Faster Secure Two-Party Computation with Less Memory

Wilko Henecka
School of Mathematical Sciences
University of Adelaide
Australia
wilko.henecka@adelaide.edu.au

Thomas Schneider
European Center for Security and Privacy by
Design (EC SPRIDE)
Technische Universität Darmstadt
Germany
thomas.schneider@ec-spride.de

ABSTRACT

Secure two-party computation is used as the basis for a large variety of privacy-preserving protocols, but often concerns about the low performance hinder the move away from non-private solutions.

In this paper we present an improved implementation of Yao's garbled circuit protocol in the semi-honest adversaries setting which is up to 10 times faster than previous implementations. Our improvements include (1) the first multi-threaded implementation of the base oblivious transfers resulting in a speedup of a factor of two, (2) techniques for minimizing the memory footprint during oblivious transfer extensions and processing of circuits, (3) compilation of sub-circuits into files, and (4) caching of circuit descriptions and network packets. We implement improved circuit building blocks from the literature and present for the first time performance results for secure evaluation of the ultra-lightweight block cipher PRESENT within 7 ms online time.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Cryptographic controls*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*

General Terms

Security, Algorithms

Keywords

Secure Computation, Garbled Circuits, Efficiency, Privacy

1. INTRODUCTION

Secure two-party computation, often called secure function evaluation (SFE), allows two mutually mistrusting parties to compute an arbitrary function on their private inputs without revealing any information about their inputs beyond the function's output. Although the real-world deployment of SFE was believed to be very expensive for a relatively long time, the cost of SFE has been dramatically reduced in the recent years thanks to many algorithmic improvements and automatic tools, as well as faster computing platforms and communication networks. SFE enables a large variety of privacy-preserving applications such as electronic auctions [NPS99], data mining [LP09b], or biometric identification [BG11, HMEK11], to name a few.

Although other approaches exist, most practical applications of SFE, including the ones listed above, are based on Yao's garbled circuits technique [Yao86] for which many improvements have been proposed (we give a summary in §2.1). In this paper we focus on secure two-party computation based on garbled circuits in the semi-honest adversary model. In this model, the adversary is assumed to be "honest-but-curious", i.e., he honestly follows the protocol specification, but tries to learn additional information from the messages seen. Although this adversary model is very weak, it allows to construct highly efficient protocols for many application scenarios, e.g., for constructing privacy-preserving protocols that protect against attacks by insiders or future break-ins. We strongly believe that pushing the performance limits of such protocols is essential in order to promote secure computation as conceivable alternative to using protocols without provable privacy guarantees.

In order to make SFE usable in practice, several frameworks with different properties have been proposed as summarized in Table 1. These frameworks allow an application developer to describe the functionality that needs to be computed securely on a high level and abstract from the details of the underlying protocol. Fairplay [MNPS04, BNP08] allows to describe the functionality to be computed in a high-level language which is compiled into a boolean circuit in an offline pre-computation phase. This compilation allows to perform global optimizations such as eliminating dead code. Subsequently, TASTY [HKS$^+$10] partitioned the garbled circuit protocol such that most expensive operations (w.r.t. both, communication and computation) are performed in the pre-computation phase. To reduce the memory footprint, VMCrypt [Mal11] introduced the concept of streaming, i.e., the garbled circuit is generated gate by gate and directly streamed into the network. To also reduce the memory footprint for the circuit, the programmer can compose the circuit by dynamically constructing and deconstructing sub-circuits. However, VMCrypt instantiates a new object for each gate such that its performance suffers from the additional overhead of garbage collection. Also in

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.
Copyright 2013 ACM 978-1-4503-1767-2/13/05 ...$15.00.

Table 1: Frameworks for GC-based secure two-party computation in the semi-honest adversaries setting.

Framework	Compilation	Streaming	Memory and Additional Overhead During Online Phase
Fairplay [MNPS04, BNP08]	Yes	No	$\mathcal{O}$(circuit size)
TASTY [HKS$^+$10]	Yes	No	$\mathcal{O}$(circuit size)
VMCrypt [Mal11]	No	Yes	depends on circuit, garbage collection
FastGC [HEKM11]	No	Yes	$\mathcal{O}$(max(size of sub-circuit)), garbage collection
GCParser [MZE12]	Yes	Yes	$\mathcal{O}$(max(size of sub-circuit)), garbage collection
[KSS12] (malicious, cluster)	Yes	Yes	$\mathcal{O}$(maximum working set(circuit)), usage counter
This Work	Yes	Yes	$\mathcal{O}$(max(max. working set(sub-circuit))), no online overhead

FastGC [HEKM11] the circuit is not compiled, but composed from sub-circuits and dynamically generated within a library. Also here, a new object is created for each gate of the sub-circuit which could be freed by the garbage collector when not used any more. GCParser [MZE12] extended the FastGC framework to read in a file which describes the way pre-defined sub-circuits should be put together; it also requires memory linear in the size of the sub-circuits. Most recently, the framework of [KSS12] implemented GC-based secure function evaluation in the malicious setting by exploiting the high degree of parallelism available in a cluster. In this framework, each gate carries a usage counter such that memory can be freed after the last use of the gate, but this requires additional overhead in the online phase.

