremost, STCs incur significant processing and communication overheads which limit what applications are feasible. Second, developing efficient applications under STC is often cumbersome and requires extensive expert knowledge. All of the above mentioned examples are tailored solutions that have been extensively hand-optimized and still provide practical performance only for problem instances which are significantly smaller than what non-secure but otherwise equivalent implementations can handle. Recent research has focused on these two issues by proposing different frameworks [6]–[10], libraries [11] and even language compilers [12]–[15] that all promise both an improved performance and easier development of STCs.

For a privacy engineer, choosing the right one among these frameworks is crucial as the scientific work presenting the frameworks claim significant performance differences. However, the used evaluation settings vary greatly, e.g., in the tested functionality, network setting, security parameters, or the computational resources. Furthermore, benchmarks are often chosen in favor of the proposed approach and potential weaknesses are downplayed. Thus, it is hard if not impossible for a privacy engineer without substantial experience in STC to make an objective and informed choice among the frameworks.

In this work, we conduct an extensive performance analysis of STC frameworks available as of January 2015 in order to address this problem[1]. The following are our key contributions:

Performance Evaluation: The main contribution of this paper is a performance comparison study incorporating five different STC frameworks, namely *Fairplay* [16], *CBMC-GC* [14], *mightbeevil* [7], *TASTY* [8], and *SeComLib* [11]. By implementing different benchmarks and varying network parameters as well as computational resources, we achieve a fair and objective comparison. Our results reveal large differences in the performance of the STC frameworks, emphasizing the importance for a privacy engineer to choose the right framework for a given task and deployment scenario.

Qualitative Analysis: Based on our experiences gained from implementing the various benchmarks, we discuss the practical usability of the considered STC frameworks.

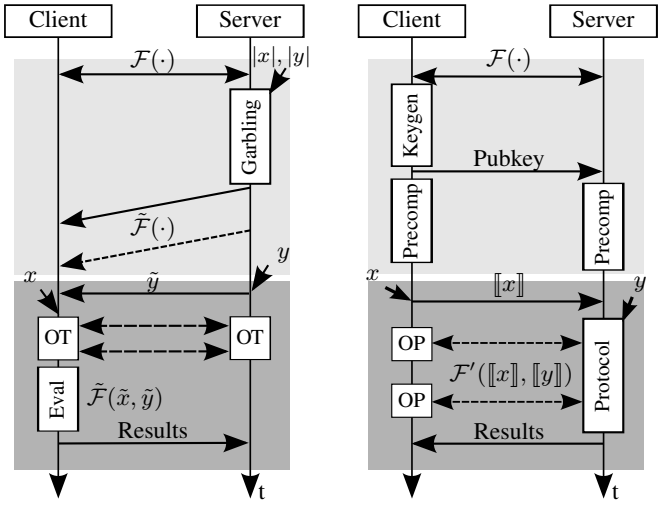
New Research Directions: Our qualitative and quantitative analysis indicate lacking support in STC approaches and frameworks for real-world problem sizes, mobile applications, and interactive scenarios. Based on these findings, we point out possible future research directions for STC.

The remainder of this work is structured as follows: In Section II, we concisely present the basic STC paradigms which are the basis of all considered STC frameworks. Section III presents our evaluation methodology and our choice of frameworks. We present our performance analysis in Section IV. Section V discusses our results and identifies optimization potential as well as new research directions. Section VI presents the related work and Section VII concludes this paper.

II. Secure Two-Party Computation

In this section, we present the basic theoretical foundations of STC which are important to understand our choice of benchmarks, the measurement methodology, the discussion of qualitative results, and new research directions. STC allows two mutually distrusting parties with private inputs x respectively y to compute a known functionality $\mathcal{F}(x, y)$ without anyone learning the private inputs. STC achieves this by cryptographically transforming the private inputs x and y such

[1] We provide the source code for our benchmarks and the entire set of evaluation results on our website: www.comsys.rwth-aachen.de/short/iwpe15/



(a) STC based on GC (b) STC based on HE

Fig. 1: Flowchart for STC based on a) GC and b) HE divided in precomputation (light grey) and online phase (dark grey).

that they are not revealed to third parties but can still be used to jointly compute $\mathcal{F}(x, y)$. As STC exhibits a performance asymmetry between the two parties, we refer to them as *client* $\mathcal{C}$ with private input x and *server* $\mathcal{S}$ with private input y in the following. Today, two predominant approaches for STC are co-existing: *Garbled Circuits* (GCs) and *Homomorphic Encryption* (HE). All frameworks we compare in this paper are based on either one or a combination of these two approaches. A qualitative flowchart for both approaches is depicted in Figure 1. It is important to differentiate between the precomputation phase (light grey), comprising operations that can be performed before the concrete inputs are known, and the online phase (dark grey), comprising those operations which require the concrete inputs. HE-based STCs, e.g., usually require encrypted randomness that can be precomputed in order to speed up the operations on the actual inputs. We now describe the two STC approaches and their combination in more detail.

A. Garbled Circuits

GCs were proposed by Yao in 1982 [17] as the first approach to STC and later proven practical by Malkhi et al. [16]. In the following, we present GCs using Figure 1(a).

Precomputation phase: In the precomputation phase, the GC approach requires that the functionality $\mathcal{F}$ that should be computed is represented as a Boolean circuit. A Boolean circuit representation of $\mathcal{F}$ can be automatically constructed, e.g., with an adequate compiler from a program written in a higher level language [16]. Compiling the Boolean circuit can be very time and memory consuming, but needs to be done only once as the constructed circuit can be reused for multiple runs (thus, it is omitted in Figure 1(a)). After the circuit has been constructed, $\mathcal{S}$ proceeds by encrypting and permuting the truth table entries for each circuit gate. This step is called *garbling* and we refer to the garbled circuit as $\tilde{\mathcal{F}}(\cdot)$. In order to construct size-efficient circuits, $\mathcal{S}$ at this point requires knowledge on the length of the inputs, denoted by $|x|$ and $|y|$. For further details on (garbled) circuit construction and numerous optimizations we refer to [16], [18], [19]. After garbling the circuit, $\mathcal{S}$ sends the garbled circuit $\tilde{\mathcal{F}}(\cdot)$ to $\mathcal{C}$. Since $\tilde{\mathcal{F}}(\cdot)$ can become quite large, this transfer can cause significant overheads (slanted arrows in Figure 1(a)). However, up to here, all operations can be precomputed if $|x|$ and $|y|$ are known.

Online phase: The online phase of the protocol can start as soon as the concrete inputs x and y are known. For this, $\mathcal{S}$ garbles the bits of its input y with the keys used to encrypt the circuit $\mathcal{F}(\cdot)$ (see [16], [18] for details) and transfers the resulting garbled input $\tilde{y}$ to $\mathcal{C}$. Since $\tilde{y}$ appears seemingly random to $\mathcal{C}$, it does not learn anything about $\mathcal{S}$'s private input y. To create its own garbled input $\tilde{x}$, $\mathcal{C}$ obtains the keys for the bits of its own private input x over an Oblivious Transfer (OT) protocol from $\mathcal{S}$. This prevents $\mathcal{S}$ from learning anything about x. For each input bit, one run of the OT protocol is required. This can cause severe processing and communication overheads. However, state-of-the-art OT protocols perform significant parts in the precomputation phase. After obtaining $\tilde{x}$ and $\tilde{y}$, $\mathcal{C}$ evaluates $\tilde{\mathcal{F}}(\tilde{x}, \tilde{y})$ by decrypting the garbled circuit gate by gate and finally obtains the result.

B. Homomorphic Encryption

HE schemes allow to compute specific arithmetic operations under encryption, e.g., the Paillier cryptosystem [20] allows addition, while ElGamal [21] is multiplicatively homomorphic. Although Fully Homomorphic Encryption (FHE) schemes that provide *both* addition and multiplication on cipher texts exist, they currently still cause prohibitive overheads. Thus, multiplication for Paillier or addition for ElGamal is currently more efficiently realized by an interactive protocol where the client helps the server to perform the respective operation. Using secure addition and multiplication, $\mathcal{C}$ and $\mathcal{S}$ can securely evaluate a representation of $\mathcal{F}$ as an arithmetic circuit as depicted in Figure 1(b).

Precompuation phase: We assume that $\mathcal{S}$ and $\mathcal{C}$ have agreed upon a protocol $\mathcal{F}'(\cdot)$ for arithmetically evaluating $\mathcal{F}(\cdot)$. Usually the privacy engineer builds $\mathcal{F}'(\cdot)$ by hand from different higher-level sub-protocols offered by the framework of choice, e.g., for comparison, which are realized using secure additions and multiplications. $\mathcal{C}$ then generates a key pair for the chosen homomorphic crypto system and shares the public key with $\mathcal{S}$. Importantly, this key pair needs to be generated only once. Hence, $\mathcal{S}$ could already be in possession of the public key from previous computations. To considerably speed up the online phase, both parties can then precompute several operations, e.g., randomness for encryption or additive blinds.

Online phase: $\mathcal{C}$ triggers the start of the protocol by sending its encrypted input $[\![x]\!]$ to $\mathcal{S}$. Then, $\mathcal{S}$ evaluates $\mathcal{F}'$ on $\mathcal{C}$'s encrypted input $[\![x]\!]$ and its own input y. For this, $\mathcal{S}$ can perform some operations locally, e.g., addition and scalar multiplication for Paillier, while other operations, e.g., ciphertext multiplication or comparisons, have to be realized through interactive protocols involving $\mathcal{C}$. This interaction has a major impact on the communication overheads and performance of HE-based STC. Finally, $\mathcal{S}$ obtains the encrypted result and sends it to $\mathcal{C}$ who decrypts it, and, if desired, gives it to $\mathcal{S}$.

C. Hybrid Approaches

The GC approach is based on Boolean logic and, thus, logical operations can be performed quite efficiently, while

arithmetic operations are more expensive. Contrary, arithmetic operations can be handled quite efficiently with HE. It thus is promising to combine both approaches and choose between them depending on the operation to be computed. This can be realized by converting between HE and GC at runtime [18].

III. Methodology

Our goal is to provide privacy engineers with a dependable and comparable performance evaluation of recent STC frameworks, compilers, and libraries (Section III-B). To achieve this, we evaluate the selected frameworks along six different benchmarks (Section III-A), four network settings, and two different computational resource settings (Section III-C) that bring out the advantages and disadvantages of each framework.

We especially emphasize that we take the point of view of a privacy engineer who regards STC only as a tool for engineering privacy-preserving systems. Such a privacy engineer wants to use the considered frameworks *out-of-the-box* and cannot be expected to significantly modify or optimize the chosen STC framework according to his needs. This approach has three notable consequences: First, we compare the STC frameworks in their current state, i.e., including all their differences in the underlying approach, protocols, and implementations. Second, we reduce any modifications for implementing our benchmarks to the minimum and implement missing functionality only according to standard solutions from literature. Finally, we implement our benchmarks in a straightforward way guided only by the frameworks' available documentation and examples and with the background knowledge presented in Section II.

A. Benchmarks

We use six different benchmarks to compare the different STC frameworks. In the following, we use $[x]$ and $[y]$ to denote the garbled or encrypted input of the client $\mathcal{C}$ and server $\mathcal{S}$, respectively. We write $[X]$ and $[Y]$ if these inputs are vectors and $[X]||[Y]$ for the concatenation of these vectors. If not stated otherwise, we assume a length of $l = 32$ bit per input.

1) Addition and Multiplication: Arithmetic operations are the basis for nearly every real-world use case of STC [2], [4], [5], [22]. Thus, our first two benchmarks consist of secure additions $\texttt{ADD}([x],[y])$ and multiplications $\texttt{MULT}([x],[y])$.