1.1 Outline and Our Contributions

After giving related works in §1.2, an overview on Yao's garbled circuit protocol in §2.1.1, and a minor remark on the choice of security parameters in §2.2, we present the following contributions:

In §3 we present several optimizations that result in lower memory consumption and significantly better performance compared to previous frameworks. More specifically, we improve the implementation of base oblivious transfers using multi-threading (§3.1), reduce the memory consumption of oblivious transfer extensions (§3.2), compile sub-circuits into files (§3.4), and cache circuit descriptions and the communication (§3.5). In §3.3 we enhance streaming by considering both, the memory footprint of the circuit and the garbled circuit with no additional overhead in the online phase. As described in §1 and summarized in Table 1, most previous frameworks have a memory consumption which is linear in the size of the evaluated circuit. As shown in [JKSS10b] and implemented in the framework of [KSS12], it is sufficient to just hold the intermediate values in memory that are needed later on, called the working set. For example, the maximum size of the working set of a Karatsuba multiplication of two 128 bit values is 1,074 whereas the circuit has 57,000 gates. In VMCrypt [Mal11], the needed memory depends on the way a programmer creates and decorates circuit components, and the framework of [KSS12] requires additional overhead in the online phase to manage a usage counter and free unused memory (see §3.3 for details). The memory consumption of our engine for simultaneously streaming circuits and garbled circuits depends only on the maximum size of the working set with no additional overhead in the online phase.

In §4 we demonstrate that our implementation is substantially more efficient than previous frameworks. As applications we consider secure evaluation of the Hamming distance, fast multiplication, and computing the minimum. Moreover, we give performance results on securely computing the AES block cipher and for the first time on secure evaluation of the ultra-lightweight block cipher PRESENT.

As many previous frameworks do, we provide the source code of our implementation as open source software to foster future works and allow a fair performance comparison. The code is available for download at
`http://code.google.com/p/me-sfe/`.

1.2 Related Works

Faster secure two-party computation using garbled circuits (FastGC) [HEKM11] is the first software implementation of streamed garbled circuits. As optimization, FastGC implements the optimization of inputs that depend only on one party as described in [PSS09, Ker11]. The following applications are implemented within FastGC: AES, Hamming distance, and Levenstein / Smith-Waterman distance with applications in privacy-preserving genome and protein alignment. Subsequently, the FastGC framework was used for various other applications, including privacy-preserving biometric identification [HMEK11] and privacy-preserving set intersection [HEK12]. The FastGC framework was also extended to achieve stronger security guarantees [HKE12], and adapted for privacy-preserving applications on smartphones [HCE11a, HCE11b]. These and future applications of the FastGC framework (e.g., by using this framework for iris and fingerprint identification [BG11]) benefit from our improvements.

Many application scenarios require a low memory footprint, e.g., privacy-preserving applications on smartphones [HCE11a, HCE11b], generating garbled circuits in resource-restricted trusted hardware [JKSS10a], evaluating garbled circuits with a hardware accelerator [JKSS10b], or securely evaluating large functionalities in cloud computing scenarios [BNSS11].

Frameworks for secure two-party computation in the semi-honest adversaries setting can be classified into the traditional compilation paradigm, where the function to be computed is first compiled and the on-the-fly paradigm that generates circuits gate by gate from a library. The compilation paradigm is used in Fairplay [MNPS04, BNP08] and TASTY [HKS$^+$10]. The on-the-fly paradigm is used in the FastGC framework [HMEK11] and VMCrypt [Mal11]. We provide the best of both worlds by compiling and optimizing sub-circuits once and dynamically composing these sub-circuits on-the-fly.

A compilation technique for memory-efficient on-the-fly generation of circuits from Fairplay's high-level description language was proposed in [MLB12]. Alternatively, circuits can also be compiled from ANSI C programs as shown in [HFKV12]. The FastGC framework [HEKM11] was recently extended to read in a description of how circuits are composed from hard-coded circuit building blocks [MZE12]. However, these techniques do not minimize the amount of memory needed during secure evaluation of the circuit.

A large-scale garbled circuits-based framework for secure computations with security against stronger active (malicious) adversaries was presented recently in [KSS12]. This framework uses the compilation paradigm and exploits the high level of parallelism available in grid computing infrastructures by running multiple instances of a garbled circuit protocol in parallel – one on each machine. We extend their ideas for memory-efficient secure evaluation of garbled circuits and use multi-threading within a single instance of the garbled circuit protocol. For completeness we note that an alternative approach to practical actively secure secure two-party computations is [NNOB12] which is based on OT extensions instead of garbled circuits.

2. PRELIMINARIES

2.1 Yao's Garbled Circuit Protocol

In the following we give a brief summary of Yao's garbled circuit protocol, its optimizations, and oblivious transfer. For a more detailed description we refer to [Sch12, Chapter 2] and for a proof of security to [LP09a].

2.1.1 Yao's Garbled Circuit Protocol

Yao's garbled circuit protocol [Yao86] allows two parties, a server and a client, to jointly compute a function f represented as boolean circuit on their respective private inputs x and y. On a very high level, the server (sometimes called creator) creates an encrypted, called garbled, version of f which is then sent to the client (sometimes called evaluator) who evaluates the function under encryption. To encrypt the function, for each wire of f, the server assigns two random-looking wire labels that correspond to the values 0 and 1, respectively. Afterwards, the server obliviously sends exactly those wire labels to the client that correspond to their inputs. For client's inputs this is done with a sub-protocol, called oblivious transfer, such that the server does not learn the client's inputs (see below for details). Additionally, for each gate G_i of f, the server creates and sends to the client a garbled table T_i with the following property: given the wire labels for G_i's inputs, T_i allows to recover *only* the wire label of the corresponding output of G_i, but nothing else. Now, the client can use the wire labels of the inputs together with the garbled tables T_i to evaluate the garbled circuit gate by gate and obtains the labels of the output wires. For these output labels (and only for them) the client obtains mappings to the plain values 0 and 1 from the server which allow to recover $f(x, y)$.

The following optimizations of garbled circuits and oblivious transfer are used in today's most efficient implementations of Yao's protocol, including [HKS$^+$10, BG11, HEKM11, Mal11, KSS12] and our implementation.