2) Minimum and Argminimum: Determining the minimum and the corresponding argument of a vector of numbers is an important building block, e.g., for secure nearest neighbor search [23]. Our third benchmark $\texttt{MIN}([X],[Y])$ is thus to find the minimum element in the combined input $X|Y$. Additionally, in a fourth benchmark, we measure the time for finding the argminimum, i.e., $\texttt{ARGMIN}([X,I_X)],[Y,I_Y])$, where I_X and I_Y are the indices corresponding to X and Y and the output is the smallest element from the combined vectors X and Y and its corresponding index from I_X or I_Y.

3) Matrix Multiplication: In our fifth benchmark, $\texttt{MATRIXMULT}([X],[Y])$, the inputs X and Y are matrices which are multiplied according to standard matrix multiplication, i.e., we separately compute each $Z_{i,j} = \sum_{k=1}^{n} X_{i,k} \cdot Y_{k,j}$. Besides being an indispensable operation in many algorithms, matrix multiplication is an interesting benchmark because it allows the evaluated frameworks to demonstrate their use of parallelization.

	Fairplay	**SeComLib**	**TASTY**	**mightbeevil**	**CBMC-GC**
Approach	GC	HE	GC/HE	GC	GC
Type	Compiler	Library	Interpreter	Framework	Compiler
Language	SFDL	C++	TASTYL	Java	ANSI-C
Network	✓	✗	✓	✓	✓
Addition	✓	✓	✓	✓	✓
Multiplication	✗	✓	(✓)	✗	✓
Comparison	✓	✓	✓	✓	✓
Minimum	✗	✓	(✓)	(✓)	✗
Argmin	✗	✗	✗	✗	✗

TABLE I: Comparison of the STC frameworks.

4) Sorting: We choose sorting, $\texttt{SORT}([X],[Y])$, as our sixth benchmark, because it requires a mix of logical operations (comparisons) and arithmetic operations (for the conditional swaps in HE-based STC). However, most popular sorting algorithms such as Quicksort or Mergesort require array access at private locations when implemented as STCs. Private array access is not available in *mightbeevil*, *TASTY*, and *SeComLib* and is also far from trivial to implement. In particular, according to our methodology, we cannot expect a privacy engineer employing STC merely as a tool to implement this functionality. Thus, we choose Bitonic sorting to implement $\texttt{SORT}([X],[Y])$, which does not require private array access and can thus be much more easily implemented by a privacy engineer using the considered framework. We use the same algorithm for all frameworks to maintain comparability, even though *Fairplay* and *CBMC-GC* support private array access.

B. STC Frameworks

In this section, we introduce the considered STC frameworks in chronological order. Table I summarizes the most important features of these frameworks and whether they support the basic operations required to implement our benchmarks[1].

1) Fairplay: Malkhi et al. propose the *Fairplay* framework [16], a proof-of-concept of Yao's garbled circuits [17]. Fairplay provides a compiler that translates a program written in the special *SFDL* language into a garbled circuit as well as a runtime environment, which handles the evaluation of the resulting garbled circuit. Since most optimizations for GCs were proposed only after Fairplay was released, Fairplay only optimizes the size of the compiled Boolean circuit and applies the point-and-permute decryption technique [16]. Fairplay's SFDL language provides arithmetic operators, bit wise operators, and comparison operators. Multiplication as well as minimum/argminimum are not natively supported by SFDL and we had to implement this functionality ourselves.

2) SeComLib: The *Secure Computation Library* (SeComLib) [11] developed at the Technical University Delft is a C++ library comprising different homomorphic encryption systems and STC protocols. It has been used to realize different applications, including face recognition [2] or recommender systems [4]. The library natively supports the required arithmetic and logical operations, comparisons, and also offers a minimum algorithm. An argminimum building block was not available, so we had to add this feature. Unlike the other frameworks, *SeComLib* offers no network support and client and server communicate locally over shared memory. For comparability, we thus added basic networking support using Boost Asio.

3) TASTY: Henecka et al. propose the *TASTY* interpreter [8] and use it to implement secure face recognition as well

as AES with distributed keys. Notably, *TASTY* is the only framework that uses the hybrid STC approach. It defines its own language called *TASTYL*, which is a subset of the Python programming language. The authors claim that *TASTYL* provides a wide range of arithmetic operations on encrypted and logical operations on garbled values. However, we found that multiplication of cipher texts did not work even in the provided tests. Thus, we had to reimplement the standard approach to ciphertext multiplication using HE and additive blinds [18]. Also, we found that garbled vectors for minimum selection could not be constructed from plain inputs but only from homomorphic vectors which incurs significant, unnecessary overheads. To avoid this indirection and offer fair comparison, we implemented an alternative minimum algorithm using the garbled value types and the built-in comparisons directly.

4) mightbeevil: Huang et al. propose *mightbeevil* [7], a Java framework for GCs which, e.g., has been used for secure biometric identification [22]. *mightbeevil* provides no language support but only circuit building blocks, which directly involves the privacy engineer in building and optimizing the resulting circuits. This approach follows the authors' claim that hand-optimized circuits are more efficient than automatically optimized circuits such as those built by *Fairplay*, *CBMC-GC*, or *TASTY*. The main contribution of *mightbeevil* is a special pipelined runtime environment, which allows to stream and evaluate gates as soon as they are created. *mightbeevil* provides circuit building blocks for addition and an optimized comparator building block for selecting the minimum of two values. We generalized minimum selection to n inputs and added building blocks for multiplication and argminimum.

5) CBMC-GC: Holzer et al. present *CBMC-GC* [14], a compiler which takes a protocol written in ANSI-C and automatically builds a garbled circuit from it using most known circuit optimization techniques and aggressive SAT-based optimizations using the CBMC model checker. *CBMC-GC* uses the *mightbeevil* framework as runtime environment for handling inputs and the necessary network communication. Because CBMC-GC actually only supports the direct language features of C but not, e.g., the standard library, addition, multiplication, and comparison were available but we had to implement minimum and argminimum algorithms ourselves.