2.1.2 Garbled Circuit Optimizations

The point-and-permute technique [NPS99] represents each wire label as a symmetric t-bit key and a permutation bit π, where t is a symmetric security parameter. The permutation bits of a gate's input wires are used as index to denote which table entry needs to be decrypted. The free XOR technique [KS08] allows to compute garbled linear gates, i.e., XOR and XNOR gates, without communication (no garbled table is needed) and only negligible computation. Thus, the dominating factor for the complexity of a circuit is the number of non-linear gates. Further, the garbled row reduction technique [NPS99, PSSW09] allows to reduce the garbled table by one entry.

2.1.3 Oblivious Transfer

In m-parallel Oblivious Transfer (OT) of ℓ-bit strings, denoted as OT_ℓ^m, the chooser inputs a vector of choice bits r_i, $i = 1 \ldots m$ and the sender inputs a vector of pairs of ℓ-bit strings $(x_0, x_1)_i$, $i = 1 \ldots n$. At the end of the protocol, the chooser learns the selected strings $x_{r_i,i}$, but nothing about the other strings $x_{1-r_i,i}$ whereas the sender learns nothing about the choices r_i.

In Yao's Garbled Circuit protocol $\ell = t+1$, whereas m corresponds to the number of input bits provided by the client which can be large. Using OT extensions of [IKNP03] it is possible to reduce a large number of OTs to a small number of only k OTs, where k is a security parameter. These remaining k base OTs are implemented with an efficient OT protocol which requires $\mathcal{O}(k)$ public-key operations, e.g., the OT protocol of [NP01].

In §3.1 we give implementation improvements for the base OTs of [NP01] and in §3.2 we show how the OT extension of [IKNP03] can be implemented with low memory footprint.

2.2 Minor Remark on the Choice of Security Parameters

As described in [HEKM11, Sect. 3.4], the implementation of FastGC uses SHA-1 as cryptographic hash function to encrypt the output wire labels of non-linear gates. Hence, the maximum security level that can be achieved is $t = 80$ bits. However, in the most recent implementation of FastGC [HEKM11] (version 0.1.1 released August 9, 2011), the last bit of the 80 bit wire labels is used as permutation bit for the point-and-permute technique of [NPS99] (cf. §2.1.2). As this permutation bit is known to the evaluator, the achieved symmetric security level is only 79 bits, but not 80 bits as stated in their paper. To correct this, we set Wire.labelBitLength=81 (actually we use 88 as internally a byte-oriented representation is used) to achieve a symmetric security level of exactly 80 bits. The longer wire labels result in a slight increase of communication. For the OT protocol of [NP01], implemented over $\mathbb{F}_P$ with a generator of order q, we use $|p| = 1024$ and $|q| = 160$ as these correspond to the current NIST recommendations for a symmetric security level of 80 bits.[1]

3. FASTER SECURE EVALUATION OF GARBLED CIRCUITS

We optimize several aspects of the FastGC framework [HEKM11] as described in the following.

We emphasize that our optimizations do not modify the underlying cryptographic protocols, but only the way they are implemented. Therefore, and because we work in the semi-honest setting, the proofs of security of the original protocols still hold for our optimizations.

[1] see `http://keylength.com`

3.1 Improved Base OTs

In order to improve the performance of the k base OTs with the OT protocol of [NP01], we split up the computationally intensive public key operations into multiple threads such that each of the N threads performs k/N base OTs (independent of each other). Furthermore, we tried to implement the base OTs over an elliptic curve instead of over $\mathbb{F}_P$ (the latter was used in the implementation of [HEKM11]). This resulted in a reduction of the communication complexity, but unfortunately no better runtimes. We assume that this is due to the additional overhead introduced by the Java VM. Our performance benchmarks for the improved base OT implementations are given in §4.1.

3.2 Extending OTs with Low Memory Footprint

A large number of m parallel OTs of ℓ-bit strings, OT_ℓ^m can be reduced to a small number of only k OTs of k-bit strings, OT_k^k, using OT extensions of [IKNP03] as implemented in [HEKM11]. The original protocol of [IKNP03] needs beyond the messages of the base OTs only two messages, but memory linear in m.

We reduce the memory requirement of this protocol by re-ordering its messages as follows: The OT extension construction of [IKNP03] proceeds in two steps: First, the large number of m parallel OTs of short ℓ-bit strings are reduced to k parallel OTs of long m-bit strings, cf. [IKNP03, Fig. 1]. These OTs are implemented using k parallel OTs of short k-bit keys that are then stretched into longer m-bit masks using a pseudo-random generator (PRG), cf. [IKNP03, Fig. 3].

A very efficient and standard way to implement a PRG is to successively apply a pseudo-random permutation (PRP) on a counter, i.e., $\mathrm{PRG}(k) = \mathrm{PRP}_k(0)||\mathrm{PRP}_k(1)||\dots$. In practice, the PRP can be instantiated with a block cipher which operates on blocks of M bits, e.g., in our implementation we use AES-128 with $M = 128$. Now, in order to reduce the memory footprint, the OT extension construction of [IKNP03] can easily be split into smaller blocks where each block performs M parallel OTs. The overall protocol is shown in Fig. 1. W.l.o.g. and to simplify presentation we assume that m is a multiple of the block size, $m = BM$ (otherwise, the last messages are shorter) and that $\log B \leq M$. $\mathbf{T}$ denotes an $M \times k$ bit matrix, $\mathbf{T}^i$ its i-th column and $\mathbf{T}_j$ its j-th row. $G : \{0,1\}^M \times \{0,1\}^k \to \{0,1\}^M$ is a PRP; $H : \{0,1\}^{\lceil \log m \rceil} \times \{0,1\}^k \to \{0,1\}^\ell$ is a random oracle which can be implemented using a cryptographic hash function (in our implementation we use SHA-1).

Overall, our modified protocol can be seen as operating on a stream of data which is processed in chunks of size M.

Security and Correctness. The only modification that we applied to the original construction of [IKNP03] is that we re-order the messages sent into B rounds of communication. For semi-honest parties, this does not reveal any additional information. Therefore, the correctness and security of our optimized protocol in the semi-honest setting directly follows from the correctness and security of the original construction as proven in [IKNP03].