6) Not considered: Apart from the mentioned five frameworks, we surveyed further approaches but did not include them for our evaluation for different reasons, which we briefly summarize here. Schropfer et al. propose $L1$ [13], an intermediate language which abstracts from different GC and HE-based STC approaches. As another hybrid approach, $L1$ would have been very interesting for our evaluation, however, neither source code nor executables were available publicly or on demand. Further, the Secure Multiparty Computation Language (SMCL) [24] was not considered, because it is officially discontinued and the code is no longer available. Finally, MacKenzie et al. propose a compiler for automatic generation of STCs in [12] based on a 2-out-of-2-secret sharing scheme. However, since their compiler supports only field operations over $\mathbb{Z}_q$, i.e., addition, multiplication, and inversion, our minimum, argminimum, and sorting benchmarks could not have been implemented without significant additions, which we consider inadequate for a privacy engineer.

We deliberately put our focus on two-party computation, but acknowledge that there exists an even greater variety of Secure Multi-Party Computation (SMC) frameworks, compilers and libraries, e.g., the open-source *SEPIA* library [9], the closed-source *Sharemind* framework [10], or most recently the *PICCO* compiler for C [15]. Because SMC follows a different setting than STC, both approaches are difficult to compare and we thus did not consider these frameworks. However, a dependable and fair comparison of SMC frameworks is just as much needed as for STC.

	Server	**Desktop Client**	**Mobile Client**
CPU	Intel Xeon E5-2650	Intel i7-4770S	Intel Atom N550
Speed	2.6 GHz	3.1 GHz	1.5 GHz
Cores	16, 32 Threads	4, 8 Threads	2, 4 Threads
RAM	32 GB	16 GB	1 GB

TABLE II: The server and clients used in the evaluation.

	LAN	**Wi-Fi**	**WAN**	**3G**
Bandwidth	100 Mbit/s	16 Mbit/s	16 Mbit/s	370 kbit/s
Latency	0.3 ms	5 ms	40 ms	300 ms

TABLE III: The four different network settings.

C. Device and Network Scenarios

Table II summarizes the devices used for the evaluation. As server we chose a high powered machine that can execute up to 32 threads on 16 cores with 2.60 GHz and 32 GB RAM. This server is able to handle the huge memory demand during compilation of garbled circuits. As clients we chose two different devices, one similar to a standard *desktop computer* with moderate resources and one similar to a lesser powered *mobile device*. The low computational resources of the mobile device should make evident the different processing demands among the considered framework and expose excessive overheads. We chose a Lenovo Ideapad S10-3 as mobile platform, because most of the considered frameworks could not directly be ported to modern mobile platforms such as Android or iOS.

We further consider four different network settings that feature different bandwidth and latency (Table III). The most performant setting is the LAN setting, where hosts are connected over a 100 Mbit/s switch. The Wi-Fi and WAN settings feature lower bandwidths which should already limit STC protocols that incur excessive traffic. The Wi-Fi setting provides a realistic impression of the performance of STC protocols in mobile but local scenarios, e.g., using Wi-Fi direct. The WAN setting features a significant latency that should be notable in STC protocols with high amount of interaction. The fourth and most challenging setting, the 3G setting, provides only little bandwidth and a high delay, which gives a realistic impression of the feasibility of STC in wide-area mobile scenarios.

IV. Performance Comparison

We executed 10 measurement runs for the combination of each of the benchmarks presented in Section III-A with each client device (Table II), all network settings (Table III), and with varying input sizes n (i.e., the number of operations of the individual benchmark). We measure the time until both client and server side have completed the computation (as illustrated in Figure 1), including precomputations. However, we exclude the time necessary for compiling the Boolean circuit, as this needs to be done only once (c.f. Section II-A). As several experiments quickly reached a run time in the order of dozens

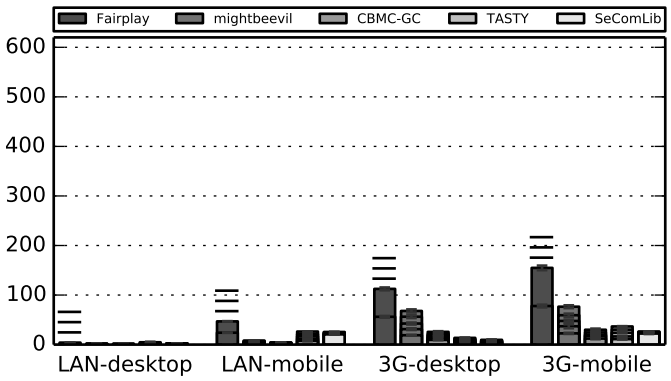


Fig. 2: Runtimes [s] for Addition.

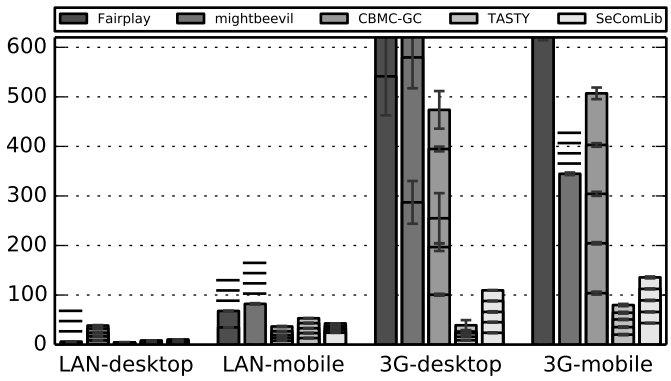


Fig. 3: Runtimes [s] for Multiplication.