Performance. The computation and communication complexity of our protocol is exactly the same as that of the original construction of [IKNP03]. In contrast to the implementation in [HEKM11] which has a constant number of communication rounds but requires memory linear in the total number of OTs m, the memory consumption of our protocol is constant (for fixed block size M), but requires m/M rounds of communication. We give performance benchmarks for the improved OT extensions in §4.1.

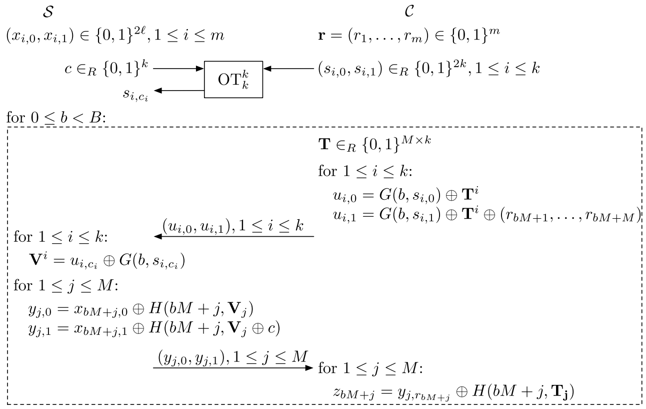
$\mathcal{S}$
$\mathcal{C}$
$(x_{i,0}, x_{i,1}) \in \{0,1\}^{2\ell}, 1 \leq i \leq m$
$\mathbf{r} = (r_1, \dots, r_m) \in \{0,1\}^m$
$c \in_R \{0,1\}^k$
OT_k^k
$(s_{i,0}, s_{i,1}) \in_R \{0,1\}^{2k}, 1 \leq i \leq k$
s_{i,c_i}
for $0 \leq b < B$:
$\mathbf{T} \in_R \{0,1\}^{M \times k}$
for $1 \leq i \leq k$:
$u_{i,0} = G(b, s_{i,0}) \oplus \mathbf{T}^i$
$u_{i,1} = G(b, s_{i,1}) \oplus \mathbf{T}^i \oplus (r_{bM+1}, \dots, r_{bM+M})$
$(u_{i,0}, u_{i,1}), 1 \leq i \leq k$
for $1 \leq i \leq k$:
$\mathbf{V}^i = u_{i,c_i} \oplus G(b, s_{i,c_i})$
for $1 \leq j \leq M$:
$y_{j,0} = x_{bM+j,0} \oplus H(bM + j, \mathbf{V}_j)$
$y_{j,1} = x_{bM+j,1} \oplus H(bM + j, \mathbf{V}_j \oplus c)$
$(y_{j,0}, y_{j,1}), 1 \leq j \leq M$
for $1 \leq j \leq M$:
$z_{bM+j} = y_{j,r_{bM+j}} \oplus H(bM + j, \mathbf{T_j})$

Figure 1: Extension of OT_ℓ^m with low memory footprint. W.l.o.g. $m = BM$ for B blocks of size M. G is a pseudo-random permutation and H is a random oracle.

3.3 Streaming Circuits and Garbled Circuits with Low Memory Footprint

As described in §1 and §1.2, previous frameworks for secure two-party computation in the semi-honest adversaries setting store the outgoing wire labels of each gate in memory and hence require memory linear in the size of the evaluated circuit. More recent frameworks [Mal11, HEKM11, MZE12] require memory only linear in the size of the sub-circuits, but these frameworks suffer from the low performance of memory management of many small objects (one object for each gate) and garbage collection.

During creation and evaluation of a (sub) circuit, only those wires need to be held in memory that are used in the future, i.e., are either an output wire of the circuit or used as input wires into a later gate. This set of wires is called the *working set*. Thus, the minimum size of the memory required to compute the circuit is defined by the maximum working set. As described and implemented in [KSS12], one strategy to keep track of the working set is to annotate to each wire a usage counter that is decremented on each use. When the counter reaches zero, the wire is deleted from the memory. This is similar to the way sub-circuits are dynamically constructed and deconstructed in VMCrypt [Mal11] and impedes additional overhead in the online phase.

We use a different approach where we shift the management of the working set from the online phase to the compilation phase in order to keep the online phase as lean as possible. In particular, the online phase of our approach neither requires a usage counter nor to allocate and later on free memory for each gate. During the compilation phase, the compiler determines the maximum size S of the working set and stores this along with the circuit description. The compiler allocates a slot ID (starting from 0) to each input wire of the circuit. Then, the circuit description is generated as an ordered list of gates where each gate is described as a tuple (output slot ID; input slot IDs; truth table). Whenever a slot ID is used for the last time, it is added to a list of available slot IDs and can be re-used as the output slot ID of a later gate. Note that all this is done in the offline (compilation) phase. In the online phase, an array with S slots is allocated where each slot can hold a wire label. Then, the gates are read one-by-one from the circuit description: the gate's input labels are taken from the slots given by its input slot IDs and the output label is stored to the slot specified by the gate's output slot ID. We internally keep a counter of the gate ID which is used to construct the garbled tables.

We emphasize that the compilation of a function has to be done just once, since the resulting circuit is independent from the inputs and can therefore be reused.

In future work, the compiler can be extended to re-arrange the order of the gates in order to reduce the size of the working set as described in [JKSS10b]. However, as noted in [KSS12], determining a topologic order of the circuit with the minimum size of the working set is known to be an NP-complete problem.

We further note that in principle generation of garbled circuits can be implemented such that the amount of memory is constant by pseudo-randomly deriving the wire labels from the gate ID as described in [JKSS10a]. However, this is essentially a time-memory tradeoff and cannot easily be combined with the highly efficient free XOR technique.