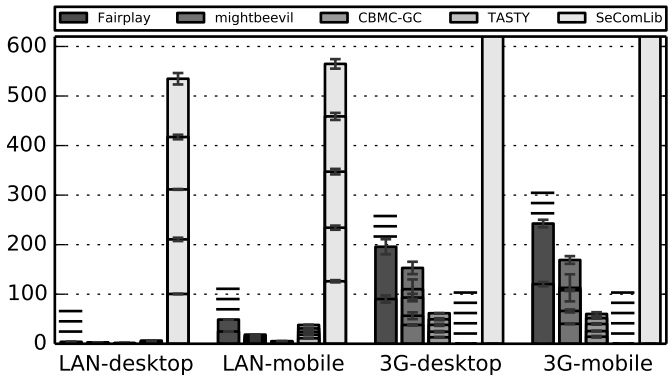


Fig. 4: Runtimes [s] for Minimum.

of minutes to hours, we cut off all experiments at a time limit of $10\,\text{min} = 600\,\text{s}$ per run as longer run times make a framework infeasible for privacy engineers.

Our extensive performance measurements provide an abundance of interesting results which we unfortunately cannot cover in full detail here. Thus, in this paper we focus on the most important and interesting settings and results[1]. In particular, we limit our analysis to the LAN and 3G network settings which represent the extremes of the considered settings. The following plots show the measurement results grouped by the combination of network setting and client device. In each group, one bar corresponds to one of the five frameworks and shows the cumulative runtime for the different input sizes n. When a framework failed for a particular input size it also always failed for all larger input sizes, which we marked by horizontal lines above the respective bar, the number of lines indicating the number of failed input size levels.

Addition: For addition (Figure 2), all frameworks except *Fairplay* were able to handle the $n = 20, 40, 60, 80, 100$ inputs in all networks and on both client devices (Figure 2). *Fairplay* failed to compile circuits for the last three input sizes $n = 60, 80, 100$. Surprisingly and contradicting the STC folklore, the GC-based approaches *CBMC-GC* and *mightbeevil* performed better than the HE-based approaches *TASTY* and *SeComLib* in the LAN setting. This is due to the comparably low computational effort for garbling circuits compared to the encryption of the inputs. The comparison of the desktop with the mobile client for which the HE-based approaches show worse performance substantiates this result. However, when switching to the bandwidth-constraint $3G$ setting, network overhead for transmitting the potentially large GCs becomes larger than the computation overhead for HE-based approaches which had to transfer far less data. Thus, in these scenarios the HE-based approaches outperform the GC-based ones.

Multiplication: For multiplication (Figure 3), HE-based STC were able to play out their widely assumed strength, with *TASTY* and *SeComLib* delivering the best performance in all networks and on all clients (Figure 3). Notably, *Fairplay* failed to handle input sizes $n = 60, 80, 100$ due to failure of compilation, while *mightbeevil* failed on the mobile device during streaming of circuits even for $n = 40$. The highly optimized circuits of CBMC-GC are only able to compete with the HE-based approaches in the GC-favorable LAN network setting. Comparing to the 3G setting, we again observe that performance of the GC-based approaches is severely limited by the excessive network overheads for transmission of the garbled circuits. *Fairplay* and *mightbeevil* exceeded the 10 min limit already for 20 and 40 inputs, respectively. *CBMC-GC* manages all inputs sizes below the cut-off threshold but is still approximately 4-5 times slower than the HE-based approaches.

Minimum: As noted earlier (Section II-C), GC-based STC is rooted in Boolean logic and thus expected to perform logical operations very efficiently. Indeed, minimum computation (Figure 4) was most efficiently handled by the GC-based approaches (Figure 4), while *Fairplay* again cannot handle $n = 60, 80, 100$ inputs. Note that *TASTY* switches to GC-based STC for minimum computation, thus *SeComLib* presents the only HE-based approach for this task. *SeComLib* implements secure comparisons in a protocol that requires one round of communication per input bit which already causes excessive overheads even in the LAN setting and clearly becomes infeasible in the 3G setting. The computational overheads, judging by the comparison of desktop and mobile devices are almost negligible in comparison. We note that more efficient comparison protocols have been proposed for HE-based STC but are not implemented by *SeComLib*. While *mightbeevil* and *CBMC-GC* build efficient circuits and thus maintain reasonable performance in the 3G setting, in our experiments *TASTY* suffered from time out errors in the 3G setting on both the desktop and mobile device.

Argminimum: For argmin computation (Figure 5) all evaluated frameworks exhibit overheads within the same order of magnitude and consistent with the previous results on minimum computation. This is expected as argmin is essentially a minimum computation combined with comparably simple operations to keep track of the argument leading to the minimum value. As before, *mightbeevil* and *CBMC-GC*

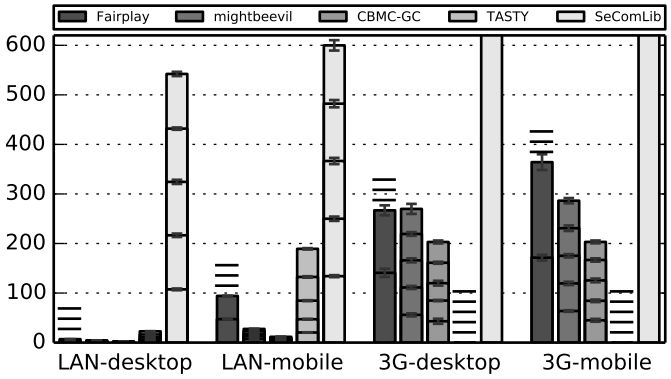


Fig. 5: Runtimes [s] for Argminimum.

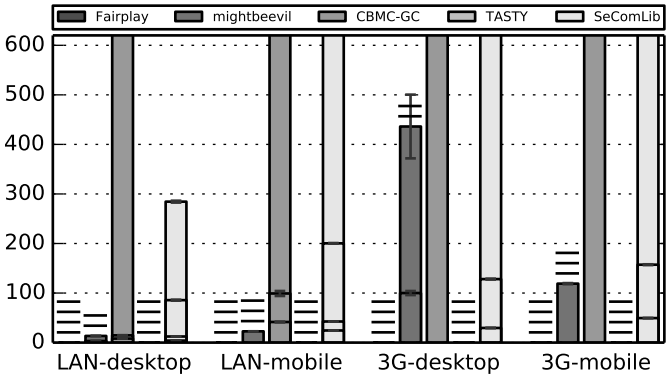


Fig. 6: Runtimes [s] for Matrix multiplication.

present the best choices, while *TASTY* fails to run in the 3G setting as before and *SeComLib* runs longer than 10 min even for only $n = 20$ inputs due to the slow comparison protocol.