3.4 Sub-Circuit Compilation

A design goal of our framework was to keep the online phase of the circuit evaluation as lean as possible. Due to the topological ordering of the circuits the engine never has to hold more than one gate description in memory. Once a gate is processed, the information can be discarded (except for the intermediate wire labels). The circuit evaluation engine reads the circuit description from a file with a format similar to Fairplay. In order to re-use sub-circuits we use slot IDs to index the wires in the sub-circuit. A slot ID can be seen as a virtual register that can hold a wire label and can also be re-used. For the gate ID, which needs to be unique in the overall circuit as it is used for encrypting the non-linear gates, we use a counter which is incremented for each non-linear gate. A gate is described by its output slot ID, its input slot IDs, and its truth-table. An example is shown in Fig. 2 which describes the one-bit comparison circuit from [KSS09]. We also support a binary file format which is more efficient to read by our engine.

The circuit to be computed often consists of several calls to the same sub-function. For instance, AES consists of 160 S-Box, 10 AddRoundKey, and 9 MixColumns calls. Our framework allows for sub-circuits to be reused in a similar fashion like [HEKM11], except that we do not instantiate a new gate object in each invocation of the sub-circuit. Overall, for AES we do not create the entire circuit with 24,720 gates, but only 3 sub-circuits for the sub-functions listed above with a total of just 803 gates. As in our implementation the creation of the gate ID is decoupled from the circuit definition, we ensure that in every reuse of a sub-circuit all gates have a unique gate ID, and therefore the security of the underlying garbled circuit protocol, as proven in [LP09a, PSSW09], still holds.

We provide a compiler that converts circuits described in the format of [HEKM11] into our format.

Figure 2: A one-bit comparison circuit. Comments (from // on) are not part of the input.

```
inputsCreator: 0    // creator's input is r0
inputsEvaluator: 1  // evaluator's input is r1
outputsCreator:
outputsEvaluator: 0 // evaluator's output is r0
numberOfRegisters: 2
numberOfGates: 2
0;1,0;1             // r0 = r1 and r0
0;1,0;6             // r0 = r1 xor r0
```

3.5 Caching of Circuits and Communication

In order to improve the performance of garbled circuits evaluation we cache both, circuit descriptions and network packets during garbled circuit streaming resulting in a corresponding time-memory trade-off as described next.

In some circuits, the same sub-circuits are reused many times (cf. §3.4). Instead of reading the description of the sub-circuit from a file on every instantiation (or re-generating it as implemented in previous frameworks), we optionally cache its description once in memory. The memory consumption is 32 bytes per cached gate.

Sending the creator's input wire labels and the garbled tables straight after creation (as implemented in [HEKM11]) leads to an inefficient use of the network because of small packet sizes and an unnecessary large number of packets. By using fixed sized buffers on the communication channels we greatly improve the performance of the network usage.

In our benchmarks in §4 we use circuit caching and network buffers of size 9,000 byte.

4. PERFORMANCE BENCHMARKS AND APPLICATIONS

In the following we show that the implementation of our improvements described in the previous section results in substantially better performance than previous frameworks.

4.1 Oblivious Transfers

The following performance benchmarks were performed on two Apple computers with a dual core processor each (Intel Core i5 2.5GHz and Core i7 1.8GHz) running MacOS X 10.7.4 and Java 1.6.0_33, connected via 802.11n WIFI.

We observed that because of the JAVA just in time compiler, the runtime decreases in the first few runs due to compiler optimizations. Therefore, we executed each protocol 1,000 times and took the average. All benchmarks were executed with the default JAVA VM parameter.

The performance of our improved implementation for the base OTs (cf. §3.1) in comparison with the original implementation of [HEKM11] run in exactly the same setting is shown in Table 2. Due to the additional overhead for thread management, our multi-threaded implementation with one thread is slightly slower than the single-threaded one. However, when running with 4 threads on the dual core processors, our multi-threaded base OTs take 0.15 seconds, an improvement by factor 2 over the single-threaded version. We emphasize that we use multi-threading *only* for the base OTs and that for small circuit sizes this time is much longer than the online time (cf. Table 3).

Table 2: Comparison of base OT implementations.

single threaded	time
over $\mathbb{F}_P$ [HEKM11]	286 ms (100%)
over EC	560 ms (196%)
multi threaded (over $\mathbb{F}_P$)	time
1 thread	314 ms (110%)
2 threads	182 ms (64%)
4 threads	153 ms (53%)

For the OT extensions (independent of the performance of the base OTs), our improved implementation of the protocol of §3.2 can evaluate about 400,000 OTs per second, i.e., 2.5μs per OT, a factor 6 improvement over the 15μs reported in [HEKM11] (which used faster Intel Core Duos E8400 3GHz and a faster local area network).[2] We emphasize that this optimization is very beneficial for applications where the circuit has many inputs, e.g., for converting from homomorphic encryption to garbled circuits and subsequently finding the minimum, a very common building block in privacy-preserving protocols for biometric matching [HKS+10, HMEK11, BG11] (cf. the example we benchmark in §4.2.4).

4.2 Online Time

The following benchmarks were executed on a single iMac A1311 with an Intel Core i3 3GHz processor using the loopback network interface. We measured the *online* time, that is the time from after the connection is established and the base OTs are done until the end of the protocol. This time includes the time for the OT extensions and streaming, i.e., creating, transferring, and evaluating, the garbled circuit.

Our improved implementation evaluates about 500,000 non-linear gates per second (2μs per gate) on the same host (setting as described above) and about 350,000 non-linear gates per second (3μs per gate) over WLAN (setting as described in §4.1). In contrast, [HEKM11] reported 96,000 non-linear gates per second (10μs per gate) over a LAN.[3]

In the following we show that the online time of our framework when evaluated on the same host as described above (i.e., assuming an ideal network) is up to 10 times faster than that of previous frameworks; we give a comparison for small circuits with FastGC [HEKM11] in §4.2.1 and §4.2.2 (using optimized circuit constructions), for medium size circuits with FastGC [HEKM11] and TASTY [HKS+10] in §4.2.3, and for large circuits with FastGC [HEKM11] and VMCrypt in §4.2.4. The comparison between our implementation and FastGC [HEKM11], both executed on exactly the same machine, is summarized in Table 3.