Matrix Multiplication: For this benchmark, we multiplied two matrices of increasing size, i.e., $3 \times 3, 5 \times 5, 10 \times 10$, and 15×15. Matrix multiplication (Figure 6) showed the most diverse results among all our benchmarks. First, *Fairplay* did not manage to build circuits, not even for the multiplication of two 3×3 matrices, i.e., a circuit comprising 27 multiplications and 18 additions, which is consistent with our results on addition and multiplication. *mightbeevil* could handle 5×5 matrices in the LAN setting but failed for this size in the 3G setting. Interestingly, *mightbeevil* is much faster at matrix multiplication than its performance in the isolated addition and multiplication benchmarks suggest. Considering, e.g., that *mightbeevil* needs approximately 290 s to multiply 20 values in the 3G setting on the desktop client it is surprising that it can handle the 3×3 matrix multiplication, which requires 27 multiplications, in less than 100 s in the same setting. A possible reason is that matrix multiplication involves, in this example, only $3 \cdot 3 = 9$ inputs, while the addition and multiplication benchmark involve 20. Thus matrix multiplication needs less Oblivious Transfer runs to exchange inputs. We validated this suspicion by running multiplication for *mightbeevil* again but this time repeatedly $n = 20, ..., 100$ times on the same input. Here, *mightbeevil* was up to four times faster which explains the performance gain. *CBMC-GC* was able to handle multiplication of 10×10 matrices, but did not exhibit an increased performance as *mightbeevil* did. This indicates potential for optimization in *CBMC-GC*. We could not obtain any results for *TASTY*, since *TASTY's* built-in functionality to multiply homomorphically encrypted values was not working when the two inputs were not encrypted by the same party. For the multiplication benchmark, we therefore implemented secure multiplication manually, but *TASTY's* lacking support for functions or loops over encrypted values prevented us from implementing matrix multiplication using our multiplication subprotocol as a building block. In consequence, we could not implement matrix multiplication in *TASTY*. The other HE-based approach, *SeComLib*, was the only approach able to handle all input sizes. However, it broke the 10 min time limit for 10×10 matrices in the 3G setting and for 15×15 matrices on the mobile client in the LAN setting.

Sorting: For this benchmark, client and server each provided a vector of length $n = 8, 16, 32, 64$ which were joined and sorted. This benchmark (Figure 7) pushed most frameworks to their limits. *Fairplay* could compile circuits only for $n = 8$ but failed to execute them. *mightbeevil* was by orders of magnitude faster than the other frameworks in all considered settings. It ran out of memory on the mobile device only for $n = 64$ and timeouts occurred in the two 3G settings only for input sizes $n = 64$ and $n = 32$, respectively. *CBMC-GC* ran out of memory on the mobile client already for input sizes $n = 16, 32, 64$ and timed out on the desktop client for $n = 32$ and $n = 16$ in the LAN and 3G setting, respectively. *TASTY* showed medial performance in the LAN setting, but failed entirely in the 3G setting again due to timeouts during execution. *SeComLib* was able to handle all settings but timed out for all but the LAN evaluation settings with $n = 8$ inputs.

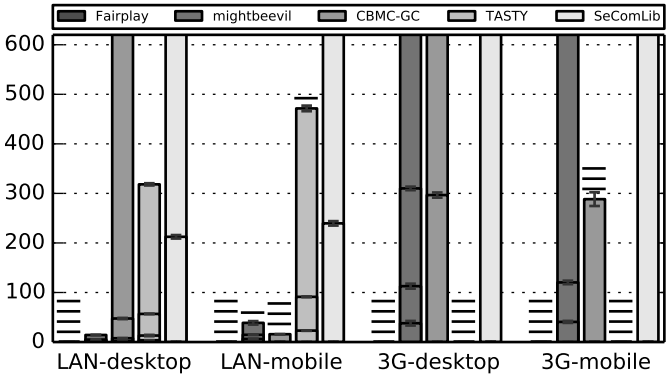


Fig. 7: Runtimes [s] for Sorting.

V. Discussion

We first qualitatively discuss usability aspects of the frameworks then proceed with a summary of our results and point out possible improvements and new research directions.

A. Qualitative Analysis

Fairplay's own protocol language has very low expressiveness, e.g., it does not allow recursion and only fixed sized loops. However, the available functionality works both reliably and is intuitive even to inexperienced STC developers. Thus, benchmarks that use only the built-in functionality were easy and anything beyond that hard to implement. *Fairplay* is only available as 32-bit Java bytecode and thus limited to 4 GB of RAM usage, which led to the highest failure rate of all compared frameworks (c.f. Table IV).

mightbeevil requires programming on circuit level, which presents a high hurdle to privacy engineers using STC merely

	Fairplay		mightbeevil		CBMC-GC		TASTY		SeComLib	
LAN+Desktop	71.4	71.4	7.1	**7.1**	3.6	14.3	17.9	17.9	**0.0**	10.7
LAN+Mobile	71.4	71.4	28.6	28.6	14.3	17.9	21.4	21.4	**0.0**	**10.7**
3G+Desktop	71.4	75.0	7.1	**21.4**	3.6	25.0	67.9	67.9	**0.0**	57.1
3G+Mobile	71.4	78.6	28.6	32.1	14.3	**25.0**	67.9	67.9	**0.0**	57.1
Average	71.4	74.1	17.9	22.3	8.9	**20.5**	43.8	43.8	**0.0**	33.9

TABLE IV: Percentage of failed evaluation settings, excluding and including timeouts, respectively.

as a tool. Also this involves a lot of repetition and boilerplate code resulting in by far the highest amount of Lines of Code (LOCs)[1] for the implementation of benchmarks. On the other side, *mightbeevil's* low level approach allows for optimizations down to the level of individual gates. This makes *mightbeevil* a good choice for building STC frameworks with better usability on top, as done by *CBMC-GC*.