4.2.1 Hamming Weight

Some applications, e.g., privacy-preserving face recognition [OPJM10], require to securely compute the Hamming distance $d_H(\vec{a}, \vec{b})$ between two ℓ-bit strings $\vec{a}$, $\vec{b}$. As shown in [HEKM11], this can be done by XORing $\vec{a}$ and $\vec{b}$ bitwise and computing its Hamming weight $h(\cdot)$, i.e., the number of "ones" in its binary representation: $d_H(\vec{a}, \vec{b}) = h(\vec{a} \oplus \vec{b})$.

Original Hamming Circuit. The authors of [HEKM11] propose to use a tree of addition circuits which requires approximately $\sum_{i=0}^{\lceil \log_2 \ell \rceil} \ell \frac{i}{2^i} = \ell(2 - \frac{\lceil \log_2 \ell \rceil + 2}{2^{\lceil \log_2 \ell \rceil}}) \approx 2\ell - \log_2 \ell$ non-linear gates. For the example of $\ell = 900$ given in their paper this yields approximately 1,790 non-linear gates.

Improved Hamming Circuit. We use the optimized Hamming weight circuit of [BP06] with size $\ell - h(\ell)$ non-linear gates, where $h(\ell)$ is the Hamming weight of ℓ. For $\ell = 900$ this yields $900 - h((1110000100)_2) = 896$ non-linear gates.

The resulting circuit sizes and performance results are shown in Table 3. Put together, the improved Hamming circuit together with our improved implementation is more than 10 times faster than the original Hamming circuit evaluated with FastGC [HEKM11] (6 w/o circuit optimization).

4.2.2 Block Ciphers

Oblivious evaluation of a block cipher where one party provides the key and the other party provides the message and obtains the ciphertext has many applications as summarized in [PSSW09]. These include oblivious pseudorandom functions (OPRFs) with applications to secure keyword searching [FIPR05] or secure set intersection [JL09], blind MACs, and blind encryption.

As noted in [HEKM11], the key schedule of the block cipher does not need to be computed securely within the garbled circuit. Instead, the party that knows the key can run the key schedule on the plain key data to expand it and provide the expanded key as input to the protocol.

AES. Secure evaluation of AES is commonly used as performance benchmark for secure computation frameworks, e.g., [PSSW09, HKS+10, HEKM11, KSS12].

Original AES Circuit. Excluding the key schedule, AES-128 consists of 10 rounds where in each round 16 S-boxes are evaluated. As shown in [HEKM11, Sect. 7], all other operations, e.g., MixColumns and AddRoundKey, can be performed using only free XOR gates. The S-box presented in [HEKM11] has 58 non-linear gates resulting in $58 \cdot 10 \cdot 16 =$ 9,280 non-linear gates for AES.

Improved AES Circuit. Instead, we implemented the S-box of [BP10] which consists of only 32 non-linear gates resulting in a total of $32 \cdot 10 \cdot 16 =$ 5,120 non-linear gates for AES. We note that the AES circuit implemented in [KSS12] uses the same S-box and has 9,100 non-linear gates, but including the key schedule.

PRESENT. For the applications mentioned above, it might be sufficient to use a block cipher that does not provide the strong security guarantees of AES, but is more efficient to evaluate. An example for such an ultra-lightweight block cipher is PRESENT with a block length of 64 bit and an 80 bit key. PRESENT consists of 31 rounds where in each round a 4-bit S-box is applied 16 times in parallel. We implemented PRESENT using the S-box representation of [CHM11, CHM12] which has 4 non-linear gates. Overall, PRESENT requires $31 \cdot 16 \cdot 4 =$ 1,984 non-linear gates.

[2] For completeness we note that [NNOB12] claim an actively-secure OT extension that can be implemented at about 500,000 OTs per second based on unpublished optimizations (cf. Appendix A and E in the full version of their paper).

[3] In the malicious setting, [KSS09] report 82,000 non-linear gates per second (12μs per gate) on a cluster and [NNOB12] 20,000 gates per second (50μs per gate) over an intranet.

Table 3: Comparison of circuit sizes and performance when run on the same machine.

	Circuit	non-linear gates		FastGC [HEKM11]		Our Implementation	
§4.2.1	Original Hamming	1,793	(100%)	64 ms	(100%)	8 ms	(13%)
	Improved Hamming	896	(44%)	39 ms	(61%)	6 ms	(9%)
§4.2.2	Original AES	9,280	(100%)	204 ms	(100%)	27 ms	(13%)
	Improved AES	5,120	(55%)	113 ms	(55%)	16 ms	(8%)
	PRESENT	1,984	(21%)	53 ms	(26%)	7 ms	(3%)
§4.2.3	Fast Multiplication	17,973		499 ms	(100%)	45 ms	(9%)
§4.2.4	Minimum	40,000,000		1,250 s	(100%)	a) 138 s	(11%)
						b) 272 s	(22%)
						c) 128 s	(10%)

Comparison. The size of the AES and PRESENT circuits and their performance comparison are shown in Table 3. The improved AES circuit evaluated with our improved implementation is about 12 times faster than the original AES circuit evaluated on the FastGC engine [HEKM11]. Due to its smaller gate count, PRESENT is almost 4 times faster than the original AES circuit and twice as fast as the improved AES circuit.

4.2.3 Fast Multiplication

To compare our implementation with FastGC [HEKM11] and TASTY [HKS$^+$10] for circuits of medium size, we implemented secure multiplication using the fast multiplication method of Karatsuba and Ofman [KO62]. For multiplication of two 128 bit numbers, this circuit has 17,973 non-linear gates and took 45 ms online time to evaluate in our optimized framework.

The same circuit implemented in FastGC [HEKM11] and run on the same machine took 499 ms, i.e., more than 10 times longer. This result supports the fact that the improved online time of our implementation, in particular for medium size circuits, is substantially faster than that of [HEKM11], even without circuit-specific optimizations.

According to [HKS$^+$10, Fig. 7], TASTY takes approximately 4,000 ms setup time and 700 ms online time to evaluate the same circuit on two desktop PCs with Intel Core 2 Duo CPU (E6850) running at 3GHz connected via Gigabit Ethernet. We emphasize that TASTY doesn't use streaming, but pre-computes the OT extensions and generates and transfers the garbled circuit already in the setup phase; hence, the online time in TASTY consists only of the very efficient online OTs of Beaver's construction [Bea95] and garbled circuit evaluation whereas our online time includes the computationally more expensive OT extensions and creating and transferring the garbled circuit which is minimized due to streaming. Overall, for this application, the online time of our improved framework is faster than TASTY by a factor of about 16 times, whereas our setup time is as low as 153 ms for the base OTs (cf. Table 2), i.e., 26 times faster.

4.2.4 Minimum

To compare our implementation with FastGC [HEKM11] and VMCrypt [Mal11] for circuits of large size, we implemented a circuit to compute the minimum of 10^6 20-bit numbers (half of the numbers are input by the server and the other half by the client) using a circuit similar to the one described in [Mal11, Fig. 2]. We use the OT extension with low memory footprint described in §3.2 such that the total memory consumption stays linear in the size of the subset and not in the order of the total number of inputs. The overall circuit has $2 \cdot (10^6 - 1) \cdot 20 \approx 40{,}000{,}000$ non-linear gates. There are different approaches to compute this functionality which demonstrate the flexibility of our system that allows to choose a trade-off between execution time (shown in Table 3) and memory consumption (shown in Table 4):

a) The first solution has the lowest OT overhead but requires the maximum amount of memory (800 MB) by doing the OTs for all inputs first and then evaluating a large circuit with a working set of size $2 * 10^7$. The runtime is 138 s.

b) As another extreme we can iteratively compare one input each at a time with the previously found minimum. This approach needs the minimal amount of memory (18.4 MB) but the maximal total runtime (272 s) as the OT protocol introduces a significant overhead.

c) Our framework allows to choose an intermediate approach where we iteratively compute the minimum of a subset of 500 inputs and the minimum of the previous iteration. This sub-circuit has 19,960 non-linear gates and a working set of 10,022 labels and is small enough to be cached in memory – the total memory requirement is 21.5 MB and the total runtime is 128 s.

When evaluating the same circuit on the same machine with FastGC [HEKM11], this took approximately 1,250 s and 189 MB memory, i.e., more than 9 times longer and 10 times more memory compared to our approach c).

According to [Mal11, Fig. 8], VMCrypt takes 44.5 min on a slower CPU (Thinkpad X301 laptop with 3 GB RAM and a 1.6 GHz Intel Core2 Duo processor running Ubuntu Linux over the loopback interface), i.e., about 10 times longer than our approach c), already considering the fact that our CPU is about twice as fast. We assume that our improved performance stems mainly from the fact that we do not allocate and free many small objects.

4.3 Memory Consumption

Measuring the memory consumption of a Java program is fuzzy, since released objects remain on the heap until the garbage collector deletes them and the garbage collector itself is managed by the Java virtual machine. Thus, the heap contains not only the currently used objects but also already released ones and therefore the size of the heap will be greater or equal to the size of the currently used objects. We measured the maximum heap consumption of every protocol, since this gives an indication of how much memory is needed for a runtime optimal execution. The protocols might run with smaller heap sizes, but then the virtual machine has to invoke the garbage collection more often which results in longer runtimes. Table 4 shows the maximum size of the heap during the execution for all protocols.

The memory consumption of FastGC is linear in the total number of gates whereas in our implementation it is linear in the size of the working set. Although for circuits that are divisible into many small sub-circuits like AES or PRESENT the memory consumption of both implementations is almost the same; the memory efficiency advantage of our implementation becomes obvious for larger circuits: Compared to FastGC we achieve a memory consumption reduction by factor 5 for Fast Multiplication and by factor 8 for Minimum for approach c) – by combining the repeated use of a sub-circuit and then just holding the working set in memory, we can evaluate the Minimum circuit with almost 140 million gates using only 21.5 MB of memory.

Acknowledgments. The authors would like to thank René Peralta for helpful discussions on efficient circuit constructions. The first author was supported by Australian Research Council grant DP0984063, by an Adelaide Scholarship International, and a Supplementary Scholarship by the Defence Systems Innovation Centre. The second author was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE and by the Hessian LOEWE excellence initiative within CASED.

5. REFERENCES

[Bea95] D. Beaver. Precomputing oblivious transfer. In *Advances in Cryptology – CRYPTO'95*, volume 963 of *LNCS*, pages 97–109. Springer, 1995.

[BG11] M. Blanton and P. Gasti. Secure and efficient protocols for iris and fingerprint identification. In *European Symposium on Research in Computer Security (ESORICS'11)*, volume 6879 of *LNCS*, pages 190–209. Springer, 2011.

[BNP08] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security (CCS'08)*, pages 257–266. ACM, 2008.

[BNSS11] S. Bugiel, S. Nürnberger, A.-R. Sadeghi, and T. Schneider. Twin Clouds: Secure cloud computing with low latency. In *Communications and Multimedia Security Conference (CMS'11)*, volume 7025 of *LNCS*, pages 32–44. Springer, 2011.

[BP06] J. Boyar and R. Peralta. Concrete multiplicative complexity of symmetric functions. In *Mathematical Foundations of Computer Science (MFCS'06)*, volume 4162 of *LNCS*, pages 179–189. Springer, 2006.