CBMC-GC allows to develop STCs in ANSI-C which is convenient and even allows the developer to use existing and well established tool chains to build, test, and debug protocols before running them as STCs. Together with *Fairplay*, *CBMC-GC* also required by far the smallest number of LOCs to implement our benchmarks. On the downside, handling in- and outputs was very cumbersome and required to write additional scripts to generate source code automatically.

TASTY, in its current state, was very hard to use. We found that some crucial functionality, such as HE-based multiplication or creation of garbled values from input, was broken. *TASTY's* source code is not available and documentation is scarce, which aggravates these issues and often prevented work-arounds. Furthermore, the runtime environment regularly failed in the $3G$ network setting. However, the results we could obtain are promising and show that a hybrid approach can provide a more balanced performance. *TASTY's* approach required more LOCs than the compiler approach taken by *Fairplay* and *CBMC-GC*, but still far less than *mightbeevil*.

SeComLib was well documented and maintained and implementing our benchmarks was mostly straightforward, except for the need to manually implement network support. The LOCs were comparable to *TASTY*. Notably, *SeComLib* was the only framework able to handle all our benchmarks, but did still time out frequently especially for the logical tasks.

B. Results Summary and New Research Directions

We conclude the discussion of our results by summing up the feasibility of our benchmarks in the considered frameworks. There were two error sources during the evaluation: i) complete failure to compile or evaluate a circuit (marked in the plots by a number of horizontal lines corresponding to the number of failed input sizes) and ii) exceeding the 10 min time limit (bars which exceed the limits of the plot). Table IV sums up how often these errors occurred relative to the total number of evaluation settings (benchmarks $\times$ networks $\times$ devices $\times$ input sizes). Each cell lists two percentages, i.e., the percentage of failed evaluation settings excluding and including successful but timed-out runs, respectively. In each row the respective minima are marked in bold.

Our general insight from Table IV and the previously discussed results is that although STC has received a lot of attention from theoretical research, the existing STC frameworks still need major improvements in the following directions, some of which open exciting new research directions.

Real-world Problem Instances: The frameworks failed not randomly but for the larger input sizes and at that quite often. Although *CBMC-GC* provides a notable exception, STC frameworks need to improve towards handling larger input sizes so that STC becomes a viable solution at least for small real-world problems. Here, *mightbeevil's* streaming approach is promising but needs to be developed further.

Balanced Performance: The performance of the frameworks was very imbalanced mainly due to the underlying strengths and weaknesses of the chosen STC approach. *TASTY* showed that a hybrid approach can potentially achieve a better balance and maintain a reasonable over all performance. We argue that STC frameworks should better balance and optimize their *overall* performance so that a privacy engineer is not forced to use different frameworks for different tasks.

Parallelization: Our benchmarks, e.g., matrix multiplication, offer much potential for parallelization. Only *mightbeevil* and *CBMC-GC* made use of parallelization during runtime, which improved performance in the LAN setting but had little effect for 3G. Notably, parallelization has shown huge performance gains in SMC frameworks [15]. We thus propose future research on how to use parallelization in STC to embrace the development towards multi-core architectures. Similar to [15], parallelization should be supported on language-level so that even an inexperienced privacy engineer can benefit from it.

Mobile STC: Mobile scenarios and applications are already pervasive today and the demand for privacy-preservation in these applications is evident. However, as the difference between the numbers including and excluding timeouts in Table IV shows, all frameworks experienced significant problems in the constrained 3G network setting and for the mobile, less powered client. STC is thus not yet a viable tool for mobile applications. In order to address mobile STCs, frameworks need to significantly reduce networking overhead as well as to account for the resource asymmetry, e.g., by shifting more computations to the server. Especially the direction of STC under asymmetric resource distribution has received only little attention and provides an interesting new research direction.

Interactive STC: In *interactive STCs* neither the input nor its length are known or bounded a priori, but arrive over time during the run of the application. Examples are, e.g., WiFi-based (indoor) localization [25] and speech recognition [26]. The HE-based STC frameworks can handle these applications, but they show inferior performance in logical tasks which are heavily used in both examples [25], [26]. GC-based frameworks are currently only efficient if the input length is known and circuits can be built and optimized accordingly. They thus do not support interactive STC. Here, hybrid STC is promising to overcome these limitations, but needs to be researched further for interactive STCs.

VI. Related Work

The performance of each of the evaluated frameworks is also evaluated in the corresponding papers. However, the measurements are performed in an incomparable manner and thus do not allow a privacy engineer to identify the right

tool for the current task. In [16], *Fairplay* is benchmarked for addition and comparisons in a LAN and WAN network setting for a single client device. *mightbeevil* is benchmarked in [7] for Hamming and Levensthein Distance, Smith Waterman similarity, as well as AES against the fastest known approaches at that time, including *TASTY*. However, no details regarding the network or used devices are given and only ultra short-time security levels are used. The *CBMC-GC* paper [14] provides the most comprehensive benchmarks and considers a LAN and a WAN scenario, but does not compare to other frameworks. For *SeComLib* [11] only performance results for different application scenarios, e.g., face recognition [2] and recommendations [4], are available, without any comparison to implementations of these use cases in other frameworks. *TASTY* [8] also uses concrete applications as benchmarks, i.e., face recognition, set intersection, and AES, and compares itself against *SeComLib's* implementation of face recognition and *Fairplay's* AES implementation. Different network settings or client devices are not considered. Overall, the results presented in these works are difficult to compare due to the differences in i) network parameters, ii) device resources, and iii) chosen benchmarks. In contrast, our work fixes the network settings, client devices, and benchmarks in order to provide a fair and dependable performance comparison of these frameworks.