[BP10] J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. In *Symposium on Experimental Algorithms (SOA'10)*, volume 6049 of *LNCS*, pages 178–189. Springer, 2010.

[CHM11] N. T. Courtois, D. Hulme, and T. Mourouzis. Solving circuit optimisation problems in cryptography and cryptanalysis. In *2nd IMA Conference Mathematics in Defence*, 2011.

[CHM12] N. T. Courtois, D. Hulme, and T. Mourouzis. Solving circuit optimisation problems in cryptography and cryptanalysis. In *Special-purpose Hardware for Attacking Cryptographic Systems (SHARCS'12)*, pages 179–191, 2012.

[FIPR05] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography Conference (TCC'05)*, volume 3378 of *LNCS*, pages 303–324. Springer, 2005.

[HCE11a] Y. Huang, P. Chapman, and D. Evans. Privacy-preserving applications on smartphones. In *USENIX Workshop on Hot Topics in Security (HotSec'11)*, 2011.

[HCE11b] Y. Huang, P. Chapman, and D. Evans. Secure computation on mobile devices, 2011. Poster at IEEE Symposium on Security and Privacy.

[HEK12] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Network and Distributed Security Symposium (NDSS'12)*. The Internet Society, 2012.

[HEKM11] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium (Security'11)*, pages 539–554. USENIX, 2011.

[HFKV12] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ANSI C. In *ACM Conference on Computer and Communications Security (CCS'12)*, pages 772–783. ACM, 2012.

[HKE12] Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE Symposium on Security and Privacy*. IEEE, 2012.

[HKS+10] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM Conference on Computer and Communications Security (CCS'10)*, pages 451–462. ACM, 2010.

[HMEK11] Y. Huang, L. Malka, D. Evans, and J. Katz. Efficient privacy-preserving biometric identification. In *Network and Distributed System Security (NDSS'11)*. The Internet Society, 2011.

[IKNP03] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[JKSS10a] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Embedded SFE: Offloading server and network using hardware tokens. In *Financial Cryptography and Data Security (FC'10)*, volume 6052 of *LNCS*, pages 207–221. Springer, 2010.

[JKSS10b] K. Järvinen, V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Garbled circuits for leakage-resilience: Hardware implementation

Table 4: Comparison of the memory consumption.

Circuit	tot. number of gates	gates in memory	working set	FastGC [HEKM11]	Our Impl.
Original Hamming	7,163	7,163	1,800	28.5 MB	17 MB
Improved Hamming	5,362	5,362	1,800	26 MB	17 MB
Original AES	36,720	878	256	20.3 MB	16.9 MB
Improved AES	24,720	803	256	20.2 MB	18.7 MB
PRESENT	8,496	77	128	18.9 MB	18.6 MB
Fast Multiplication	57,072	57,072	1,074	74.4 MB	15 MB
Minimum	137,999,862	a) 137,999,862 b) 414 c) 69,000	a) 20,000,002 b) 104 c) 10,022	189 MB	a) 799.7 MB b) 18.4 MB c) 21.5 MB

and evaluation of one-time programs. In *Cryptographic Hardware and Embedded Systems (CHES'10)*, volume 6225 of *LNCS*, pages 383–397. Springer, 2010.

[JL09] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In *Theory of Cryptography Conference (TCC'09)*, volume 5444 of *LNCS*, pages 577–594. Springer, 2009.

[Ker11] F. Kerschbaum. Automatically optimizing secure computation. In *ACM Computer and Communications Security (CCS'11)*, pages 703–714. ACM, 2011.

[KO62] A. A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *SSSR Academy of Sciences*, 145:293–294, 1962.

[KS08] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages and Programming (ICALP'08)*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[KSS09] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security (CANS'09)*, LNCS. Springer, 2009.

[KSS12] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium (Security'12)*. USENIX, 2012.

[LP09a] Y. Lindell and B. Pinkas. A proof of Yao's protocol for secure two-party computation. *Journal of Cryptology*, 22(2):161–188, 2009.

[LP09b] Y. Lindell and B. Pinkas. Secure multiparty computation for privacy-preserving data mining. *Journal of Privacy and Confidentiality*, 1(1):59–98, 2009.

[Mal11] L. Malka. VMCrypt - modular software architecture for scalable secure computation. In *ACM Conference on Computer and Communications Security (CCS'11)*, pages 715–724. ACM, 2011.

[MLB12] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Financial Cryptography and Data Security (FC'12)*, LNCS. Springer, 2012.

[MNPS04] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.

[MZE12] W. Melicher, S. Zahur, and D. Evans. An intermediate language for garbled circuits. Poster at IEEE Symposium on Security and Privacy, 2012.

[NNOB12] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO'12*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.

[NP01] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *ACM-SIAM Symposium On Discrete Algorithms (SODA'01)*, pages 448–457. Society for Industrial and Applied Mathematics, 2001.

[NPS99] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *ACM Conference on Electronic Commerce*, pages 129–139. ACM, 1999.

[OPJM10] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovich. SCiFI - a system for secure face identification. In *IEEE Symposium on Security and Privacy*, pages 239–254. IEEE, 2010.

[PSS09] A. Paus, A.-R. Sadeghi, and T. Schneider. Practical secure evaluation of semi-private functions. In *Applied Cryptography and Network Security (ACNS'09)*, volume 5536 of *LNCS*, pages 89–106. Springer, 2009.

[PSSW09] B. Pinkas, T. Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

[Sch12] T. Schneider. *Engineering Secure Two-Party Computation Protocols: Design, Optimization, and Applications of Efficient Secure Function Evaluation*. Springer, 2012.

[Yao86] A. C. Yao. How to generate and exchange secrets. In *Foundations of Computer Science (FOCS'86)*, pages 162–167. IEEE, 1986.