Schröpfer et al. [27] present an analytical performance model for forecasting the runtime of STCs, which they validate by an empirical study of the problem of secure divisions implemented in their own $L1$ framework [13]. Unfortunately, they do not generalize, parameterize, and validate their performance model for other STC frameworks. This would be interesting future work, for which our presented benchmarks already provide a good basis. Blanton also considers the problem of secure divisions and provides an empirical study in [28], but focuses on different division protocols rather than different frameworks, network settings, or devices. While these works rather focus on analyzing the theoretical differences in the performance of STC approaches and protocols, ours takes a rather end-user centric point-of-view and focuses on the practical performance of concrete STC frameworks.

VII. Conclusion

STC has proven feasible in several use cases, e.g., face recognition [2], [3], recommender systems [4], and even genomic testing [5]. However, adoption of STC in privacy engineering is still scarce due to significant performance (Section IV), usability issues (Section V-A), and feasibility restrictions (Section V-B). These particularly challenge inexperienced developers of STCs. Recent STC frameworks have begun to tackle these issues, but are mostly evaluated in an incomparable manner which makes it difficult for privacy engineers to choose the right framework for their purpose. Thus, in our work, we carried out an objective and thorough performance evaluation of five recent STC frameworks using different benchmarks, network settings, and devices. While our performance results are mostly in favor of the newer GC-based approaches, we also observe that these approaches could not handle a significant part of the evaluation settings due to the huge overhead imposed by the Boolean circuit function representation. Based on the shortcomings identified in both our quantitative and qualitative analysis, we point out potential for improvements and the need for further research in STC. In conclusion, none of the considered framework presents the single optimal solution for privacy engineers. Instead, privacy engineers must make a choice that considers the type of operations used, the network environment, and the available resources on client *and* server. Our results provide privacy engineers with a thorough data basis for making this choice in an informed manner.

Acknowledgment: This work has been funded by the Excellence Initiative of the German federal and state governments.

References

[1] PEW Research, "Public Perceptions of Privacy and Security in the Post-Snowden Era," http://www.pewinternet.org/?p=12225, 2014.

[2] Z. Erkin *et al.*, "Privacy-Preserving Face Recognition," in *PETS*, 2009.

[3] A.-R. Sadeghi *et al.*, "Efficient Privacy-Preserving Face Recognition," in *ICISC*, 2009.

[4] Z. Erkin *et al.*, "Generating Private Recommendations Efficiently Using Homomorphic Encryption and Data Packing," *IEEE Trans. Inf. Forensics Security*, vol. 7, no. 3, 2012.

[5] E. De Cristofaro *et al.*, "Genodroid: Are Privacy-preserving Genomic Tests Ready for Prime Time?" in *ACM WPES*, 2012.

[6] I. Damgård *et al.*, "Asynchronous Multiparty Computation: Theory and Implementation," in *PKC*, 2009.

[7] Y. Huang *et al.*, "Faster Secure Two-party Computation Using Garbled Circuits," in *USENIX Security*, 2011.

[8] W. Henecka *et al.*, "TASTY: Tool for Automating Secure Two-party Computations," in *ACM CCS*, 2010.

[9] M. Burkhart *et al.*, "SEPIA: Privacy-preserving Aggregation of Multi-domain Network Events and Statistics," in *USENIX Security*, 2010.

[10] D. Bogdanov *et al.*, "Sharemind: A Framework for Fast Privacy-Preserving Computations," in *ESORICS*, 2008.

[11] "SeComLib," http://cybersecurity.tudelft.nl/content/secomlib.

[12] P. MacKenzie *et al.*, "Automatic Generation of Two-Party Computations," in *ACM CCS*, 2003.

[13] A. Schropfer *et al.*, "L1 – An Intermediate Language for Mixed-Protocol Secure Computation," in *IEEE COMPSAC*, 2011.

[14] A. Holzer *et al.*, "Secure Two-Party Computations in ANSI C," in *ACM CCS*, 2012.

[15] Y. Zhang *et al.*, "PICCO: A General-Purpose Compiler for Private Distributed Computation," in *ACM CCS*, 2013.

[16] D. Malkhi *et al.*, "Fairplay - a Secure Two-party Computation System," in *USENIX Security*, 2004.

[17] A. Yao, "How to generate and exchange secrets," in *IEEE FOCS*, 1986.

[18] V. Kolesnikov *et al.*, "From Dust to Dawn: Practically Efficient Two-Party Secure Function Evaluation Protocols and their Modular Design," *IACR Cryptology ePrint Archive*, 2010.

[19] V. Kolesnikov and T. Schneider, "Improved Garbled Circuit: Free XOR Gates and Applications," in *ICALP*, 2008.

[20] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in *EUROCRYPT*, 1999.

[21] T. ElGamal, "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," in *CRYPTO*, 1984.

[22] D. Evans *et al.*, "Efficient Privacy-Preserving Biometric Identification," in *NDSS*, 2011.

[23] S. Rane and P. Boufounos, "Privacy-Preserving Nearest Neighbor Methods," *IEEE Signal Process. Mag.*, vol. 30, no. 2, 2013.

[24] J. D. Nielsen and M. I. Schwartzbach, "A Domain-Specific Programming Language for Secure Multiparty Computation," in *PLAS*, 2007.

[25] J. H. Ziegeldorf *et al.*, "POSTER: Privacy-preserving Indoor Localization," in *WiSec*, 2014.

[26] M. Aliasgari and M. Blanton, "Secure Computation of Hidden Markov Models," in *SECRYPT*, 2013.

[27] A. Schröpfer and F. Kerschbaum, "Forecasting Run-Times of Secure Two-Party Computation," in *QEST*, 2011.

[28] M. Blanton, "Empirical Evaluation of Secure Two-Party Computation Models," CERIAS TR 2005-58, Purdue University, Tech. Rep., 2005.